Who, What, Where and When

- **Instructor: Charles Hughes;**
  HEC-247C
  charles.hughes@ucf.edu
  (e-mail is a good way to get me)
  Use Subject: COT6410
  Office Hours: TR 3:15PM-4:30PM
- **Web Page: http://www.cs.ucf.edu/courses/cot6410/Spring2019**
- **Meetings: TR 1:30PM-2:45PM, HEC-103;**
  28 periods, each 75 minutes long.
  Final Exam (Tuesday, April 30 from 1:00PM to 3:50PM) is separate from class meetings
- **GTA:** Harish Raviprakash; harishr@knights.ucf.edu
  Use Subject: COT6410
  Office Hours: MW 1:30PM-3:00PM; Room: HEC-308

Text Material

- References:
- Cooper, Computability Theory 2nd Ed., Chapman-Hall/CRC Mathematics Series, 2003.
- Garey&Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., 1979.
- Davis, Sigal&Weyuker, Computability, Complexity and Languages 2nd Ed., Acad. Press (Morgan Kaufmann), 1994.
- Papadimitriou & Lewis, Elements of the Theory of Computation, Prentice-Hall, 1997.
- Bernard Moret, The Theory of Computation, Addison-Wesley, 1998.
- Hopcroft, Motwani&Ullman, Intro to Automata Theory, Languages and Computation 3rd Ed., Prentice-Hall, 2006.
- Oded Goldreich, Computational Complexity: A Conceptual Approach, Cambridge University Press, 2008.
- Draft available at http://www.wisdom.weizmann.ac.il/~/oded/cc-drafts.html
- Oded Goldreich, P, NP, and NP-Completeness: The Basics of Complexity Theory, Cambridge University Press, 2010.
- Draft available at http://www.wisdom.weizmann.ac.il/~/oded/bc-drafts.html
- Arora&Barak, Computational Complexity: A Modern Approach, Cambridge University Press, 2009.
- Draft available at http://www.cs.princeton.edu/theory/complexity/
- Sipser, Introduction to the Theory of Computation 3rd Ed., Cengage Learning, 2013.

Goals of Course

- Introduce Computability and Complexity Theory, including
  - Review background on automata and formal languages
  - Basic notions in theory of computation
    - Algorithms and effective procedures
    - Decision and optimization problems
    - Decision problems have yes/no answer to each instance
  - Limits of computation
    - Turing Machines and other equivalent models
    - Determinism and non-determinism
    - Undecidable problems
    - The technique of reducibility; The ubiquity of undecidability (Rice’s Theorem)
    - The notions of semi-decidable (re) and of co-re sets
  - Complexity theory
    - Order notation (quick review)
    - Polynomial reducibility
    - Time complexity, the sets P, NP, co-NP, NP-complete, NP-hard, etc., and the question does P=NP? Sets in NP and NP-Complete.
    - Gadgets and other reduction techniques

Expected Outcomes

- You will gain a solid understanding of various types of computational models and their relations to one another.
- You will have a strong sense of the limits that are imposed by the very nature of computation, and the ubiquity of unsolvable problems throughout CS.
- You will understand the notion of computational complexity and especially of the classes of problems known as P, NP, co-NP, NP-complete and NP-Hard.
- You will (hopefully) come away with stronger formal proof skills and a better appreciation of the importance of discrete mathematics to all aspects of CS.

Keeping Up

- I expect you to visit the course web site regularly (preferably daily) to see if changes have been made or material has been added.
- Attendance is preferred, although I do not take roll.
- I do, however, ask lots of questions in class and give lots of hints about the kinds of questions I will ask on exams. It would be a shame to miss the hints, or to fail to impress me with your insightful in-class answers.
- You are responsible for all material covered in class, whether in the notes or not.

Rules to Abide By

- Do Your Own Work
  - When you turn in an assignment, you are implicitly telling me that these are the fruits of your labor. Do not copy anyone else's homework or let anyone else copy yours. In contrast, working together to understand lecture material and solutions to problems not posed as assignments is encouraged.
- Late Assignments
  - I will accept no late assignments, except under very unusual conditions, and those exceptions must be arranged with me in advance unless associated with some tragic event.
- Exams
  - No communication during exams, except with me or a designated proctor, will be tolerated. A single offense will lead to termination of your participation in the class, and the assignment of a failing grade.

Grading

- Grading of Assignments and Exams
  - I will endeavor to return each exam within a week of its taking place and each assignment within a week of its due date.
- Exam Weights
  - The weights of exams will be adjusted to your personal benefits, as I weigh the exam you do well in more than one in which you do less well.

Important Dates

- Midterm – Tues., March 5 (tentative)
- Spring Break – March11-16
- Withdraw Deadline – Wednesday, March 20
- Final – Tues., April 30, 1:00PM–3:50PM

Evaluation (tentative)

- Mid Term – 125 points ; Final – 200 points
- Assignments – 75 points;
  Paper and Presentation – 75 points
- Extra – 25 points used to increase weight of exams or maybe paper/presentation, always to your benefit
- Total Available: 500 points
- Grading will be A >= 90%, B+ >= 85%,
  B >= 80%, C+ >= 75%, C >= 70%,
  D >= 50%, F < 50% (Minuses might be used)

Decision Problems

- A set of input data items (input "instances” or domain)
- Each input data item defines a question with an answer Yes/No or True/False or 1/0.
- A decision problem can be viewed as a relation between its domain and its binary range
- A decision problem can also be viewed as a partition of the input domain into those that give rise to true instances and those that give rise to false instances.
- In each case, we seek an algorithmic solution (in the form of a predicate) or a proof that none exists
- When an algorithmic solution exists, we seek an efficient algorithm, or proofs of the problem’s inherent complexity

UNIVERSE OF DISCOURSE
USUALLY STRINGS OR NATURAL NUMBERS

DECISION PROBLEMS

S
Subset of interest, maybe with ordered elements

For some element, x, is x in S?

Question: How many subsets of Natural Numbers are there?

Example 1: S is set of Primes and x is a natural number; is x in S (is x a prime)?
Example 2: S is an undirected graph (pairs for neighbors); is S 3-colorable?
Example 3: S is a program in C; is S syntactically correct?
Example 4: S is program in C; does S halt on all input?
Example 5: S is a set of strings; is the language S Regular, Context-Free, … ?

Recognizer and Generators

1. When we discuss languages and classes of languages, we discuss recognizers and generators
2. A recognizer for a specific language is a program or computational model that differentiates members from non-members of the given language
3. A portion of the job of a compiler is to check to see if an input is a legitimate member of some specific programming language – we refer to this as a syntactic recognizer
4. A generator for a specific language is a program that generates all and only members of the given language
5. In general, it is not individual languages that interest us, but rather classes of languages that are definable by some specific class of recognizers or generators
6. One type of recognizer is called an automata and there are multiple classes of automata
7. One type of generator is called a grammar and there are multiple classes of grammars
8. Our first journey will be a review of automata and grammars

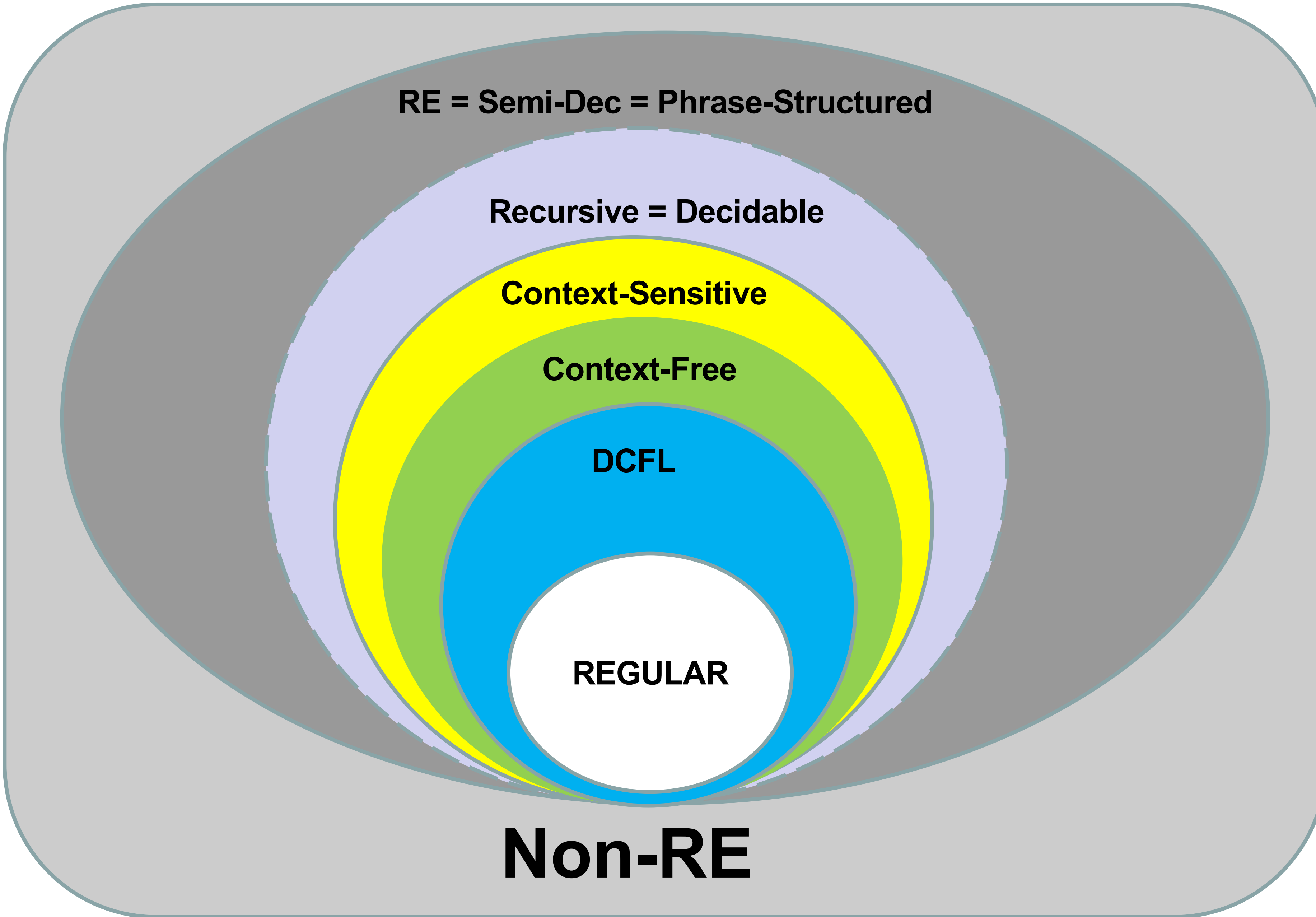
Alphabets and Strings

- DEFINITION 1. An *alphabet* Σ is a finite, non-empty set of abstract symbols.
- DEFINITION 2. Σ^*, the set of <u>all strings over the alphabet, S</u>, is given inductively as follows.
  - Basis: $\lambda \in \Sigma^*$ (the *null string* is denoted by λ, it is the <u>string of length 0</u>, that is $|\lambda| = 0$) [text uses ε but I avoid that as hate saying $\varepsilon \in A$; it's really confusing when manually written]
    $\forall a \in \Sigma$, $a \in \Sigma^*$ (the members of S are <u>strings of length 1</u>, $|a| = 1$)
  - Induction rule: If $x \in \Sigma^*$, and $a \in \Sigma$, then $a \cdot x \in \Sigma^*$ and $x \cdot a \in \Sigma^*$. Furthermore, $\lambda \cdot x = x \cdot \lambda = x$, and $|a \cdot x| = |x \cdot a| = 1 + |x|$.
  - *NOTE:* "a·x" denotes "a *concatenated to* x" and is formed by appending the symbol a to the left end of x. Similarly, x·a, denotes appending a to the right end of x. In either case, if x is the null string (λ), then the resultant string is "a".
  - We could have skipped saying $\forall a \in \Sigma$, $a \in \Sigma^*$, as this is covered by the induction step.

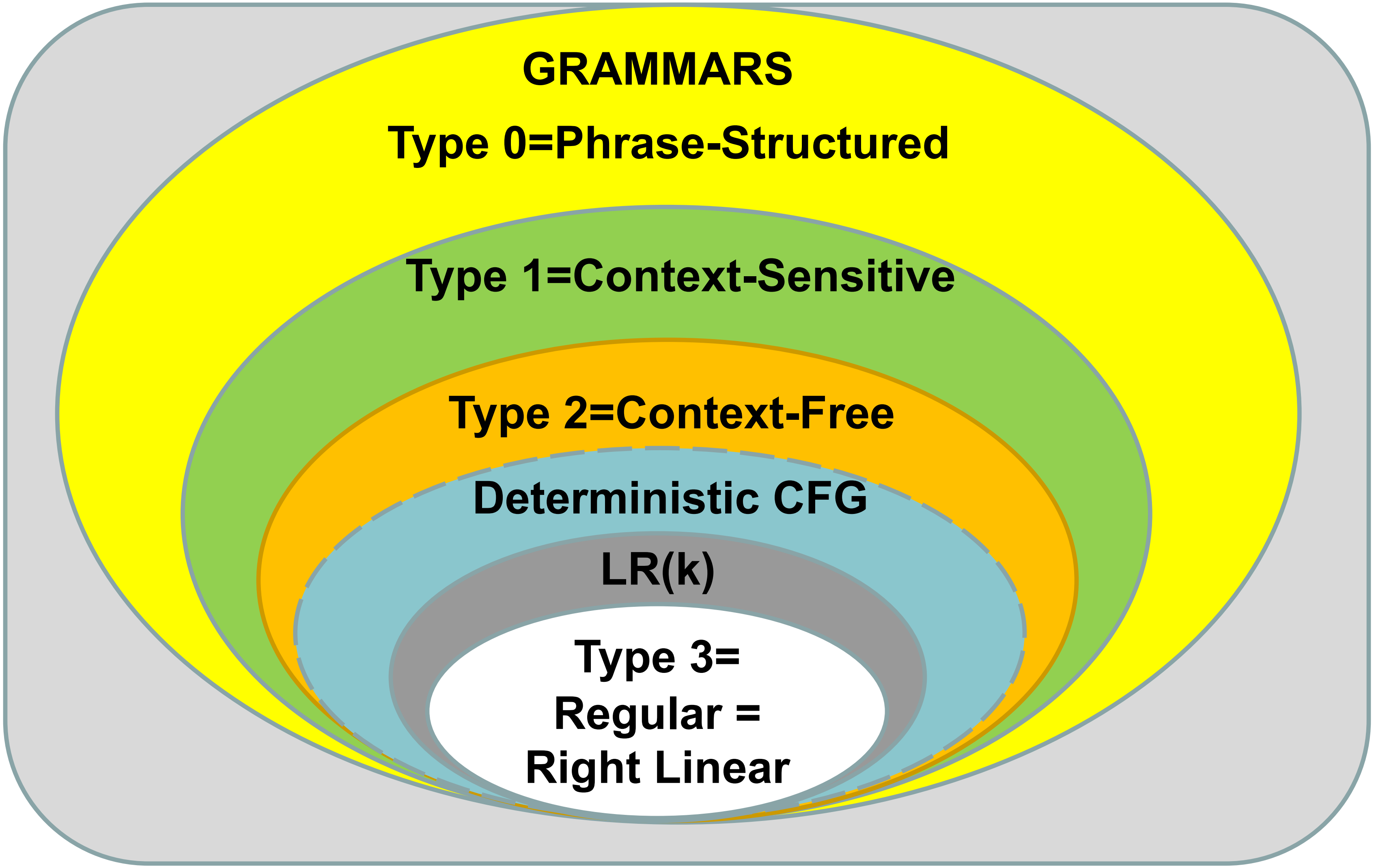
Languages

- DEFINITION 3. Let Σ be an alphabet. A *language over* Σ is a subset, L, of Σ^*.
- Example. Languages over the alphabet $\Sigma = \{a, b\}$.
  - Ø (the empty set) is a language over Σ
  - Σ^* (the universal set) is a language over Σ
  - {a, bb, aba } (a finite subset of Σ^*) is a language over Σ.
  - $\{ ab^na^m \mid n = m^2, n, m \geq 0 \}$ (infinite subset) is a language over Σ.
- DEFINITION 4. Let L and M be two languages over Σ. Then the *concatenation of L with M*, denoted L·M is the set,
  $L \cdot M = \{ x \cdot y \mid x \in L \text{ and } y \in M \}$
  The concatenation of arbitrary strings x and y is defined inductively as follows.
  Basis: When $|x| \leq 1$ or $|y| \leq 1$, then $x \cdot y$ is defined as in Definition 2.
  Inductive rule: when $|x| > 1$ and $|y| > 1$, then $x = x' \cdot a$ for some $a \in \Sigma$ and $x' \in \Sigma^*$, where $|x'| = |x|-1$. Then $x \cdot y = x' \cdot (a \cdot y)$.

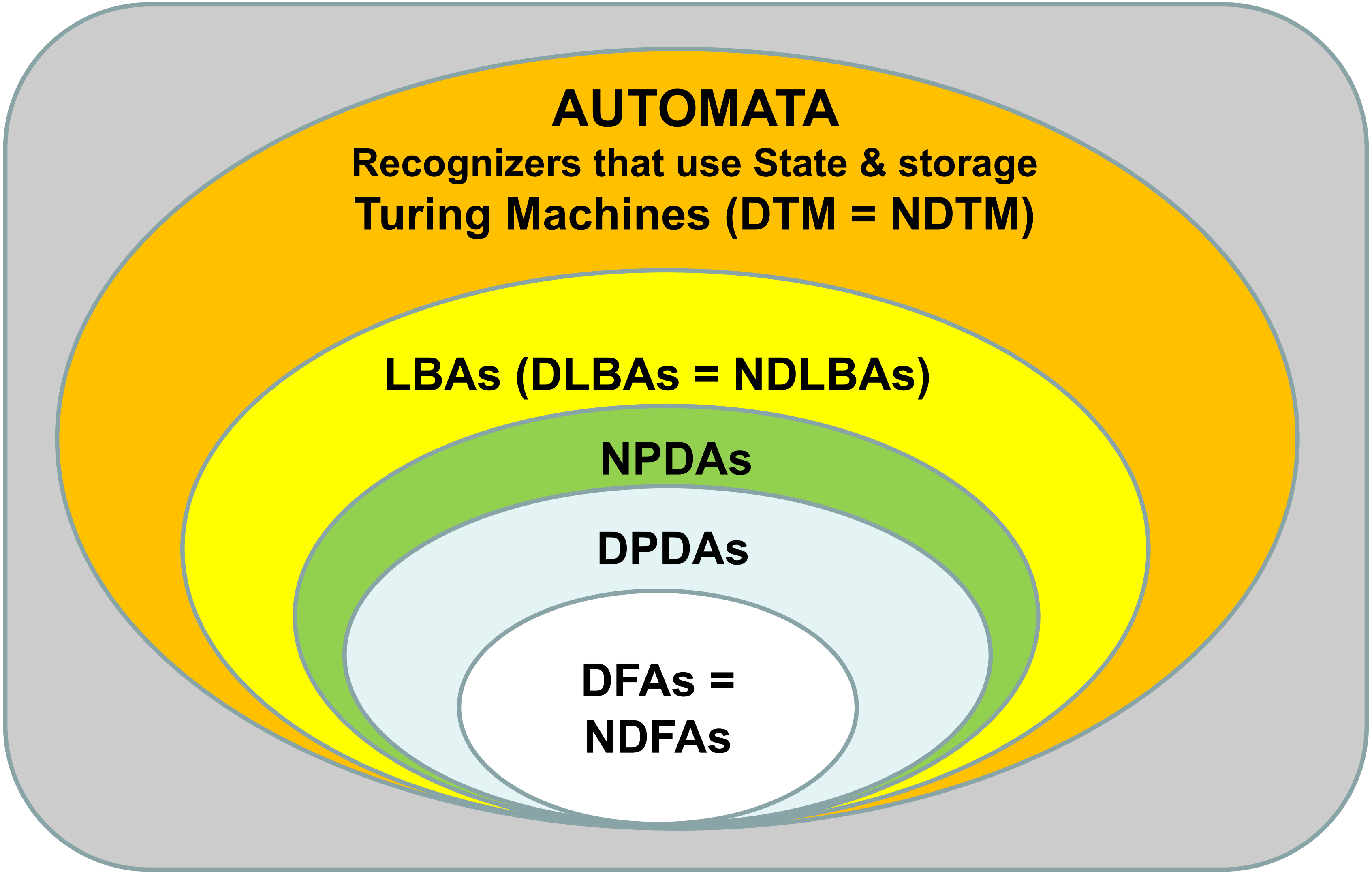
UNIVERSE OF LANGUAGES



REWRITING SYSTEMS



MODELS OF COMPUTATION



Of these models, only TMs can do general computation

What We are Studying

Computability Theory

The study of what can/cannot be done via purely computational means.

Complexity Theory

The study of what can/cannot be done <u>well</u> via purely computational means.

Graph Coloring

- Instance: A graph **G = (V, E)** and an integer **k**.
- Question: Can **G** be "properly colored" with at most **k** colors?

- Proper Coloring: a color is assigned to each vertex so that adjacent vertices have different colors.

- Suppose we have two instances of this problem (1) is True (Yes) and the other (2) is False (No).

- AND, you know (1) is Yes and (2) is No. (Maybe you have a secret program that has analyzed the two instance.)

Checking a “Yes” Answer

- Without showing how your program works (you may not even know), how can you convince someone else that instance (1) is, in fact, a Yes instance?
- We can assume the output of the program was an actual coloring of **G**. Just give that to a doubter who can easily check that no adjacent vertices are colored the same, and that no more than **k** colors were used.
- How about the No instance?
- What could the program have given that allows us to quickly "verify" (2) is a No instance?
    - No One Knows!!

Checking a “No” Answer

- The only thing anyone has thought of is to have it test all possible ways to **k**-color the graph – all of which fail, of course, if “No” is the correct answer.
- There are an exponential number of things (colorings) to check.
- For some problems, there seems to be a big difference between verifying Yes and No instances.
- To solve a problem efficiently, we must be able to solve both Yes and No instances efficiently.

Hard and Easy

- True Conjecture: If a problem is easy to solve, then it is easy to verify (just solve it and compare).
- Contrapositive: If a problem is hard to verify, then it is (probably) hard to solve.
- There is nothing magical about Yes and No instances – sometimes the Yes instances are hard to verify and No instances are easy to verify.
- And, of course, sometimes both are hard to verify.

Easy Verification

- Are there problems in which both Yes and No instances are easy to verify?

- Yes. For example: Search a list **L** of **n** values for a key **x**.
- Question: Is **x** in the list **L**?

- Yes and No instances are both easy to verify.

- In fact, the entire problem is easy to solve!!

Verify vs Solve

- Conjecture: If both Yes and No instances are easy to verify, then the problem is easy to solve.
- No one has yet proven this claim, but most researchers believe it to be true.
- Note: It is usually relatively easy to prove something is easy – just write an algorithm for it and prove it is correct and that it is fast (usually, we mean polynomial).
- But, it is usually very difficult to prove something is hard – we may not be clever enough yet. So, you will often see "appears to be hard."

Instances vs Problems

- Each instance has an *'answer*.'
  - An instance's answer is the solution of the instance - it is *not* the solution of the problem.
  - A solution of the problem is a computational procedure that finds the answer of any instance given to it – the procedure must halt on all instances – it must be an *'algorithm*.'

Three Classes of Problems

Problems can be classified to be in one of three groups (classes):

Undecidable, Exponential, and Polynomial.

Theoretically, all problems belong to exactly one of these three classes and our job is often to find which one.

Why do we Care?

When given a new problem to solve (design an algorithm for), if it's undecidable, or even exponential, you will waste a lot of time trying to write a polynomial solution for it!!

If the problem really is polynomial, it will be worthwhile spending some time and effort to find a polynomial solution and, better yet, the lowest degree polynomial solution.

You should know something about how hard a problem is before you try to solve it.

Procedure (Program)

- A finite set of operations (statements) such that
  - Each statement is finitely presented and formed from a predetermined finite set of symbols and is constrained by some set of language syntax rules.
  - The current state of the machine model is finitely presentable.
  - The semantic rules of the language specify the effects of the operations on the machine's state and the order in which these operations are executed.
  - If the procedure (eventually) halts when started on some input, it produces the correct answer to this given instance of the problem.

Algorithm

- A procedure that
  - Correctly solves any instance of a given problem.
  - Completes execution in a finite number of steps no matter what input it receives.

Sample Algorithm/Procedure

- Example algorithm:
  Linear search of a finite list for a key;
  If key is found, answer “Yes”;
  If key is not found, answer “No”; }
- Example procedure:
  Linear search of a finite list for a key;
  If key is found, answer “Yes”;
  If key is not found, try this strategy again; }

Note: Latter is not unreasonable if the list can be increased in size by some properly synchronized concurrent thread.

Procedure vs Algorithm

Looking back at our approaches to “find a key in a finite list,” we see that the algorithm always halts and always reports the correct answer. In contrast, the procedure does not halt in some cases, but never lies.

What this illustrates is the essential distinction between an algorithm and a procedure – algorithms always halt in some finite number of steps, whereas procedures may run on forever for certain inputs. A particularly silly procedure that never lies is a program that never halts for any input.

Notion of Solvable

- A problem is *solvable* if there exists an algorithm that solves it (provides the correct answer for each instance).
- The fact that a problem is solvable or, equivalently, *decidable* does not mean it is *solved*. To be solved, someone must have actually produced a correct algorithm.
- The distinction between solvable and solved is subtle. Solvable is an innate property – an unsolvable problem can never become solved, but a solvable one may or may not be solved in an individual's lifetime.

An Old Solvable Problem

Does there exist a set of positive whole numbers, a, b, c and an n>2 such that $a^n+b^n = c^n$?

In 1637, the French mathematician, Pierre de Fermat, claimed that the answer to this question is “No”. This was called Fermat’s Last Theorem, despite the fact that he never produced a proof of its correctness.

While this problem remained *unsolved* until Fermat’s claim was verified in 1995 by Andrew Wiles, the problem was always *solvable*, since it had just one question, so the solution was either “Yes” or “No”, and an algorithm *exists* for each of these candidate solutions.

Research Territory

Decidable – vs – Undecidable

(area of Computability Theory)

Exponential – vs – polynomial

(area of Computational Complexity)

For "easy" problems, we want to determine lower and upper bounds on complexity and develop best Algorithms

(area of Algorithm Design/Analysis)

A CS Grand Challenge

Does *P*=*NP*?

There are many equivalent ways to describe *P* and *NP*. For now, we will use the following.

P is the set of decision problems (those whose instances have "Yes"/ "No" answers) that can be solved in polynomial time on a deterministic computer (no concurrency or guesses allowed).

NP is the set of decision problems that can be solved in polynomial time on a non-deterministic computer (equivalently one that can spawn an unbounded number of parallel threads; equivalently one that can be verified in polynomial time on a deterministic computer).

Again, as "Does ***P*=*NP*?**" has just one question, it is solvable, we just don't yet know which solution, "Yes" or "No", is the correct one.

Computability vs Complexity

Computability focuses on the distinction between solvable and unsolvable problems, providing tools that may be used to identify unsolvable problems – ones that can never be solved by mechanical (computational) means. Surprisingly, unsolvable problems are everywhere as you will see.

In contrast, complexity theory focuses on how hard it is to solve problems that are known to be solvable. Hard solvable problems abound in the real world. We will address computability theory for the first part of this course, returning to complexity theory later in the semester.

REVIEW
REGULAR LANGUAGES

Regular Languages # 1

- Finite state automata and Regular languages
  - Definitions: Deterministic and Non-Deterministic
  - Notions of state transitions, acceptance and language accepted
  - State diagrams and state tables
  - Construction from descriptions of languages
  - Conversion of NFA to DFA
    - λ-Closure
    - Subset construction
    - Reachable states
    - Reaching states
    - Minimizing DFAs (distinguishable states)

Regular Languages # 2

- Regular expressions and Regular Sets
  - Definition of regular expressions and regular sets
  - Every regular set is a regular language
  - Every regular language is a regular set
    - Ripping states (GNFA)
    - $R_{i,j}^{k}$ expressions
      - $R_{ij}^{k+1} = (R_{ij}^{k} + R_{ik}^{k} \cdot (R_{kk}^{k})^{*} \cdot R_{kj}^{k})$
      - $L(A) = +_{f \in F} R_{1f}^{n}$
    - Regular equations
      - Uniqueness of solution to R=Q+RP
      - Solving for expressions associated with states

Regular Languages # 3

- Pumping Lemma
  - Classic non-regular languages $\{0^n 1^n \mid n >= 0\}$
  - Formal statement and proof of Pumping Lemma for Regular Languages
  - Use of Pumping Lemma (Adversarial Nature)
- Minimization (using distinguishable states)
- Myhill-Nerode
  - Right Invariant Equivalence Relations (RIER)
  - Specific RIER, $x \; R_L \; y \; \forall z \; [xz \in L \Leftrightarrow yz \in L]$ is minimal
  - Uniqueness of minimum state DFA based on R_L
  - Use to show languages are no Regular

Regular Languages # 4

- Grammars
  - Definition of grammar and notions of derivation and language
  - Restricted grammars: Regular (right and left linear)
  - Why you can’t mix right and left linear and stay in Regular domain
  - Relation of regular grammars to finite state automata

Regular Languages # 5

- Closures
  - Union, Concatenation, Keene star
  - Complement, Exclusive Union, Intersection, Set Difference, Reversal
  - Substitution, Homomorphism, Quotient, Prefix, Suffix, Substring
  - Max, Min
- Decidable Properties
  - Membership
  - $L = \emptyset$
  - $L = \Sigma^*$
  - Finiteness / Infiniteness
  - Equivalence

REVIEW
CONTEXT-FREE & CONTEXT-SENSITIVE LANGUAGES

Context-Free #1

- Context free grammars
  - Writing grammars for specific languages
  - Leftmost and rightmost derivations, Parse trees, Ambiguity
  - Closure (union, concatenation, reversal, substitution, homomorphism)
  - Pumping Lemma for CFLs

Context-Free #2

- Context free grammars
  - Chomsky Normal Form
    - Remove lambda rules
    - Remove chain rules
    - Remove non-generating (unproductive) non-terminals (and rules)
    - Remove unreachable non-terminals (and rules)
    - Make rhs match CNF constraints
  - CKY algorithm

Context-Free #3

- Push-down automata
  - Various notions of acceptance and their equivalence
  - Deterministic vs non-deterministic
  - Equivalence to CFLs
    - CFG to PDA definitely; PDA to CFG, only conceptually
  - Top-down vs bottom up parsing

Context-Free #4

- Closure
  - Union, concatenation, star
  - Substitution
  - Intersection with regular
  - Quotient with regular, Prefix, Suffix, Substring
- Non-Closure
  - Intersection, complement, min, max

Context-Sensitive

- Context sensitive grammars and LBAs
  - Rules for CSG
  - Ability to shuttle symbols to preset places
  - Just basic definition of LBA

Concrete Model of FSA

L is a finite state (regular) language over finite alphabet Σ
Each x_i is a character in Σ
$w = x_1\ x_2 \ldots x_n$ is a string to be tested for membership in L

- Arrow above represents read head that starts on left.
- $q_0 \in Q$ (finite state set) is initial state of machine.
- Only action at each step is to change state based on character being read and current state. State change is determined by a transition function δ: $Q \times \Sigma \rightarrow Q$.
- Once state is changed, read head moves right.
- Machine stops when head passes last input character.
- Machine accepts string as member of L if it ends up in a state from Final State set $F \subseteq Q$.

Finite State Automata

- A deterministic finite state automaton (DFA) A is defined by a 5-tuple
  $A = (Q,\Sigma,\delta,q_0,F)$, where
  - Q is a finite set of symbols called the states of A
  - Σ is a finite set of symbols called the alphabet of A
  - δ is a function from $Q \times \Sigma$ into Q (δ: $Q \times \Sigma \to Q$) called the transition function of A
  - $q_0 \in Q$ is a unique element of Q called the start state
  - F is a subset of Q ($F \subseteq Q$) called the final states (can be empty)

DFA Transitions

- Given a DFA, $A = (Q,\Sigma,\delta,q_0,F)$, we can definition the reflexive transitive closure of δ, $\delta^*:Q \times \Sigma^* \rightarrow Q$, by
  - $\delta^*(q,\lambda) = q$ where λ is the string of length 0
    - Note that text uses ϵ rather than λ as symbol for string of length zero
  - $\delta^*(q,ax) = \delta^*(\delta(q,a),x)$, where $a \in \Sigma$ and $x \in \Sigma^*$
  - Note that this means
    $\delta^*(q,a) = \delta(q,a)$, where $a \in \Sigma$ as $a = a\lambda$
- We also define the transitive closure of δ, δ^+, by
  - $\delta^+(q,w) = \delta^*(q,w)$ when $|w|>0$ or, equivalently, $w \in \Sigma^+$
- The function δ^* describes every step of computation by the automaton starting in some state until it runs out of characters to read

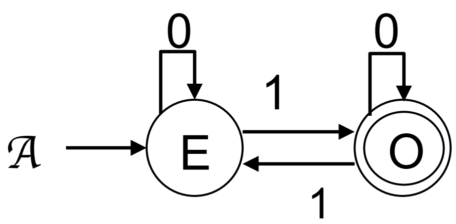
Regular Languages and DFAs

- Given a DFA, $A = (Q,\Sigma,\delta,q_0,F)$, we can define the language accepted by A as those strings that cause it to end up in a final state once it has consumed the entire string
- Formally, the language accepted by A is
  - $\{ w \mid \delta^*(q_0,w) \in F \}$
- We generally refer to this language as $L(A)$
- We define the notion of a Regular Language by saying that a language is Regular if and only if it is accepted (recognized) by some DFA

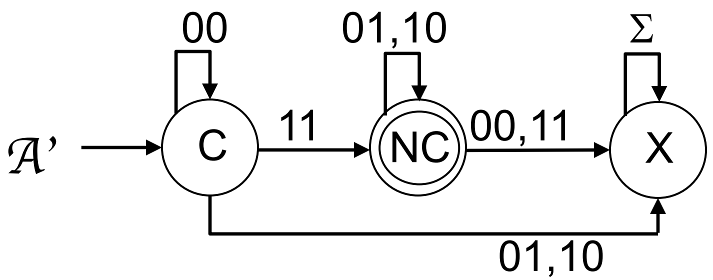
State Diagram

- A finite state automaton can be described by a state diagram, where
  - Each state is represented by a node labelled with that state, e.g., q
  - The state state has an arc entering it with no source, e.g., $\rightarrow q_0$
  - Each transition $\delta(q,a) = s$ is represented by a directed arc from node q to node s that is labelled with the letter a, e.g., $q \xrightarrow{a} s$
  - Each final state has an extra circle around its node, e.g., f

Sample DFAs # 1, 2

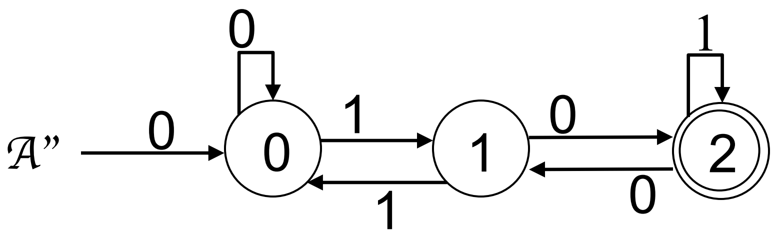


$\mathcal{A}$ = ({E,O}, {0,1}, δ, E, {O}), where δ is defined by above diagram. L($\mathcal{A}$) = { w | w is a binary string of odd parity }



$\mathcal{A}$' = ({C,NC,X}, {00,01,10,11}, δ', C, {NC}), where δ' is defined by above diagram. L($\mathcal{A}$') = { w | w is a pair of binary strings where the bottom string is the 2's complement of the top one, both read least (lsb) to most significant bit (msb) }

Sample DFA # 3



$\mathcal{A}$” = ({0,1,2}, {0,1}, δ”, 0, {2}), where δ” is defined by above diagram. L($\mathcal{A}$”) = { w | w is a binary string of length at least 1 being read left to right (msb to lsb) that, when interpreted as a decimal number divided by 3, has a remainder of 2 }

State Transition Table

- A finite state automaton can be described by a state transition table with $|Q|$ rows and $|\Sigma|$ columns
- Rows are labelled with state names and columns with input letters
- The start state has some indicator, e.g., a greater than sign ($>q$) and each final state has some indicator, e.g., an underscore ($\underline{f}$)
- The entry in row q, column a, contains $\delta(q,a)$
- In general we will use state diagrams, but transition tables are useful in some cases (state minimization)

Sample DFA # 4

	0	1
0 % 5	0 % 5	1 % 5
1 % 5	2 % 5	3 % 5
2 % 5	4 % 5	0 % 5
Accept State **3 % 5**	1 % 5	2 % 5
4 % 5	3 % 5	4 % 5

$\mathcal{A}'''$ = ({0%5,1%5,2%5,3%5,4%5}, {0,1}, δ''', 0, {3%5}), where δ''' is defined by above diagram.

L($\mathcal{A}''$) = { w | w is a binary string of length at least 1 being read left to right (msb to lsb) that, when interpreted as a decimal number divided by 5, has a remainder of 3 }

Really, this is better done as a state diagram, but have put this up so you can see the pattern.

Sample DFA # 5

	A-Z	a-z	0-9	@#$%^&
⇨ **Empty**	A	a	0	@
A	A	Aa	A0	A@
a	Aa	a	a0	a@
0	A0	a0	0	0@
@	A@	a@	0@	@
Aa	Aa	Aa	Aa0	Aa@
A0	A0	Aa0	A0	A0@
A@	A@	Aa@	A0@	A@
a0	Aa0	a0	a0	a0@
a@	Aa@	a@	a0@	a@
0@	A0@	a0@	0@	0@
Aa0	Aa0	Aa0	Aa0	Aa0@
Aa@	Aa@	Aa@	Aa0@	Aa@
A0@	A0@	Aa0@	A0@	A0@
a0@	Aa0@	a0@	a0@	a0@
Aa0@	Aa0@	Aa0@	Aa0@	Aa0@

This checks a string to see if it's a legal password. In our case, a legal password must contain at least one of each of the following: lower case letter, upper case letter, number, and special character from the following set {!@#$%^&}. No other characters are allowed

DFA Closure

- Regular languages (those recognized by DFAs) are closed under complement, union, intersection, difference and exclusive or ($\oplus$) and many other set operations
- Let $A_1 = (Q_1,\Sigma,\delta_1,q_0,F_1)$, $A_2 = (Q_2,\Sigma,\delta_2,s_0,F_2)$ be arbitrary DFAs
- $\Sigma^*\text{-}L(A_1)$ is recognized by $A_1^C = (Q_1,\Sigma,\delta_1,q_0,Q_1\text{-}F_1)$
- Define $A_3 = (Q_1 \times Q_2,\Sigma,\delta_3,<q_0,s_0>,F_3)$ where $\delta_3(<q,s>,a)= <\delta_1(q,a),\delta_2(s,a)>$, $q \in Q_1$, $s \in Q_2$, $a \in \Sigma$
  - $L(A_1) \cup L(A_2)$ is recognized when $F_3=(F_1 \times Q_2) \cup (Q_1 \times F_2)$
  - $L(A_1) \cap L(A_2)$ is recognized when $F_3=F_1 \times F_2$
  - $L(A_1)$ - $L(A_2)$ is recognized when $F_3=F_1 \times (Q_2\text{-}F_2)$
  - $L(A_1) \oplus L(A_2)$ is recognized when $F_3=F_1 \times (Q_2\text{-}F_2) \cup (Q_1\text{-}F_1) \times F_2$

Complement of Regular Sets

- Let $A = (Q,\Sigma,\delta,q_0,F)$
- Simply create new automaton
  $A^C = (Q,\Sigma,\delta,q_0,Q\text{-}F)$
- $L(A^C) = \{ w \mid \delta^*(q_0,w) \in Q\text{-}F \} =$
  $\{ w \mid \delta^*(q_0,w) \notin F \} =$
  $\{ w \mid w \notin L(A) \}$
- Again, imagine trying to do this in the context of regular expressions
- Choosing the right representation can make a very big difference in how easy or hard it is to prove some property is true

Parallelizing DFAs

- Regular sets can be shown closed under many binary operations using the notion of parallel machine simulation
- Let $A_1 = (Q_1,\Sigma,\delta_1,q_0,F_1)$ and $A_2 = (Q_2,\Sigma,\delta_2,s_0,F_2)$ where $Q_1 \cap Q_2 = \emptyset$
- $B = (Q_1 \times Q_2,\Sigma,\delta_3,<q_0,s_{0>,}F_3)$ where $\delta_3(<q,s>,a) = < \delta_1(q,a), \delta_2(s,a) >$
- Union is $F_3 = F_1 \times Q_2 \cup Q_1 \times F_2$
- Intersection is $F_3 = F_1 \times F_2$
  - Can do by combining union and complement
- Difference is $F_3 = F_1 \times (Q_2 - F_2)$
  - Can do by combining intersection and complement
- Exclusive Or is $F_3=F_1 \times (Q_2-F_2) \cup (Q_1-F_1) \times F_2$

Non-determinism NFA

- A non-deterministic finite state automaton (NFA) A is defined by a 5-tuple $A = (Q,\Sigma,\delta,q_0,F)$, where
  - Q is a finite set of symbols called the states of A
  - Σ is a finite set of symbols called the alphabet of A
  - δ is a function from $Q \times \Sigma_e$ into $P(Q) = 2^Q$; Note: $\Sigma_e = (\Sigma \cup \{\lambda\})$ ($\delta: Q \times \Sigma_e \rightarrow P(Q)$) called the transition function of A; by definition $q \in \delta(q,\lambda)$
  - $q_0 \in Q$ is a unique element of Q called the start state
  - F is a subset of Q ($F \subseteq Q$) called the final states
  - Note that a state/input (called a discriminant) can lead nowhere new, one place or many places in an NFA; moreover, an NFA can jump between states even without reading any input symbol
  - For simplicity, we often extend the definition of $\delta: Q \times \Sigma_e$ to a variant that handles sets of states, where $\delta: P(Q) \times \Sigma_e$ is defined as $\delta(S,a) = \cup_{q \in S} \delta(q,a)$, where $a \in \Sigma_e$ – if $S=\emptyset$, $\cup_{q \in S} \delta(q,a) = \emptyset$

NFA Transitions

- Given an NFA, $A = (Q,\Sigma,\delta,q_0,F)$, we can define the reflexive transitive closure of δ, $\delta^*:P(Q) \times \Sigma^* \rightarrow P(Q)$, by
  - λ-Closure(S) = $\{ t \mid t \in \delta^*(S,\lambda)\}$, $S \in P(Q)$ – extended δ
  - $\delta^*(S,\lambda) = \lambda$-Closure(S)
  - $\delta^*(S,ax) = \delta^*(\lambda\text{-Closure}(\delta(S,a),x))$, where $a \in \Sigma$ and $x \in \Sigma^*$
    - Note that $\delta^*(S,ax) = \cup_{q\in S} \cup_{p\in\lambda\text{-Closure}(\delta(q,a))} \delta^*(p,x)$, where $a \in \Sigma$ and $x \in \Sigma^*$
- We also define the transitive closure of δ, δ^+, by
  - $\delta^+(S,w) = \delta^*(S,w)$ when $|w|>0$ or, equivalently, $w \in \Sigma^+$
- The function δ^* describes every "possible" step of computation by the non-deterministic automaton starting in some state until it runs out of characters to read

NFA Languages

- Given an NFA, $A = (Q,\Sigma,\delta,q_0,F)$, we can define the language accepted by A as those strings that allow it to end up in a final state once it has consumed the entire string – here we just mean that there is some accepting path
- Formally, the language accepted by A is
  - $\{ w \mid (\delta^*(\lambda\text{-Closure}(\{q_0\}),w) \cap F) \neq \varnothing \}$
- Notice that we accept if there is any set of choices of transitions that lead to a final state

Finite State Diagram

- A non-deterministic finite state automaton can be described by a finite state diagram, except
  - We now can have transitions labelled with λ
  - The same letter can appear on multiple arcs from a state q to multiple distinct destination states

Equivalence of DFA and NFA

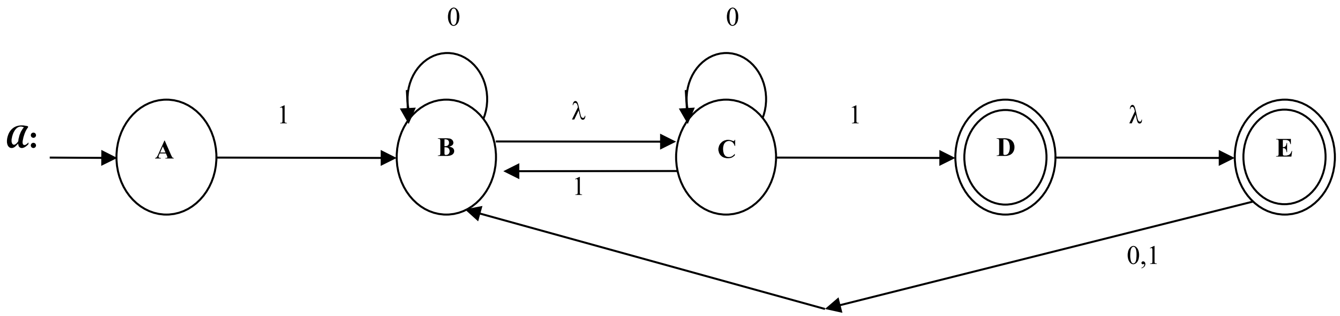
- Clearly every DFA is an NFA except that $\delta(q,a) = s$ becomes $\delta(q,a) = \{s\}$, so any language accepted by a DFA can be accepted by an NFA.
- The challenge is to show every language accepted by an NFA is accepted by an equivalent DFA. That is, if A is an NFA, then we can construct a DFA A', such that $L(A') = L(A)$.

Constructing DFA from NFA

- Let $A = (Q,\Sigma,\delta,q_0,F)$ be an arbitrary NFA
- Let S be an arbitrary subset of Q.
  - Construct the sequence seq(S) to be a sequence that contains all elements of S in lexicographical order, using angle brackets to . That is, if S={q1, q3, q2} then seq(S)=<q1,q2,q3>. If S=Ø then seq(S)=<>
- Our goal is to create a DFA, A', whose state set contains seq(S), whenever there is some w such that $S=\delta^*(q_0,w)$
- To make our life easier, we will act as if the states of A' are sets, knowing that we really are talking about corresponding sequences

λ-Closure

- Define the λ-Closure of a state q as the set of states one can arrive at from q, without reading any additional input.
- Formally $\lambda\text{-Closure}(q) = \{ t \mid t \in \delta^*(q,\lambda) \}$
- We can extend this to $S \in P(Q)$ by
  $\lambda\text{-Closure}(S) = \{ t \mid t \in \delta^*(q,\lambda), q \in S \} = \{ t \mid t \in \lambda\text{-Closure}(q), q \in S\}$



State	A	B	C	D	E
λ-closure	{ A }	{ B , C }	{ C }	{ D, E }	{ E }

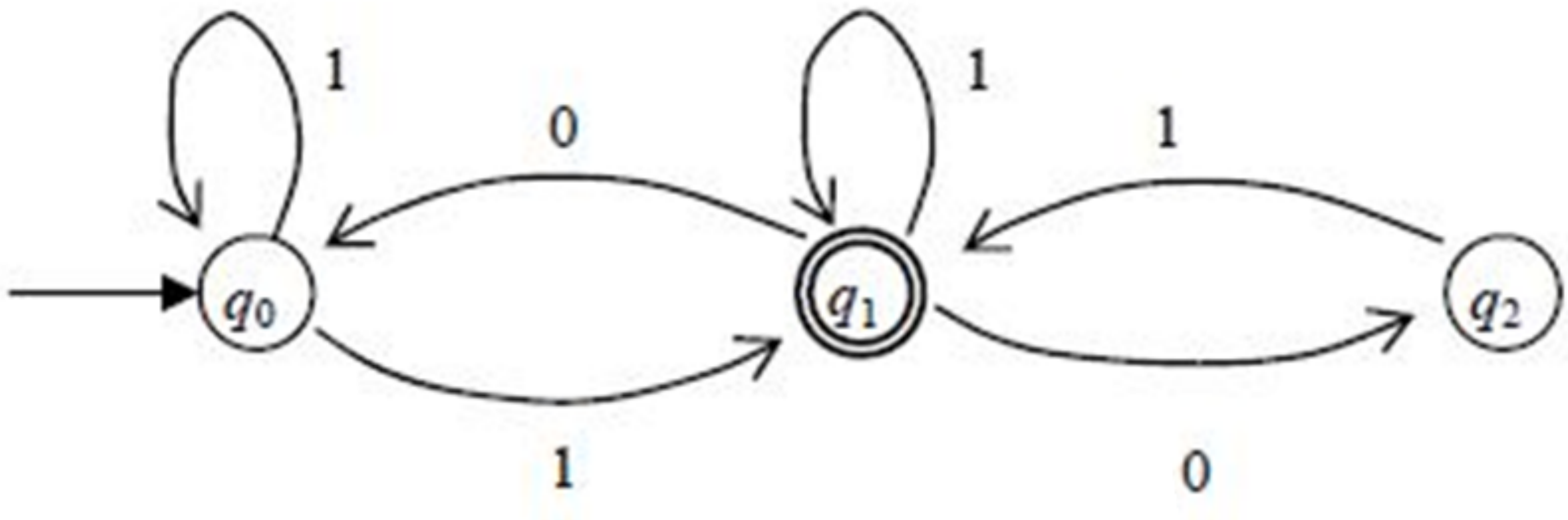
Details of DFA

- Let $A = (Q,\Sigma,\delta,q_0,F)$ be an arbitrary NFA
- In an abstract sense,
  $A' = (<P(Q)>,\Sigma,\delta', <\lambda\text{-Closure}(\{q_0\})>, F')$,
  but we really don't need so many states ($2^{|Q|}$) and we can iteratively determine those needed by starting at λ-Closure($\{q_0\}$) and keeping only states reachable from here
- Define $\delta'(<S>,a) = <\lambda\text{-Closure}(\delta(S,a))> = <\cup_{q\in S}\, \lambda\text{-Closure}(\delta(q,a))>$, where $a\in\Sigma$, $S \in P(Q)$
- $F' = \{<S> \in <P(Q)> \mid (S \cap F) \neq \emptyset \}$

Regular Languages and NFAs

- Showing that every NFA can be simulated by a DFA that accepts the same language proves the following
- A language is Regular if and only if it is accepted (recognized) by some NFA

Convert from NFA to DFA



Lexical Analysis

- Consider distinguishing variable names from keywords like IF, THEN, ELSE, etc.
- This really screams for non-determinism
- Non deterministic automata typically have fewer states
- However, non-deterministic FSA interpretation is not as fast as deterministic

Practice Problems

Practice

1. Using DFA's (not any equivalent notation) show that the Regular Languages are closed under Min, where Min(L) = { w | w $\in$ L, but no proper prefix of w is in L}.. This means that w $\in$ Min(L) iff w $\in$ L and for no y≠λ is x in L, where w=xy. Said a third way, w is not an extension of any element in L.
2. a.) Present a transition diagram for an NFA for the language associated with the regular expression (1011 + 111 + 101)*.

   b.) Use the standard conversion technique (subsets of states) to convert the NFA from (a) to an equivalent DFA. Be sure to not include unreachable states.

Practice DFA/NFA

1. Present a transition diagram for a DFA that recognizes the set of binary strings that, when interpreted as entering the DFA most to least significant digit, each represents a binary number that is divisible by either 2 or 3 or both. Thus, 100, 110, 1001 and 1100 are in the language, but 01, 101, 111 and 1011 are not.
2. a.) Present a transition diagram with no lambda transitions for an NFA associated with the regular expression (0111 + 111 + 011)*.
   Your NFA must have no more than four states.
   b.) Use the standard conversion technique (subsets of states) to convert the NFA from (a) to an equivalent DFA. Be sure to not include unreachable states.

Regular Expressions

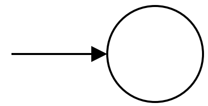
- Primitive:
  - Φ denotes {}
  - λ denotes $\{\lambda\}$
  - a where a is in Σ denotes {a}
- Closure:
  - If R and S are regular expressions then so are $R \cdot S$, $R + S$ and R^*, where
    - $R \cdot S$ denotes $RS = \{ xy \mid x \text{ is in } R \text{ and } y \text{ is in } S \}$
    - $R + S$ denotes $R \cup S = \{ x \mid x \text{ is in } R \text{ or } x \text{ is in } S \}$
    - R^* denotes R^*
- Parentheses are used as needed

Regular Sets = Regular Languages

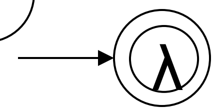
- Show every regular expression denotes a language recognized by a finite state automaton (can do deterministic or non-deterministic)
- Show every Finite State Automata recognizes a language denoted by a regular expression

Every Regular Set is a Regular Language

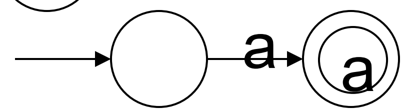
- Primitive:
  - Φ denotes {}

  - λ denotes {λ}

  - a where a is in Σ denotes {a}

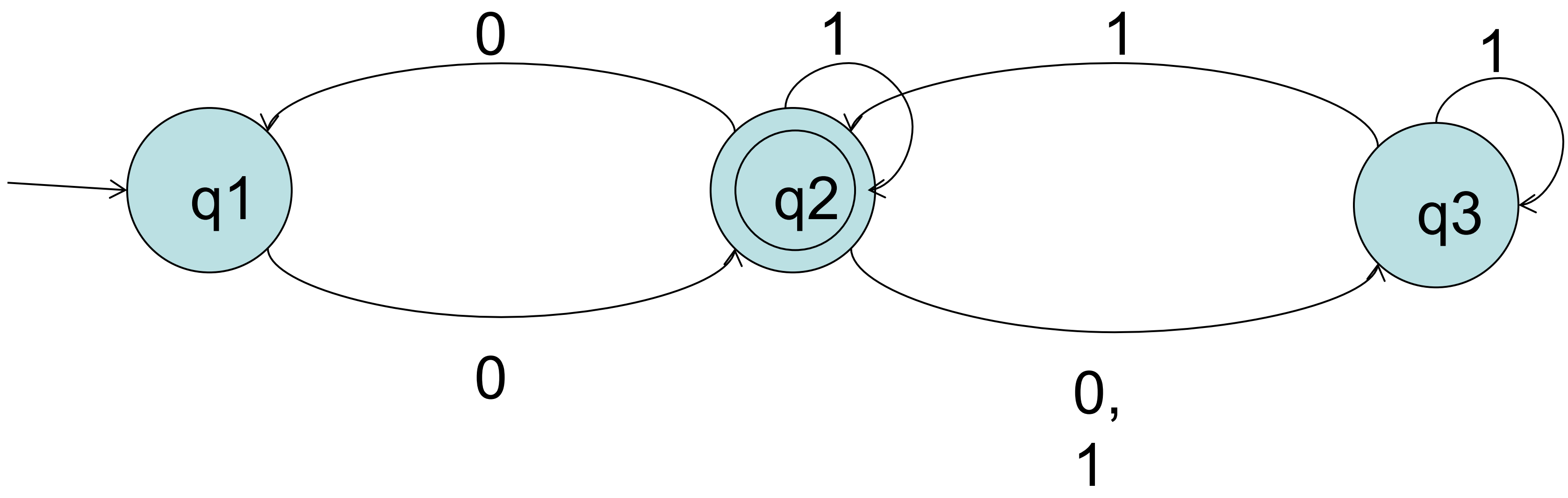
    

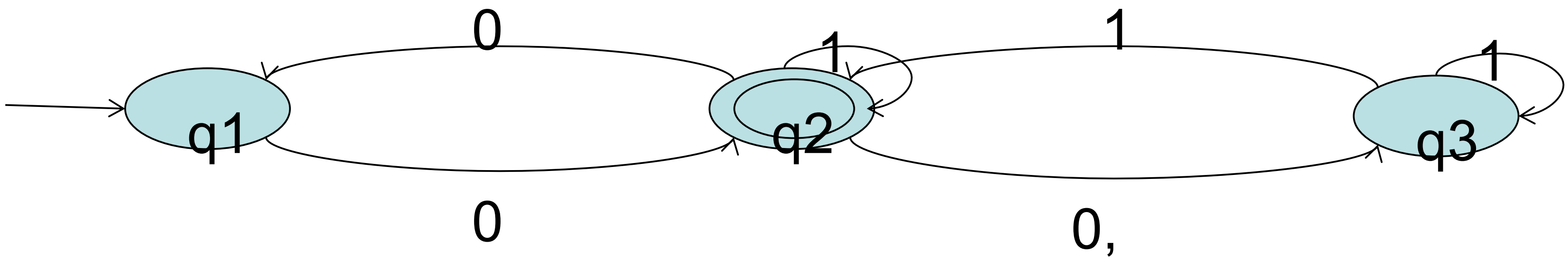
- Closure: (Assume that R's and S's states do not overlap)
  - R · S start with machine for R, add λ transitions from every final state of R's recognizer to start state of S, making final state of S final states of new machine
  - R + S create new start state and add λ transitions from new state to start states of each of R and S, making union of R's and S's final states the new final states
  - R* add λ transitions from each final state of R back to its start state, keeping original start and final states (gets R^+) – FIX?

Every Regular Language is a Regular Set Using R_{ij}^k

- This is a challenge that can be addressed in multiple ways but I like to start with the R_{ij}^k approach. Here's how it works.
- Let $A = (Q,\Sigma,\delta,q_1,F)$ be a DFA, where $Q = \{q_1,q_2, \ldots , q_n\}$
- $R_{ij}^k = \{w \mid \delta^*(q_i,w) = q_j$, and no intermediate state visited between q_i and q_j, while reading w, has index $> k$
- Basis: $k=0$, $R_{ij}^0 = \{ a \mid \delta(q_i,a) = q_j \}$ sets are either Φ, λ, or an element of Σ or λ + element of Σ, and so are regular sets
- Inductive hypothesis: Assume R_{ij}^m are regular sets for $0 \leq m \leq k$
- Inductive step: $k+1$, $R_{ij}^{k+1} = (R_{ij}^k + R_{ik+1}^k \cdot (R_{k+1k+1}^k)^* \cdot R_{k+1j}^k)$
- $L(A) = +_{f\in F}\, R_{1f}^n$

Convert to RE





$R_{11}^0 = \lambda$	$R_{12}^0 = 0$	$R_{13}^0 = \phi$
$R_{21}^0 = 0$	$R_{22}^0 = \lambda + 1$	$R_{23}^0 = 0 + 1$
$R_{31}^0 = \phi$	$R_{32}^0 = 1$	$R_{33}^0 = \lambda + 1$
$R_{11}^1 = \lambda$	$R_{12}^1 = 0$	$R_{13}^1 = \phi$
$R_{21}^1 = 0$	$R_{22}^1 = \lambda + 1 + 00$	$R_{23}^1 = 0 + 1$
$R_{31}^1 = \phi$	$R_{32}^1 = 1$	$R_{33}^1 = \lambda + 1$
$R_{11}^2 = \lambda + 0(1+00)^*0$	$R_{12}^2 = 0(1+00)^*$	$R_{13}^2 = 0(1+00)^*(0+1)$
$R_{21}^2 = (1+00)^*0$	$R_{22}^2 = (1+00)^*$	$R_{23}^2 = (1+00)^*(0+1)$
$R_{31}^2 = 1(1+00)^*0$	$R_{32}^2 = 1(1+00)^*$	$R_{33}^2 = \lambda+1+1(1+00)^*(0+1)$

- $L = R_{12}^3 =$
  $0(1+00)^* + 0(1+00)^*(0+1)\ (1+1(1+00)^*(0+1))^*\ 1(1+00)^*$

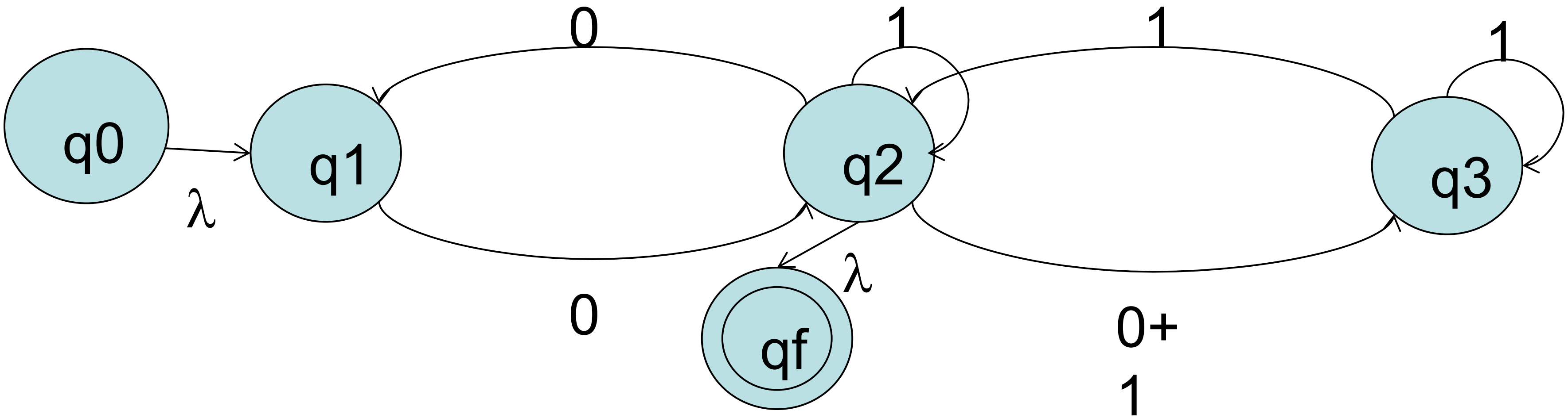
State Ripping Concept

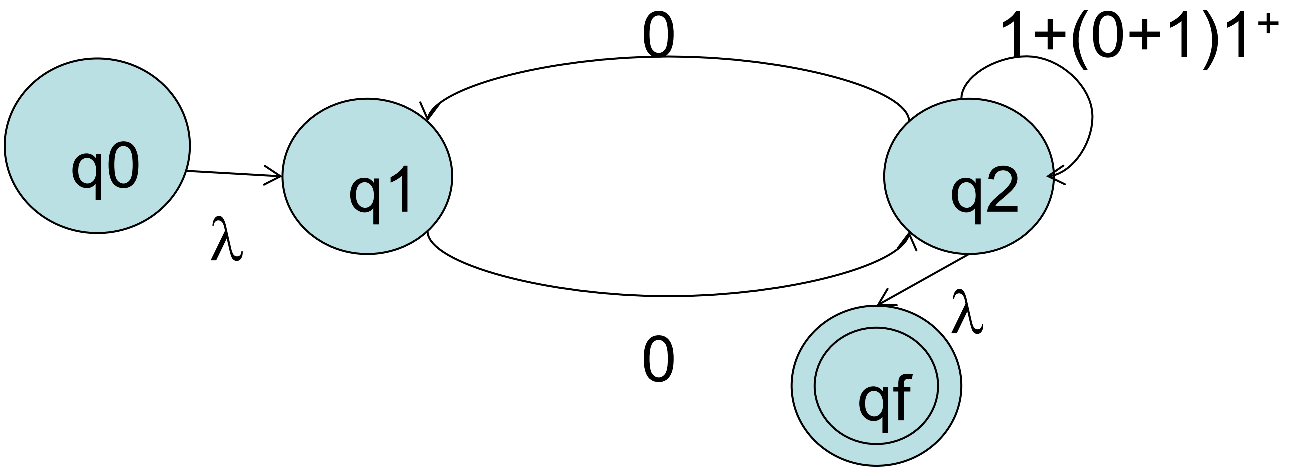
- This is similar to generalized automata approach but with fewer arcs than text. It actually gets some of its motivation from R_{ij}^k approach as well
- Add a new start state and add a λ–transition to existing start state
- Add a new final state q_f and insert λ–transitions from all existing final states to the new one; make the old final states non-final
- Leaving the start and final states, successively pick states to remove
- For each state to be removed, change the arcs of every pair of externally entering and exiting arcs to reflect the regular expression that describes all strings that could result is such a double transition; be sure to account for loops in the state being removed. Also, or (+) together expressions that have the same start and end nodes
- When have just start and final, the regular expression that leads from start to final describes the associated regular set

State Ripping Details

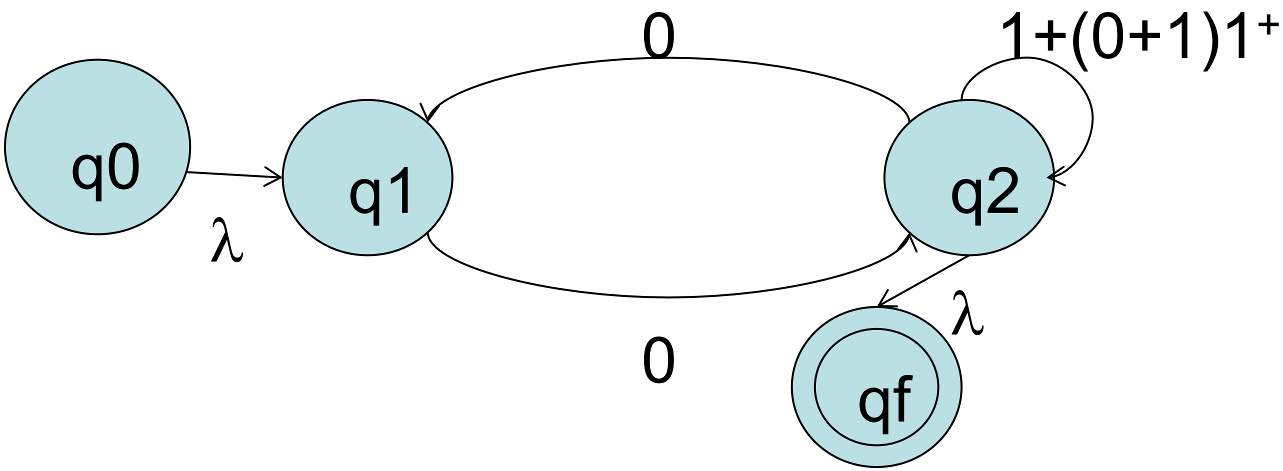
- Let B be the node to be removed
- Let e1 be the regular expression on the arc from some node A to some node B (A≠B); e2 be the expression from B back to B (or λ if there is no recursive arc); e3 be the expression on the arc from B to some other node C (C ≠B but C could be A); e4 be the expression from A to C
- Erase the existing arcs from A to B and A to C, adding a new arc from A to C labelled with the expression
  e4 + e1 e2* e3
- Do this for all nodes that have edges to B until B has no more entering edges; at this point remove B and any edges it has to other nodes and itself
- Iterate until all but the start and final nodes remain
- The expression from start to final describes regular set that is equivalent to regular language accepted by original automaton
- Note: Your choices of the order of removal make a big difference in how hard or easy this is

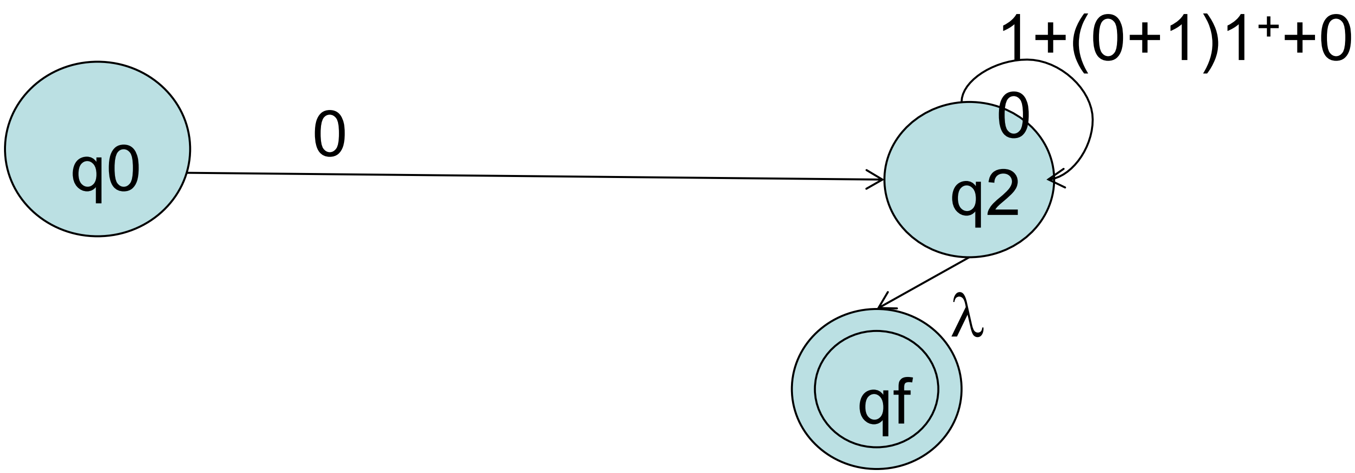
Use Ripping; Rip q3



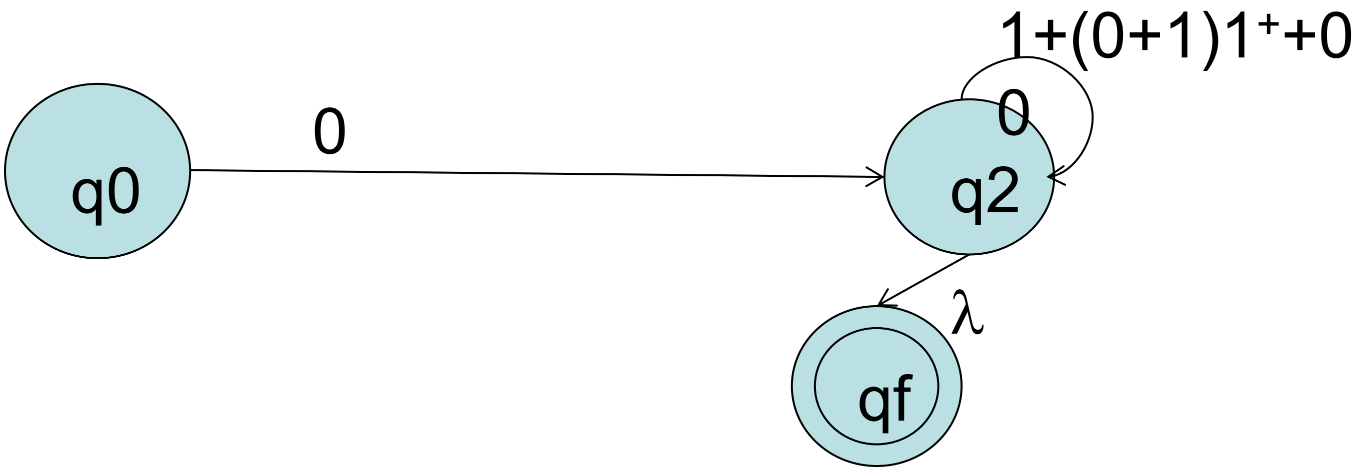


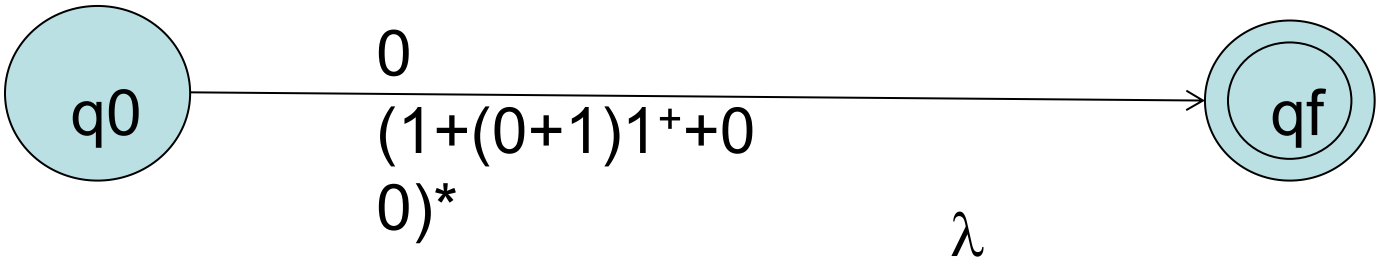
Use Ripping; Rip q1





Use Ripping; Rip q2





$L = 0\ (1+(0+1)1^{+}+00)^{*} = 0\ (1+(0+1)1^{+}+00)^{*}$

Regular Equations

- Assume that R, Q and P are sets such that P does not contain the string of length zero, and R is defined by
- $R = Q + RP$
- We wish to show that
- $R = QP^*$

Show QP* is a Solution

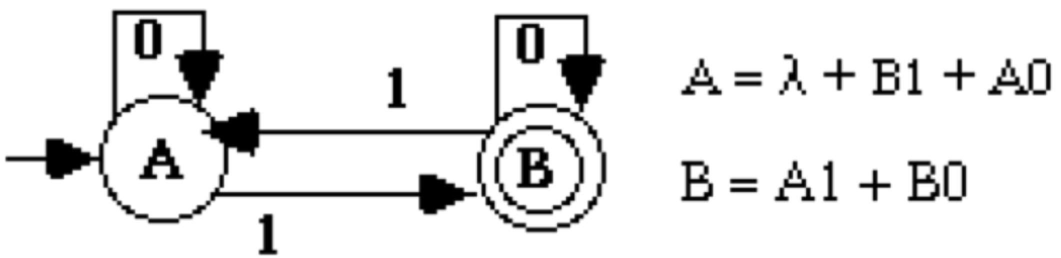
- We first show that QP* is contained in R. By definition, R = Q + RP.
- To see if QP* is a solution, we insert it as the value of R in Q + RP and see if the equation balances
- R = Q + QP*P = Q(λ+P*P) = QP*
- Hence QP* is a solution, but not necessarily the only solution.

Uniqueness of Solution

- To prove uniqueness, we show that R is contained in QP*.
- By definition, R = Q+RP = Q+(Q+RP)P
- = $Q+QP+RP^2$ = $Q+QP+(Q+RP)P^2$
- = $Q+QP+QP^2+RP^3$
- ...
- = $Q(\lambda+P+P^2+ \ldots +P^i)+RP^{i+1}$, for all i>=0
- Choose any w in R, where |W| = k. Then, from above,
- R = $Q(\lambda+P+P^2+ \ldots +P^k)+RP^{k+1}$
- but, since P does not contain the string of length zero, w is not in RP^{k+1}. But then w is in
- $Q(\lambda+P+P^2+ \ldots +P^k)$ and hence w is in QP*.

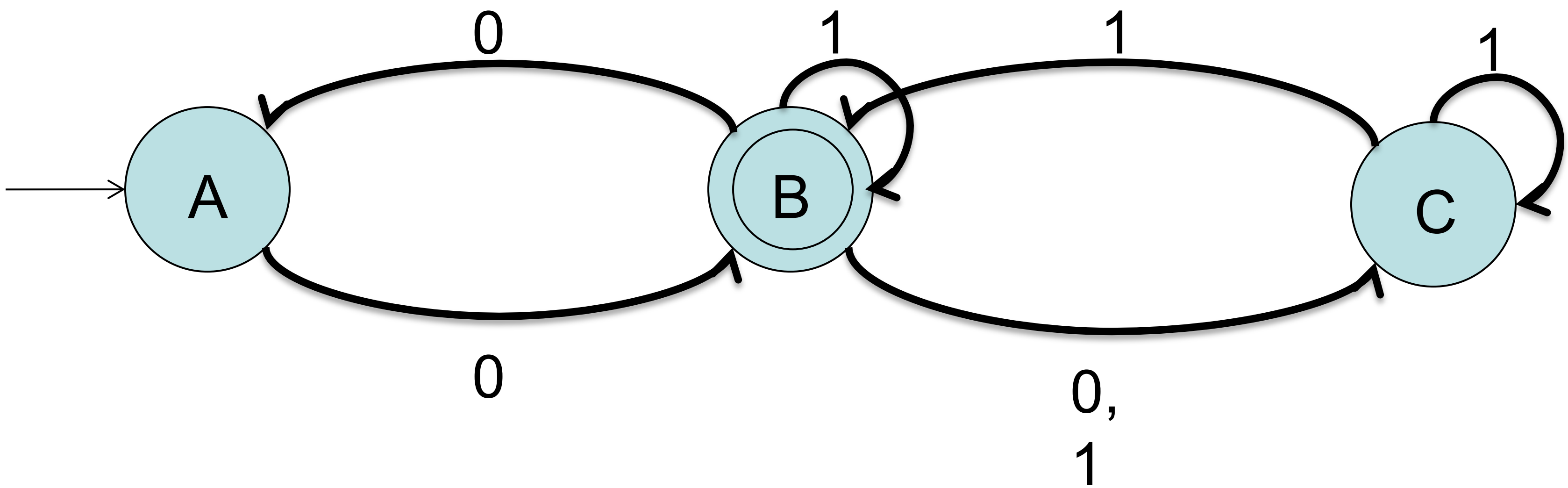
Example

- We use the above to solve simultaneous regular equations. For example, we can associate regular expressions with finite state automata as follows
- Hence,
- For A, $Q=\lambda+B1$; $P=0$
  $A = QP^* = (\lambda+B1)0^*$
  $= B10^* + 0^*$



- $B = B10^*1 + B0 + 0^*1$
  For B, $Q=0^*1$; $P= B10^*1 + B0 = B(10^*1 + 0)$
- and therefore
- $B = 0^*1(10^*1 + 0)^*$
- Note: This technique fails if there are lambda transitions.

Using Regular Equations



$A = \lambda + B0$

$B = A0 + C1 + B1$

$C = B(0+1) + C1; \; C = B(0+1)1^*$

$B = 0 + B00 + B(0+1)1^+ + B1$

$B = 0 + B\,(00+(0+1)\,1^+ + 1); \; B = 0(00 +(0+1)1^+ + 1)^*$

This is same form as with state ripping. It won't always be so.

Practice NFAs

- Write NFAs for each of the following
  - $(111 + 000)^+$
  - $(0+1)^* 101 (0+1)^+$
  - $(1 (0+1)^* 0) + (0 (0+1)^* 1)$
- Convert each NFA you just created to an equivalent DFA.

DFAs to REs

- For each of the DFAs you created for the previous page, use ripping of states and then regular equations to compute the associated regular expression. Note: You obviously ought to get expressions that are equivalent to the initial expressions.

State Minimization

- Text makes it an assignment on Page 299 in Sipser Edition 2.
- This is too important to defer, IMHO.
- First step is to remove any state that is unreachable from the start state; a depth first search rooted at start state will identify all reachable states
- One seeks to merge compatible states – states q and s are compatible if, for all strings x, $\delta^*(q,x)$ and $\delta^*(s,x)$ are either both an accepting or both rejecting states
- One approach is to discover incompatible states – states q and s are incompatible if there exists a string x such that one of $\delta^*(q,x)$ and $\delta^*(s,x)$ is an accepting state and the other is not
- There are many ways to approach this but my favorite is to do incompatible states via an n by n lower triangular matrix

Sample Minimization

- This uses a transition table
- Just an X denotes Immediately incompatible
- Pairs are dependencies for compatibility
- If a dependent is incompatible, so are pairs that depend on it
- When done, any not x--ed out are compatible
- Here, new states are <1,3>, <2,4,5>, <6>; <1,3> is start and not accept; others are accept
- Write new diagram

	a	b	c
>1	5	2	2
2	1	6	2
3	2	4	5
4	3	6	2
5	3	6	5
6	1	3	4

2	X				
3	2,5 2,4	X			
4	X	1.3	X		
5	X	1.3	X	2,5	
6	X	3,6 X 2,4	X	1,3 3,6 X 2,4	1,3 3,6 X 4,5
	1	2	3	4	5

Reversal of Regular Sets

- It is easier to do this with regular sets than with DFAs
- Let E be some arbitrary expression; E^R is formed by
  - Primitives: $\varnothing^R=\varnothing$ $\lambda^R=\lambda$ $a^R=a$
  - Closure:
    - $(A \cdot B)^R = (B^R \cdot A^R)$
    - $(A + B)^R = (A^R + B^R)$
    - $(A^*)^R = (A^{R*})$
- Challenge: How would you do this with FSA models?
  - Start with DFA; change all final to start states; change start to a final state; and reverse edges
  - Note that this creates multiple start states; can create a new start state with λ-transitions to multiple starts

Substitution

- A substitution is a function, f, from each member, a, of an alphabet, Σ, to a language L_a
- Regular languages are closed under substitution of regular languages (i.e., each L_a is regular)
- Easy to prove by replacing each member of Σ in a regular expression for a language L with regular expression for L_a
- A homomorphism is a substitution where each L_a is a single string

Quotient with Regular Sets

- Quotient of two languages B and C, denoted B/C, is defined as $B/C = \{x \mid \exists y \in C \text{ where } xy \in B\}$
- Let B be recognized by DFA
  $A_B = (Q_B,\Sigma,\delta_B,q_{1B},F_B)$ and C by
  $A_C = (Q_C,\Sigma,\delta_C,q_{1C},F_C)$
- Define the recognizer for B/C by
  $A_{B/C} = (Q_B \cup Q_B \times Q_C,\Sigma,\delta_{B/C},q_{1B}, F_B \times F_C)$
  $\delta_{B/C}(q,a) = \{\delta_B(q,a)\}$ $\quad a\in\Sigma, q\in Q_B$
  $\delta_{B/C}(q,\lambda) = \{<q,q_{1C}>\}$ $\quad q\in Q_B$
  $\delta_{B/C}(<q,p>,\lambda) = \{\delta_B(q,a),\delta_C(p,a)\}$ $\quad a\in\Sigma, q\in Q_B, p\in Q_C$
- The basic idea is that we simulate B and then randomly decide it has seen x and continue by looking for y, simulating B continuing after x but with C starting from scratch

Quotient Again

- Assume some class of languages, $\mathfrak{C}$, is closed under concatenation, intersection with regular and substitution of members of $\mathfrak{C}$, show $\mathfrak{C}$ is closed under Quotient with Regular
- $L/R = \{ x \mid \exists y \in R \text{ where } xy \in L \}$
  - Define $\Sigma' = \{ a' \mid a \in \Sigma \}$
  - Let $h(a) = a$; $h(a') = \lambda$ where $a \in \Sigma$
  - Let $g(a) = a'$ where $a \in \Sigma$
  - Let $f(a) = \{a, a'\}$ where $a \in \Sigma$
  - $L/R = h(f(L) \cap (\Sigma^* \cdot g(R)))$

Applying Meta Approach

- $INIT(L) = \{ x \mid \exists y \in \Sigma^* \text{ where } xy \in L \}$
  - $INIT(L) = h(f(L) \cap (\Sigma^* \cdot g(\Sigma^*)))$
  - Also $INIT(L) = L / \Sigma^*$
- $LAST(L) = \{ y \mid \exists x \in \Sigma^* \text{ where } xy \in L \}$
  - $LAST(L) = h(f(L) \cap (g(\Sigma^*) \cdot \Sigma^*))$
- $MID(L) = \{ y \mid \exists x,z \in \Sigma^* \text{ where } xyz \in L \}$
  - $MID(L) = h(f(L) \cap (g(\Sigma^*) \cdot \Sigma^* \cdot g(\Sigma^*)))$
- $EXTERIOR(L) = \{ xz \mid \exists y \in \Sigma^* \text{ where } xyz \in L \}$
  - $EXTERIOR(L) = h(f(L) \cap (\Sigma^* \cdot g(\Sigma^*) \cdot \Sigma^*))$

Making Life Easy

- The key in proving closure is to always try to identify the "best" equivalent formal model for regular sets when trying to prove a particular property
- For example, how could you even conceive of proving closure under intersection and complement in regular expression notations?
- Note how much easier quotient is when have closure under concatenation, and substitution and intersection with regular languages than showing in FSA notation

Reachable and Reaching

- Reachable*from*(q) = { p | $\exists$w $\ni$ δ(q,w)=p }
  – Just do depth first search from q, marking all reachable states. Works for NFA as well.
- Reaching*to*(q) = { p | $\exists$w $\ni$ δ(p,w)=q }
  – Do depth first from q, going backwards on transitions, marking all reaching states. Works for NFA as well.

Min and Max

- $\text{Min}(L) = \{ w \mid w \in L$ and no proper prefix of w is in $L \} =$
  $\{ w \mid w \in L$ and if $w=xy$, $x \in \Sigma^*$, $y \in \Sigma^+$ then $x \notin L\}$
- $\text{Max}(L) = \{ w \mid w \in L$ and w is not the proper prefix of any word in $L \} =$
  $\{ w \mid w \in L$ and if $y \in \Sigma^+$ then $wy \notin L \}$
- Examples:
  - $\text{Min}(0(0+1)^*) = \{0\}$
  - $\text{Max}(0(0+1)^*) = \{\}$
  - $\text{Min}(01 + 0 + 10) = \{0,10\}$
  - $\text{Max}(01 + 0 + 10) = \{01,10\}$
  - $\text{Min}(\{a^i b^j c^k \mid i \le k \text{ or } j \le k\}) = \{a^i b^j c^k \mid \mid i,j \ge 0, k = \min(i, j)\}$
  - $\text{Max}(\{a^i b^j c^k \mid i \le k \text{ or } j \le k\}) = \{\}$ because k has no bound
  - $\text{Min}(\{a^i b^j c^k \mid i \ge k \text{ or } j \ge k\}) = \{\lambda\}$
  - $\text{Max}(\{a^i b^j c^k \mid i \ge k \text{ or } j \ge k\}) = \{a^i b^j c^k \mid \mid i,j \ge 0, k = \max(i, j)\}$

Regular Closed under Min

- Assume L is regular then Min(L) is regular
- Let L= *L*(A), where A = (Q,Σ,δ,q_0,F) is a DFA with no state unreachable from q_0
- Define $A_{min} = (Q\cup\{dead\},\Sigma,\delta_{min},q_0,F)$, where for $a\in\Sigma$ $\delta_{min}(q,a) = \delta(q,a)$, if $q\in Q-F$; $\delta_{min}(q,a) = dead$, if $q\in F$; $\delta_{min}(dead,a) = dead$

The reasoning is that the machine A_{min} accepts only elements in L that are not extensions of shorter strings in L. By making it so transitions from all final states in A_{min} go to the new “dead” state, we guarantee that extensions of accepted strings will not be accepted by this new automaton.

Therefore, Regular Languages are closed under Min.

Regular Closed under Max

- Assume L is regular then Max(L) is regular
- Let L= *L*(A), where A = (Q,Σ,δ,q_0,F) is a DFA with no state unreachable from q_0
- Define $A_{max} = (Q,\Sigma,\delta,q_0,F_{max})$, where
  F_{max}= { f | f∈F and Reachable*from*⁺(f)∩F=Φ }
  where Reachable*from*⁺(q) = { p | ∃w ∍ |w|>0 and δ(q,w) = p }

The reasoning is that the machine A_{max} accepts only elements in L that cannot be extended. If there is a non-empty string that leads from some final state f to any final state, including f, then f cannot be final in A_{max}. All other final states can be retained.

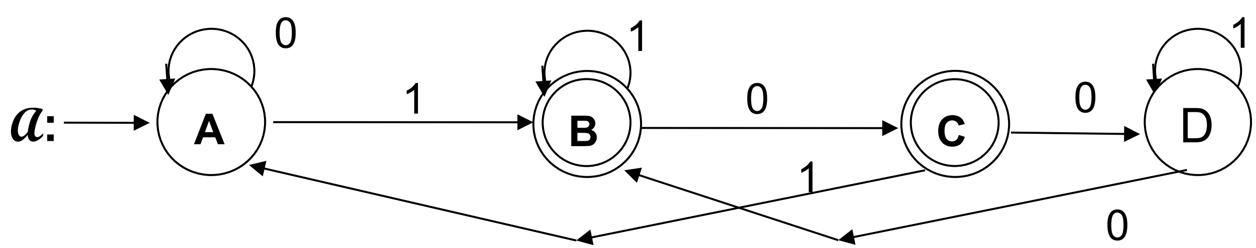
The inductive definition of Reachable*from*⁺ is:

1. Reachable*from*⁺(q) contains { s | there exists an element of Σ, a, such that δ(q,a) = s }
2. If s is in Reachable*from*⁺ (q) then Reachable*from*⁺ (q) contains
   { t | there exists an element of Σ, a, such that δ(s,a) = t }
3. No other states are in Reachable*from*⁺(q)

Therefore, Regular Languages are closed under Max.

Practice R_{ij}^k

Convert the DFA below to a regular expression, first by using either the GNFA (or state ripping) or the R_{ij}^k approach, and then by using regular equations. You must show all steps in each part of this solution.



Practice Minimization

Minimize the number of states in the following DFA, showing the determination of incompatible states (table on right).

	a	b	c
>1	2	3	5
2	5	4	4
3	2	4	5
4	6	4	2
5	5	2	4
6	5	4	2

2					
3					
4					
5					
6					
	>1	2	3	4	5

Construct and write down your new, equivalent automaton!!

Pumping Lemma Concept

- Let $A = (Q,\Sigma,\delta,q_1,F)$ be a DFA, where $Q = \{q_1,q_2, \ldots , q_N\}$
- The “pigeon hole principle” tells us that whenever we visit N+1 or more states, we must visit at least one state more than once (loop)
- Any string, w, of length N or greater leads to us making N transitions after visiting the start state, and so we visit at least one state more than once when reading w

Pumping Lemma For Regular

- Theorem: Let L be regular then there exists an $N>0$ such that, if $w \in L$ and $|w| \geq N$, then w can be written in the form xyz, where $|xy| \leq N$, $|y|>0$, and for all $i \geq 0$, $xy^iz \in L$
- This means that interesting regular languages (infinite ones) have a very simple self-embedding property that occurs early in long strings

Pumping Lemma Proof

- If L is regular then it is recognized by some DFA, $A=(Q,\Sigma,\delta,q_0,F)$. Let $|Q| = N$ states. For any string w, such that $|w| \geq N$, A must make N+1 state visits to consume its first N characters, followed by $|w|-N$ more state visits.
- In its first N+1 state visits, A must enter at least one state two or more times.
- Let $w = v_1 \ldots v_j \ldots v_k \ldots v_m$, where $m = |w|$, and $\delta(q_0,v_1 \ldots v_j)=\delta(q_0,v_1 \ldots v_k)$, $k > j$, and let this state represent the first one repeated while A consumes w.
- Define $x = v_1 \ldots v_j$, $y = v_{i+1} \ldots v_k$, and $z = v_{k+1} \ldots v_m$. Clearly w=xyz. Moreover, since $k > j$, $|y| > 0$, and since $k \leq N$, $|xy| \leq N$.
- Since A is deterministic, $\delta(q_0,xy)=\delta(q_0,xy^i)$, for all $i \geq 0$.
- Thus, if $w \in L$, $\delta(q_0,xyz) \in F$, and so $\delta(q_0,xy^iz) \in F$, for all $i \geq 0$.
- Consequently, if $w \in L$, $|w| \geq N$, then w can be written in the form xyz, where $|xy| \leq N$, $|y| > 0$, and for all $i \geq 0$, $xy^iz \in L$.

Lemma's Adversarial Process

- Assume $L = \{a^n b^n \mid n>0\}$ is regular
- P.L.: Provides $N > 0$
  - We CANNOT choose N; that's the P.L.'s job
- Our turn: Choose $a^N b^N \in L$
  - We get to select a string in L
- P.L.: $a^N b^N = xyz$, where $|xy| \leq N$, $|y| > 0$, and for all $i \geq 0$, $xy^iz \in L$
  - We CANNOT choose split, but P.L. is constrained by N
- Our turn: Choose $i = 0$.
  - We have the power here
- P.L: $a^{N-|y|}b^N \in L$; just a consequence of P.L.
- Our turn: $a^{N-|y|}b^N \notin L$; just a consequence of L's structure
- CONTRADICTION, so L is NOT regular

xwx is not Regular (PL)

- **L = { x w x | x,w∈{a,b}+} :**
- Assume that L is Regular.
- PL: Let N > 0 be given by the Pumping Lemma.
- YOU: Let s be a string, s ∈ L, such that s = $a^N baa^N b$
- PL: Since s ∈ L and |s| ≥ N, s can be split into 3 pieces, s = xyz, such that |xy| ≤ N and |y| > 0 and ∀ i ≥ 0 xy^iz ∈ L
- YOU: Choose i = 2
- PL: xy^2z = xyyz ∈ L
- Thus, $a^{N + |y|}baa^Nb$ would be in L, but this is not so since N+|y| ≠ N
- We have arrived at a contradiction.
- Therefore L is not Regular.

$a^{Fib(k)}$ is not Regular (PL)

- **L = {$a^{Fib(k)}$ | k>0} :**
- Assume that L is regular
- Let N be the positive integer given by the Pumping Lemma
- Let *s* be a string **s = $a^{Fib(N+3)}$**$\in$ **L**
- Since *s* $\in$ L and |s| ≥ N (Fib(N+3)>N in all cases; actually Fib(N+2)>N as well), s is split by PL into xyz, where |xy| ≤ N and |y| > 0 and for all i ≥ 0, $xy^iz \in$ L
- We choose i = 2; by PL: xy^2z = xyyz$\in$ L
- Thus, $a^{Fib(N+3)+|y|}$ would be $\in$ L. This means that there is a Fibonacci number between Fib(N+3) and Fib(N+3)+N, but the smallest Fibonacci greater than Fib(N+3) is Fib(N+3)+Fib(N+2) and Fib(N+2)>N
  This is a contradiction, therefore L is not regular ■
- Note: Using values less than N+3 could be dangerous because N could be 1 and both Fib(2) and Fib(3) are within N (1) of Fib(1).

Pumping Lemma Problems

- Use the Pumping Lemma to show each of the following is not regular
  - $\{ 0^m 1^{2n} \mid m \leq n \}$
  - $\{ ww^R \mid w \in \{a,b\}^+ \}$
  - $\{ 1^{n^2} \mid n > 0 \}$
  - $\{ ww \mid w \in \{a,b\}^+ \}$

Myhill-Nerode Theorem

The following are equivalent:

1. L is accepted by some DFA
2. L is the union of some of the classes of a right invariant equivalence relation, R, of finite index.
3. The specific right invariance equivalence relation R_L where $x \ R_L \ y$ iff $\forall z$ [$xz \in L$ iff $yz \in L$] has finite index

Definition. R is a right invariant equivalence relation iff R is an equivalence relation and $\forall z$ [$x \ R \ y$ implies $xz \ R \ yz$].

Note: This is only meaningful for relations over strings.

Myhill-Nerode 1 $\Rightarrow$ 2

1. Assume L is accepted by some DFA, $A = (Q,\Sigma,\delta,q_1,F)$
2. Define R_A by $x\ R_A\ y$ iff $\delta^*(q_1,x) = \delta^*(q_1,y)$. First, R_A is defined by equality and so is obviously an equivalence relation (Clearly if $\delta^*(q_1,x) = \delta^*(q_1,y)$ then $\forall z\ \delta^*(q_1,xz) = \delta^*(q_1,yz)$ because A is deterministic. Moreover if $\forall z\ \delta^*(q_1,xz) = \delta^*(q_1,yz)$ then $\delta^*(q_1,x) = \delta^*(q_1,y)$, just by letting $z = \lambda$. Putting it together $x\ R_A\ y$ L iff $\forall z\ xz\ R_A\ yz$. Thus, R_A is right invariant; its index is $|Q|$ which is finite; and $L(A) = \cup_{\delta^*(x)\in F}[x]_{RA}$, where $[x]_{RA}$ refers to the equivalence class containing the string x.

Myhill-Nerode 2 $\Rightarrow$ 3

2. Assume L is the union of some of the classes of a right invariant equivalence relation, R, of finite index.
3. Since x R y iff $\forall z$ [xz R yz], R is right invariant and L is the union of some of the equivalence classes, then
   $x \; R \; y \Rightarrow \forall z \, [\, xz \in L \text{ iff } yz \in L \,] \Rightarrow x \; R_L \; y$.
   This means that the index of R_L is less than or equal to that of R and so is finite. Note than the index of R_L is then less than or equal to that of any other right invariant equivalence relation, R, of finite index that defines L.

Myhill-Nerode 3 $\Rightarrow$ 1

3. Assume the specific right invariance equivalence relation R_L where $x\ R_L\ y$ iff $\forall z\ [\ xz \in L$ iff $yz \in L\ ]$ has finite index
1. Define the automaton $A = (Q,\Sigma,\delta,q_1,F)$ by
   $Q = \{\ [x]_{R_L} \mid x \in \Sigma^*\ \}$
   $\delta([x]_{R_L},a) = [xa]_{R_L}$
   $q1 = [\lambda]$
   $F = \{\ [x]_{R_L} \mid x \in L\ \}$

   Note: This is the minimum state automaton and all others are either equivalent or have redundant indistinguishable states

Use of Myhill-Nerode

- $L = \{a^n b^n \mid n>0\}$ is NOT regular.
- Assume otherwise.
- M-N says that the specific r.i. equiv. relation R_L has finite index, where $x\ R_L\ y$ iff $\forall z\ [\ xz \in L$ iff $yz \in L\ ]$.
- Consider the equivalence classes $[a^i b]$ and $[a^j b]$, where $i,j>0$ and $i \neq j$.
- $a^i b b^{i-1} \in L$ but $a^j b b^{i-1} \notin L$ and so $[a^i b]$ is not related to $[a^j b]$ under R_L and thus $[a^i b] \neq [a^j b]$.
- This means that R_L has infinite index.
- Therefore L is not regular.

xwx is not Regular (MN)

- **L = { x a x | x∈{a,b}+} :**
- We consider the right invariant equivalence class $[a^ib]$, $i>0$.
- It's clear that a^ibaa^ib is in the language, but a^kbaa^ib is not when $k < i$.
- This shows that there is a separate equivalence class, $[a^ib]$, induced by R_L, for each $i>0$. Thus, the index of R_L is infinite and Myhill-Nerode states that L cannot be Regular.

$a^{Fib(k)}$ is not Regular (MN)

- **L = $\{a^{Fib(k)} \mid k>0\}$:**
- We consider the collection of right invariant equivalence classes $[a^{Fib(j)}]$, $j > 2$.
- It's clear that $a^{Fib(j)}a^{Fib(j+1)}$ is in the language, but $a^{Fib(k)}a^{Fib(j+1)}$ is not when $k>2$ and $k \neq j$ and $k \neq j+2$
- This shows that there is a separate equivalence class $[a^{Fib(j)}]$ induced by R_L, for each $j > 2$.
- Thus, the index of R_L is infinite and Myhill-Nerode states that L cannot be Regular.

Myhill-Nerode and Minimization

- Corollary: The minimum state DFA for a regular language, L, is formed from the specific right invariance equivalence relation R_L where
$x \, R_L \, y$ iff $\forall z \, [\, xz \in L$ iff $yz \in L \,]$
- Moreover, all minimum state machines have the same structure as the above, except perhaps for the names of states

What is Regular So Far?

- Any language accepted by a DFA
- Any language accepted by an NFA
- Any language specified by a Regular Expression
- Any language representing the unique solution to a set of properly constrained regular equations

What is <u>NOT</u> Regular?

- Well, anything for which you cannot write an accepting DFA or NFA, or a defining regular expression, or a right/left linear grammar, or a set of regular equations, but that's not a very useful statement
- There are two tools we have:
  - Pumping Lemma for Regular Lnaguges
  - Myhill-Nerode Theorem

Finite State Transducers

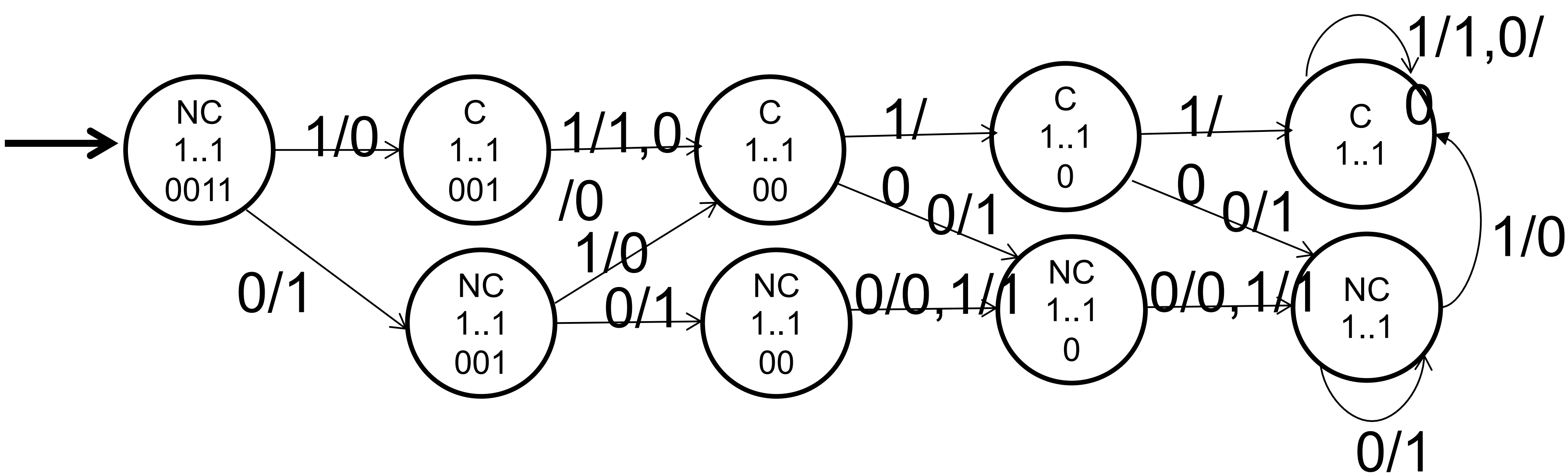
- A transducer is a machine with output
- Mealy Model
  - $M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0)$

    Γ is the finite output alphabet

    $\gamma: Q \times \Sigma \rightarrow \Gamma$ is the output function
  - Essentially a Mealy Model machine produced a character of output for each character of input it consumes, and it does so on the transitions from one state to the next.
  - A Mealy Model represents a synchronous circuit whose output is triggered each time a new input arrives.

Sample Mealy Model

- Write a Mealy finite state machine that produces the 2's complement result of subtracting 1101 from a binary input stream (assuming at least 4 bits of input)



Finite State Transducers

- Moore Model
  - $M = (Q, \Sigma, \Gamma, \delta, \gamma, q_0)$

    Γ is the finite output alphabet

    $\gamma: Q \rightarrow \Gamma$ is the output function
  - Essentially a Moore Model machine produced a character of output whenever it enters a state, independent of how it arrived at that state.
  - A Moore Model represents an asynchronous circuit whose output is a steady state until new input arrives.

Practice MNT, Grammar, Mealy

1. For each of the following, prove it is not regular by using the Pumping Lemma or Myhill-Nerode. You must do at least one of these using the Pumping Lemma and at least one using Myhill-Nerode.

a. **$L = \{ x\#y \mid x, y \in \{0,1\}^+$ and y is the ones complement of x $\}$**

b. **$L = \{ a^i b^j c^k \mid i > j * k \}$**

c. **$L = \{ x\ w\ x \mid x, w \in \{a,b\}^+$ Here $|x|>0$ and $|w|>0$ $\}$**

2. Write a regular (right linear) grammar that generates **$L = \{ w \mid w \in \{0,1\}^+$** and **w** interpreted as a binary number is divisible by either 2 or 3 or both. .

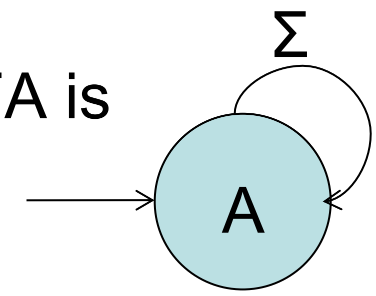
3. Present a Mealy Model finite state machine that reads an input **$x \in \{0, 1\}^+$** and produces the binary number that represents the result of adding binary **1001** to **x** (assumes all numbers are positive, including results). Assume that **x** is read starting with its least significant digit.
Examples: **00010 → 01011; 00101 → 01110;**
00111 → 10000; 00110 → 01111

Decision and Closure Properties

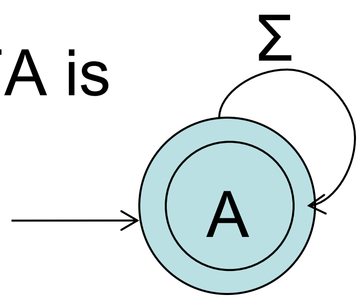
Regular Languages

Decidable Properties

- Membership (just run DFA over string)
- L = Ø: Minimize and see if minimum state DFA is



- L = Σ*: Minimize and see if minimum state DFA is



- Finiteness: Minimize and see if there are no loops emanating from a final state
- Equivalence: Minimize both and see if isomorphic

Closure Properties

- Virtually everything with members of its own class as we have already shown

- Union, concatenation, Kleene *, complement, intersection, set difference, reversal, substitution, homomorphism, quotient with regular sets, Prefix, Suffix, Substring, Exterior, Min, Max and so much more

Formal Languages

Includes and Expands on Chapter 2 of Sipser

History of Formal Language

- In 1940s, Emil Post (mathematician) devised rewriting systems as a way to describe how mathematicians do proofs. Purpose was to mechanize them.
- Early 1950s, Noam Chomsky (linguist) developed a hierarchy of rewriting systems (grammars) to describe natural languages.
- Late 1950s, Backus-Naur (computer scientists) devised BNF (a variant of Chomsky's context-free grammars) to describe the programming language Algol.
- 1960s was the time of many advances in parsing. In particular, parsing of context free was shown to be no worse than $O(n^3)$. More importantly, useful subsets were found that could be parsed in $O(n)$.

Formalism for Grammars

Definition : A language is a set of strings of characters from some alphabet.

The strings of the language are called sentences or statements.

A string over some alphabet is a finite sequence of symbols drawn from that alphabet.

A meta-language is a language that is used to describe another language.

A very well known meta-language is BNF (Backus Naur Form)

It was developed by John Backus and Peter Naur, in the late 50s, to describe programming languages.

Noam Chomsky in the early 50s developed context free grammars that can be expressed using BNF.

Grammars

- $G = (V, \Sigma, R, S)$ is a Phrase Structured Grammar (PSG) where
  - V: Finite set of non-terminal symbols
  - Σ: Finite set of terminal symbols
  - R: finite set of rules of form $\alpha \to \beta$,
    - α in $(V \cup \Sigma)^* \, V \, (V \cup \Sigma)^*$
    - β in $(V \cup \Sigma)^*$
  - S: a member of V called the start symbol
- Right linear restricts all rules to be of forms
  - α in V
  - β of form ΣV, Σ or λ

Derivations

- $x \Rightarrow y$ reads as x derives y iff
  - $x = \gamma\alpha\delta$, $y = \gamma\beta\delta$ and $\alpha \rightarrow \beta$
- $\Rightarrow^*$ is the reflexive, transitive closure of $\Rightarrow$
- $\Rightarrow^+$ is the transitive closure of $\Rightarrow$
- $x \Rightarrow^* y$ iff $x = y$ or $x \Rightarrow^* z$ and $z \Rightarrow y$
- Or, $x \Rightarrow^* y$ iff $x = y$ or $x \Rightarrow z$ and $z \Rightarrow^* y$
- $L(G) = \{ w \mid S \Rightarrow^* w \}$ is the language generated by G.

Regular Grammars

- Regular grammars are also called right linear grammars
- Each rule of a regular grammar is constrained to be of one of the three forms:

$A \rightarrow a, \qquad A \in V,\ a \in \Sigma^*$

$A \rightarrow \lambda, \qquad A \in V,\ a \in \Sigma^*$

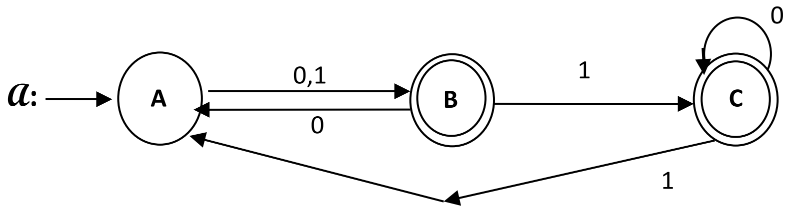
$A \rightarrow aB, \qquad A, B \in V,\ a \in \Sigma^*$

DFA to Regular Grammar

- Every language recognized by a DFA is generated by an equivalent regular grammar
- Given $A = (Q,\Sigma,\delta,q_0,F)$, $L(A)$ is generated by $G_A = (Q,\Sigma,R,q_0)$ where R contains
  $q \to as$ iff $\delta(q,a) = s$
  $q \to \lambda$ iff $q \in F$

Example of DFA to Grammar

- **DFA**



- **Grammar**

A	$\rightarrow$	**0 B**	|	**1 B**		
B	$\rightarrow$	**0 A**	|	**1 C**	|	λ
C	$\rightarrow$	**0 C**	|	**1 A**	|	λ

Regular Grammar to NFA

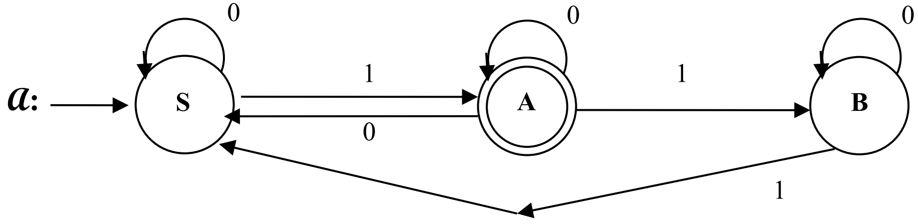
- Every language generated by a regular grammar is recognized by an equivalent NFA
- Given G = (V, Σ, R, S), *L*(G) is recognized by $A_G = (V \cup \{f\}, \Sigma, \delta, S, \{f\})$ where δ is defined by
  $\delta(A,a) \subseteq \{B\}$ iff $A \rightarrow aB$
  $\delta(A,a) \subseteq \{f\}$ iff $A \rightarrow a$
  $\delta(A,\lambda) \subseteq \{f\}$ iff $A \rightarrow \lambda$

Example of Grammar to NFA

- **Grammar**

S	$\rightarrow$	**0 S**	**|**	**1 A**				
A	$\rightarrow$	**0 S**	**|**	**0 A**	**|**	**1 B**	**|**	λ
B	$\rightarrow$	**1 S**	**|**	**0 B**				

- **DFA**



What More is Regular?

- Any language, L, generated by a right linear grammar
- Any language, L, generated by a left linear grammar ($A \rightarrow a$, $A \rightarrow \lambda$, $A \rightarrow Ba$)
  - Easy to see L is regular as we can reverse these rules and get a right linear grammar that generates L^R, but then L is the reverse of a regular language which is regular
  - Similarly, the reverse L^R of any regular language L is right linear and hence the language itself is left linear
- Any language, L, that is the union of some of the classes of a right invariant equivalence relation of finite index

Mixing Right and Left Linear

- We can get non-Regular languages if we present grammars that have both right and left linear rules
- To see this, consider G = ({S,T}, Σ, R, S), where R is:
  - $S \rightarrow aT$
  - $T \rightarrow Sb \mid b$
- *L*(G) = $\{ a^n b^n \mid n > 0 \}$ which is a classic non-regular, context-free language

Context Free Languages

Context Free Grammar

$G = (V, \Sigma, R, S)$ is a PSG where

Each member of R is of the form

$A \rightarrow \alpha$ where α is a strings $(V \cup \Sigma)^*$

Note that the left hand side of a rule is a letter in V;

The right hand side is a string from the combined alphabets

The right hand side can even be empty (ε or λ)

A context free grammar is denoted as a CFG and the language generated is a Context Free Language (CFL).

A CFL is recognized by a Push Down Automaton (PDA) to be discussed a bit later.

Sample CFG

Example of a grammar for a small language:

G = ({<program>, <stmt-list>, <stmt>, <expression>},
{begin, end, ident, ;, =, +, -}, R, <program>) where R is

<program> → begin <stmt-list> end

<stmt-list> → <stmt> | <stmt> ; <stmt-list>

<stmt> → ident = <expression>

<expression> → ident + ident | ident - ident | ident

Here “ident” is a token return from a scanner, as are “begin”, “end”, “;”, “=”, “+”, “-”

Note that “;” is a separator (Pascal style) not a terminator (C style).

Derivation

A sentence generation is called a derivation.

Grammar for a simple assignment statement:

R1	**<assgn>**	**→**	**<id> := <expr>**
R2	**<id>**	**→**	**a | b | c**
R3	**<expr>**	**→**	**<id> + <expr>**
R4		**|**	**<id> * <expr>**
R5		**|**	**(<expr>)**
R6		**|**	**<id>**

In a **leftmost derivation** only the leftmost non-terminal is replaced

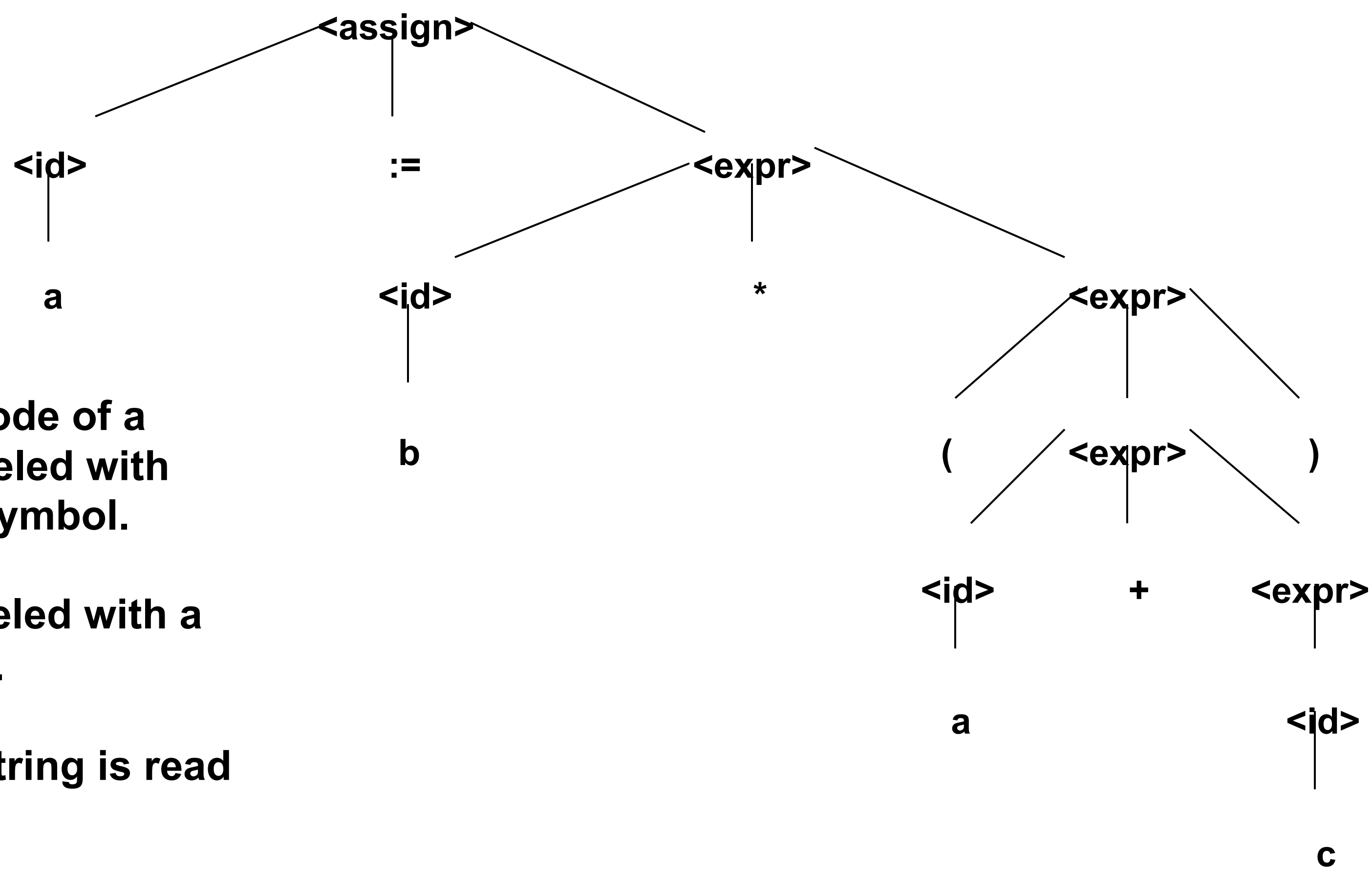
The statement a := b * (a + c)
Is generated by the leftmost derivation:

<assgn>	**⇒ <id> := <expr>**	**R1**
	⇒ a := <expr>	**R2**
	⇒ a := <id> * <expr>	**R4**
	⇒ a := b * <expr>	**R2**
	⇒ a := b * (<expr>)	**R5**
	⇒ a := b * (<id> + <expr>)	**R3**
	⇒ a := b * (a + <expr>)	**R2**
	⇒ a := b * (a + <id>)	**R6**
	⇒ a := b * (a + c)	**R2**

Parse Trees

<u>**A parse tree is a graphical representation of a derivation**</u>

For instance the parse tree for the statement a := b * (a + c) is:



Every internal node of a parse tree is labeled with a non-terminal symbol.

Every leaf is labeled with a terminal symbol.

The generated string is read left to right

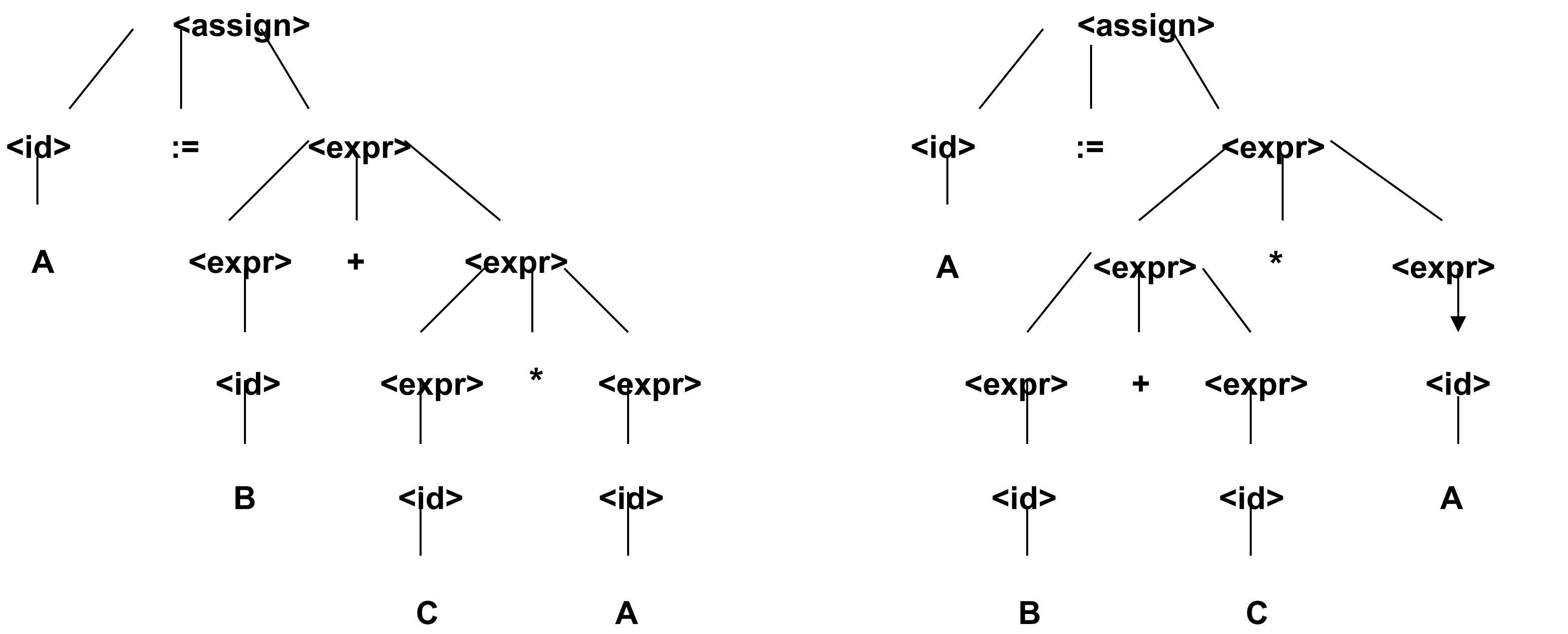
Ambiguity

A grammar that generates a sentence for which there are two or more distinct parse trees is said to be "<u>ambiguous</u>"

For instance, the following grammar is ambiguous because it generates distinct parse trees for the expression a := b + c * a

<assgn> → <id> := <expr>
<id> → a | b | c
<expr> → <expr> + <expr>
| <expr> * <expr>
| (<expr>)
| <id>

Ambiguous Parse



This grammar generates two parse trees for the same expression.

If a language structure has more than one parse tree, the meaning of the structure cannot be determined uniquely.

Precedence

Operator precedence:

If an operator is generated lower in the parse tree, it indicates that the operator has precedence over the operator generated higher up in the tree.

An unambiguous grammar for expressions:

<assign> → <id> := <expr>
<id> → a | b | c
<expr> → <expr> + <term>
| <term>
<term> → <term> * <factor>
| <factor>
<factor> → (<expr>)
| <id>

This grammar indicates the usual precedence order of multiplication and addition operators.

This grammar generates unique parse trees independently of doing a rightmost or leftmost derivation

Left (right)most Derivations

Leftmost derivation:

<assgn> → <id> := <expr>
→ a := <expr>
→ a := <expr> + <term>
→ a := <term> + <term>
→ a := <factor> + <term>
→ a := <id> + <term>
→ a := b + <term>
→ a := b + <term> *<factor>
→ a := b + <factor> * <factor>
→ a := b + <id> * <factor>
→ a := b + c * <factor>
→ a := b + c * <id>
→ a := b + c * a

Rightmost derivation:

<assgn> ⇒ <id> := <expr>
⇒ <id> := <expr> + <term>
⇒ <id> := <expr> + <term> *<factor>
⇒ <id> := <expr> + <term> *<id>
⇒ <id> := <expr> + <term> * a
⇒ <id> := <expr> + <factor> * a
⇒ <id> := <expr> + <id> * a
⇒ <id> := <expr> + c * a
⇒ <id> := <term> + c * a
⇒ <id> := <factor> + c * a
⇒ <id> := <id> + c * a
⇒ <id> := b + c * a
⇒ a := b + c * a

Ambiguity Test

- A Grammar is Ambiguous if there are two distinct parse trees for some string
- Or, two distinct leftmost derivations
- Or, two distinct rightmost derivations
- Some languages are inherently ambiguous but many are not
- Unfortunately (to be shown later) there is no systematic test for ambiguity of context free grammars

Unambiguous Grammar

When we encounter ambiguity, we try to rewrite the grammar to avoid ambiguity.

The ambiguous expression grammar:

<expr> → <expr> <op> <expr> | id | int | (<expr>)
<op> → + | - | * | /

Can be rewritten as:

<expr> → <term> | <expr> + <term> | <expr> - <term>
<term> → <factor> | <term> * <factor> | <term> / <factor>.
<factor> → id | int | (<expr>)

Parsing Problem

The parsing Problem: Take a string of symbols in a language (tokens) and a grammar for that language to construct the parse tree or report that the sentence is syntactically incorrect.

For correct strings:

Sentence + grammar → parse tree

For a compiler, a sentence is a program:

Program + grammar → parse tree

Types of parsers:

Top-down aka predictive (recursive descent parsing)

Bottom-up aka shift-reduce

Removing Left Recursion if doing Top Down

Given left recursive and non left recursive rules

$A \to A\alpha_1 \mid \ldots \mid A\alpha_n \mid \beta_1 \mid \ldots \mid \beta_m$

Can view as

$A \to (\beta_1 \mid \ldots \mid \beta_m) (\alpha_1 \mid \ldots \mid \alpha_n)^*$

Star notation is an extension to normal notation with obvious meaning

Now, it should be clear this can be done right recursive as

$A \to \beta_1 B \mid \ldots \mid \beta_m B$

$B \to \alpha_1 B \mid \ldots \mid \alpha_n B \mid \lambda$

Right Recursive Expressions

Grammar: Expr → Expr + Term | Term
Term → Term * Factor | Factor
Factor → (Expr) | Int

Fix: Expr → Term ExprRest
ExprRest → + Term ExprRest | λ
Term → Factor TermRest
TermRest → * Factor TermRest | λ
Factor → (Expr) | Int

Bottom Up vs Top Down

- Bottom-Up: Two stack operations
  - Shift (move input symbol to stack)
  - Reduce (replace top of stack α with A, when $A \rightarrow \alpha$)
  - Challenge is when to do shift or reduce and what reduce to do.
    - Can have both kinds of conflict
- Top-Down:
  - If top of stack is terminal
    - If same as input, read and pop
    - If not, we have an error
  - If top of stack is a non-terminal A
    - Replace A with some α, when $A \rightarrow \alpha$
    - Challenge is what A-rule to use

Chomsky Normal Form

- Each rule of a CFG is constrained to be of one of the three forms:
  $A \rightarrow a, \qquad A \in V,\ a \in \Sigma$
  $A \rightarrow BC, \qquad A,B,C \in V$
- If the language contains λ then we allow
  $S \rightarrow \lambda$
  and constrain all non-terminating rules of form to be
  $A \rightarrow BC, \qquad A \in V,\ B,C \in V\text{-}\{S\}$

Nullable Symbols

- Let $G = (V, \Sigma, R, S)$ be an arbitrary CFG
- Compute the set $\text{Nullable}(G) = \{A \mid A \Rightarrow^* \lambda\}$
- Nullable(G) is computed as follows

  $\text{Nullable}(G) \supseteq \{ A \mid A \rightarrow \lambda \}$

  Repeat

  $\quad \text{Nullable}(G) \supseteq \{ B \mid B \rightarrow \alpha \text{ and } \alpha \in \text{Nullable}^* \}$

  until no new symbols are added

Removal of λ-Rules

- Let $G = (V, \Sigma, R, S)$ be an arbitrary CFG
- Compute the set Nullable(G)
- Remove all λ-rules
- For each rule of form $B \rightarrow \alpha A \beta$ where A is nullable, add in the rule $B \rightarrow \alpha\beta$
- The above has the potential to greatly increase the number of rules and add unit rules (those of form $B \rightarrow C$, where $B,C \in V$)
- If S is nullable, add new start symbol S_0, as new start state, plus rules $S_0, \rightarrow \lambda$ and $S_0 \rightarrow \alpha$, where $S \rightarrow \alpha$

Chains (Unit Rules)

- Let G = (V, Σ, R, S) be an arbitrary CFG that has had its λ-rules removed
- For A$\in$V, Chain(A) = { B | A $\Rightarrow^*$ B, B$\in$V }
- Chain(A) is computed as follows

  Chain(A) $\supseteq$ { A }

  Repeat

      Chain(A) $\supseteq$ { C | B $\rightarrow$ C and B $\in$ Chain(A) }

  until no new symbols are added

Removal of Unit-Rules

- Let G = (V, Σ, R, S) be an arbitrary CFG that has had its λ-rules removed, except perhaps from start symbol
- Compute Chain(A) for all A∈V
- Create the new grammar G = (V, Σ, R, S) where R is defined by including for each A∈V, all rules of the form $A \rightarrow \alpha$, where $B \rightarrow \alpha \in R$, $\alpha \notin V$ and $B \in$ Chain(A)
  Note: A∈Chain(A) so all its non unit-rules are included

Non-Productive Symbols

- Let $G = (V, \Sigma, R, S)$ be an arbitrary CFG that has had its λ-rules and unit-rules removed
- Non-productive non-terminal symbols never lead to a terminal string (not productive)
- Productive(G) is computed by
  Productive(G) $\supseteq \{ A \mid A \to \alpha, \alpha \in \Sigma^* \}$
  Repeat
  Productive(G) $\supseteq \{ B \mid B \to \alpha, \alpha \in (\Sigma \cup \text{Productive})^* \}$
  until no new symbols are added
- Keep only those rules that involve productive symbols
- If no rules remain, grammar generates nothing

Unreachable Symbols

- Let G = (V, Σ, R, S) be an arbitrary CFG that has had its λ-rules, unit-rules and non-productive symbols removed
- Unreachable symbols are ones that are inaccessible from start symbol
- We compute the complement (Useful)
- Useful(G) is computed by
  Useful(G) $\supseteq$ { S }
  Repeat
  Useful(G) $\supseteq$ $\{ C \mid B \rightarrow \alpha C \beta,\ C \in V \cup \Sigma,\ B \in \text{Useful}(G) \}$
  until no new symbols are added
- Keep only those rules that involve useful symbols
- If no rules remain, grammar generates nothing

Reduced CFG

- A reduced CFG is one without λ-rules (except possibly for start symbol), no unit-rules, no non-productive symbols and no useless symbols

CFG to CNF

- Let $G = (V, \Sigma, R, S)$ be arbitrary reduced CFG
- Define $G' = (V \cup \{<a> | a \in \Sigma\}, \Sigma, R, S)$
- Add the rules $<a> \rightarrow a$, for all $a \in \Sigma$
- For any rule, $A \rightarrow \alpha$, $|\alpha| > 1$, change each terminal symbol, a, in α to the non-terminal $<a>$
- Now, for each rule $A \rightarrow BC\alpha$, $|\alpha| > 0$, introduce the new non-terminal $B<C\alpha>$, and replace the rule $A \rightarrow BC\alpha$ with the rule $A \rightarrow B<C\alpha>$ and add the rule $<C\alpha> \rightarrow C\alpha$
- Iteratively apply the above step until all rules are in CNF

Example of CNF Conversion

Starting Grammars

- $L = \{ a^i b^j c^k \mid i=j \text{ or } j=k \}$
- $G = (\{S,A,<B=C>,C,<A=B>\}, \{a,b\}, R, S)$
- R:
  - $S \rightarrow A \mid C$
  - $A \rightarrow a\ A \mid <B=C>$
  - $<B=C> \rightarrow b\ <B=C>\ c \mid \lambda$
  - $C \rightarrow C\ c \mid <A=B>$
  - $<A=B> \rightarrow a\ <A=B>\ b \mid \lambda$

Remove Null Rules

- Nullable = {<B=C>, <A=B>, A, C, S}
  - S’ → S | λ // S’ added to V
  - S → A | C
  - A → a A | a |<B=C>
  - <B=C> → b <B=C> c | b c
  - C → C c | c | <A=B>
  - <A=B> → a <A=B> b | ab

Remove Unit Rules

- Chains={[S’:S’,S,A,C,<A=B>,<B=C>],[S:S,A,C,<A=B>,<B=C>], [A:A,<B=C>],[C:C,<B=C>],[<B=C>:<B=C>], [<A=B>:<A=B>]}
  - S’ → λ | aA | a | b<B=C>c | bc | Cc | c | a<A=B>b | ab
  - S → aA | a | b<B=C>c | bc | Cc | c | a<A=B>b | ab
  - A → aA | a | b<B=C>c | bc
  - <B=C> → b<B=C>c | bc
  - C → Cc | c | a<A=B>b | ab
  - <A=B> → a<A=B>b | ab

Remove Useless Symbols

- All non-terminal symbols are productive (lead to terminal string)

- S is useless as it is unreachable from S' (new start).
- All other symbols are reachable from S'

Normalize rhs as CNF

- S’ → λ | <a>A | a | <b><<B=C><c>> | <b><c> | C<c> | c | <a><<A=B><b>> | <a><b>
- A → <a>A | a |<b><<B=C><c>> | <b><c>
- <B=C> → <b><<B=C><c>> | <b><c>
- C → C<c> | c | <a><<A=B><b>> | <a><b>
- <A=B> → <a> <<A=B><b>> | <a><b>
- <<B=C><c>> → <B=C><c>
- <<A=B><b>> → <A=B><b>
- <a> → a
- <b> → b
- <c> → c

CKY (Cocke, Kasami, Younger) $O(N^3)$ PARSING

Dynamic Programming

To solve a given problem, we solve small parts of the problem (subproblems), then combine the solutions of the subproblems to reach an overall solution.

The Parsing problem for arbitrary CFGs was elusive, in that its complexity was unknown until the late 1960s. In the meantime, theoreticians developed notion of simplified forms that were as powerful as arbitrary CFGs. The one most relevant here is the Chomsky Normal Form – CNF. It states that the only rule forms needed are:

$A \rightarrow$	BC	where B and C are non-terminals
$A \rightarrow$	a	where a is a terminal

This is provided the string of length zero is not part of the language.

CKY (Bottom-Up Technique)

Let the input string be a sequence of *n* letters a_1 ... a_n.

Let the grammar contain *r* terminal and nonterminal symbols R_1 ... R_r,

Let R_1 be the start symbol.

Let P[n,n,r] be an array of Booleans. Initialize all elements of P to false.

For each i = 1 to n

    For each unit production $R_j \rightarrow a_i$, set P[i,1,j] = true.

For each i = 2 to n

    For each j = 1 to n-i+1

        For each k = 1 to i-1

            For each production R_A -> R_B R_C

                If P[j,k,B] and P[j+k,i-k,C] then set P[j,i,A] = true

If P[1,n,1] is true then a_1 ... a_n is member of language

else a_1 ... a_n is not member of language

CKY Parser

Present the **CKY** recognition matrix for the string **abba** assuming the Chomsky Normal Form grammar, **G = ({S,A,B,C,D,E}, {a,b}, R, S)**, specified by the rules **R**:

S → AB | BA
A → CD | a
B → CE | b
C → a | b
D → AC
E → BC

	a	b	b	a
1	A,C	B,C	B,C	A,C
2	S,D	E	S,E	
3	B	B		
4	S,E			

2^{nd} CKY Example

E → E F | M E | P E | a
F → M F | P F | M E | P E
P → +
M → –

	a	–	a	+	a	–	a
1	E	M	E	P	E	M	E
2		E, F		E, F		E, F	
3	E		E		E		
4		E, F		E, F			
5	E		E				
6		E, F					
7	E						

Practice CFGs

1. Write a CFG for the following languages:
   L = $\{a^n b^m c^t \mid n < m \text{ or } m > t \text{ or } n = t\}$

2. Convert the following grammar to a CNF equivalent grammar. Show all steps.
   G = ({ S, S1, S2, B, C} , { a , b, c } , R , S), where **R** is:
   $S \to S1 \mid S2$
   $S1 \to a\ S1\ b \mid S1\ b \mid b$
   $S2 \to c\ C\ a\ B$
   $C \to c\ C\ a \mid C\ a \mid a$
   $B \to a\ B\ b \mid \lambda$

3. Present the **CKY** recognition matrix for the string **b a a b a** assuming the Chomsky Normal Form grammar **G = ({ S,T, B } , { a,b } , R, S)**, where **R** is specified by the rules
   $S \to S\ T \mid T\ S \mid a$
   $T \to B\ S \mid b$
   $B \to B\ T \mid SS \mid b$

CFL Pumping Lemma Concept

- Let L be a context free language the there is CNF grammar
  G = (V, Σ, R, S) such that *L*(G) = L.
- As G is in CNF all its rules that allow the string to grow are of the form
  $A \rightarrow BC$, and thus growth has a binary nature.
- Any sufficiently long string z in L will have a parse tree that must have deep branches to accommodate z's growth.
- Because of the binary nature of growth, the width of a tree with maximum branch length k at its deepest nodes is at most 2^k; moreover, if the frontier of the tree is all terminal, then the string so produced is of length at most 2^{k-1}; since the last rule applied for each leaf is of the form $A \rightarrow a$.
- Any terminal branch in a derivation tree of height > |V| has more than |V| internal nodes labelled with non-terminals. The "pigeon hole principle" tells us that whenever we visit |V| +1 or more nodes, we must use at least one variable label more than once. This creates a self-embedding property that is key to the repetition patterns that occur in the derivation of sufficiently long strings.

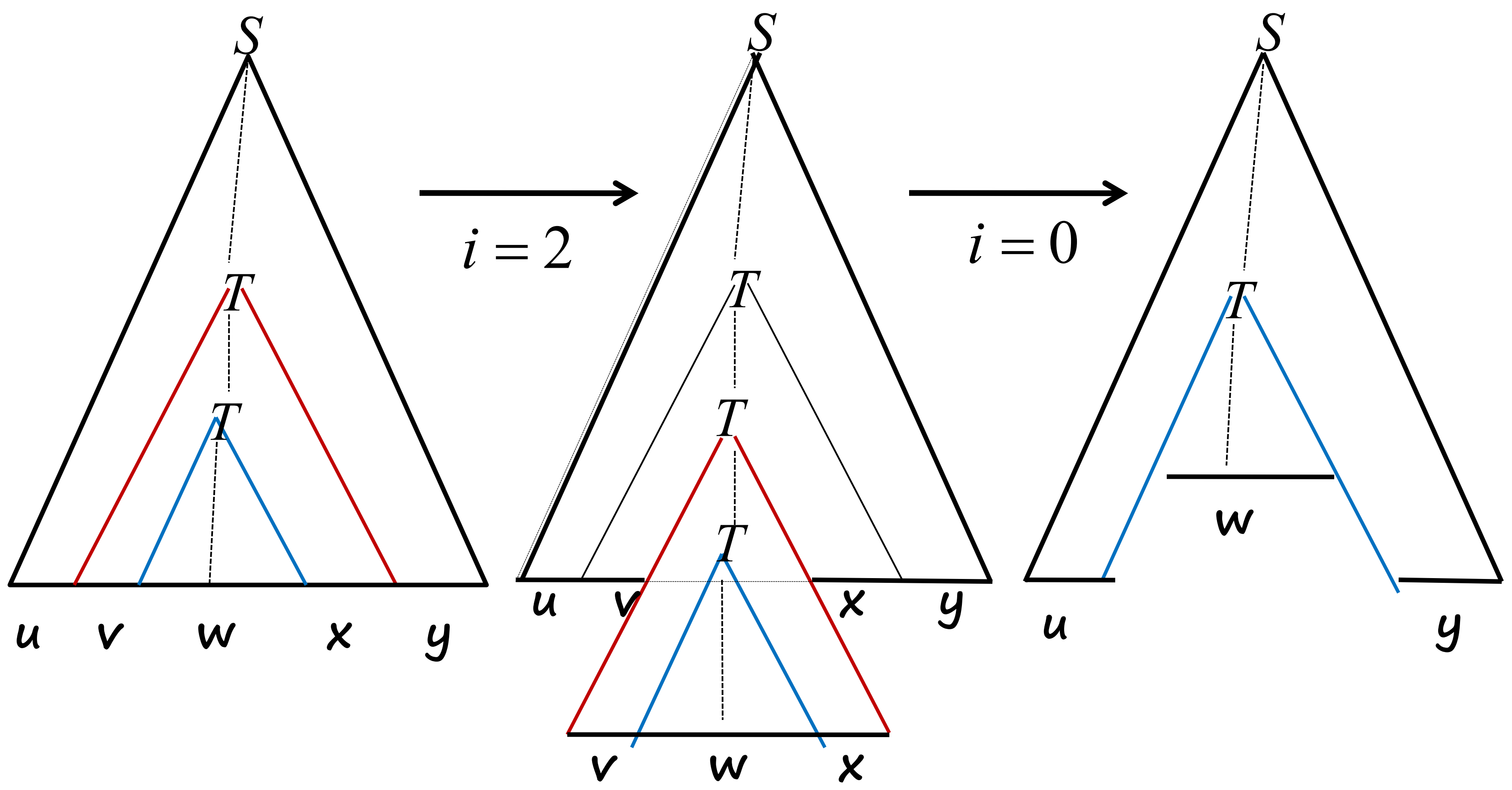
Pumping Lemma For CFL

- Let L be a CFL then there exists an $N>0$ such that, if $z \in L$ and $|z| \geq N$, then z can be written in the form uvwxy, where $|vwy| \leq N$, $|vx|>0$, and for all $i \geq 0$, $uv^iwx^iy \in L$.
- This means that interesting context free languages (infinite ones) have a self-embedding property that is symmetric around some central area, unlike regular where the repetition has no symmetry and occurs at the start.

Pumping Lemma Proof

- If L is a CFL then it is generated by some CNF grammar, $G = (V, \Sigma, R, S)$. Let $|V| = k$. For any string z, such that $|z| \geq N = 2^k$, the derivation tree for z based on G must have a branch with at least k+1 nodes labelled with variables from G.
- By the Pigeon Hole Principle at least two of these labels must be the same. Let the first repeated variable be T and consider the last two instances of T on this path.
- Let $z = uvwxy$, where $S \Rightarrow^* uTy \Rightarrow^* uvTxy \Rightarrow^* uvwxy$
- Clearly, then, we know $S \Rightarrow^* uTy$; $T \Rightarrow^* vTx$; and $T \Rightarrow^* w$
- But then, we can start with $S \Rightarrow^* uTy$; repeat $T \Rightarrow^* vTx$ zero or more times; and then apply $T \Rightarrow^* w$.
- But then, $S \Rightarrow^* uv^iwx^iy$ for all $i \geq 0$, and thus $uv^iwx^iy \in L$, for all $i \geq 0$.

Visual Support of Proof



Lemma's Adversarial Process

- Assume L = $\{a^n b^n c^n \mid n>0\}$ is a CFL
- P.L.: Provides N>0 We CANNOT choose N; that's the P.L.'s job
- Our turn: Choose $a^N b^N c^N \in L$ We get to select a string in L
- P.L.: $a^N b^N c^N = uvwxy$, where $|vwx| \leq N$, $|vx|>0$, and for all $i \geq 0$, $uv^iwx^iy \in L$ We CANNOT choose split, but P.L. is constrained by N
- Our turn: Choose i=0. We have the power here
- P.L: Two cases:
  (1) vwx contains some a's and maybe some b's. Because $|vwx| \leq N$, it cannot contain c's if it has a's. i=0 erases some a's but we still have N c's so $uwy \notin L$
  (2) vwx contains no a's. Because $|vx|>0$, vx contains some b's or c's or some of each. i=0 erases some b's and/or c's but we still have N a's so $uwy \notin L$
- CONTRADICTION, so L is NOT a CFL

Practice CFL Pumping Lemma

1. Write a CFG to show the language is a CFL or use the Pumping Lemma for CFLs to prove that it is not for each of the following.
   a) $L = \{ a^i b^j \mid j > i^3 \ , I>0\}$
   b) $L = \{a^i b^j \mid j < 3*i \ , i>0\}$

2. Consider the context-free grammar **$G = \{ \{S\}, \{a,b\}, R, S \}$**
   R:
   $S \rightarrow a\ S\ b\ S\ b\ S \mid b\ S\ a\ S\ b\ S \mid b\ S\ b\ S\ a\ S \mid \lambda$
   Provide a proof that shows
   $L = \{ w \mid |w_b| = 2|w_a| \}$
   That is, the number of **b**'s in **w** is twice that of the **a**'s
   You will need to provide an inductive proof in both directions

Non-Closure

- Intersection ($\{ a^n b^n c^n \mid n \geq 0 \}$ is not a CFL)
  $\{ a^n b^n c^n \mid n \geq 0 \} =$
  $\{ a^n b^n c^m \mid n,m \geq 0 \} \cap \{ a^m b^n c^n \mid n,m \geq 0 \}$
  Both of the above are CFLs
- Complement
  If closed under complement then would be closed under Intersection as
  $A \cap B = \sim(\sim A \cup \sim B)$

Max and Min of CFL

- Consider the two operations on languages max and min, where
  - max(L) = $\{ x \mid x \in L$ and, for no non-null y does $xy \in L \}$ and
  - min(L) = $\{ x \mid x \in L$ and, for no proper prefix of x, y, does $y \in L \}$
- Describe the languages produced by max and min. for each of :
  - $L1 = \{ a^i b^j c^k \mid k \le i \text{ or } k \le j \}$ CFL
    - max(L1) = $\{ a^i b^j c^k \mid k = \max(i, j) \}$ Non-CFL
    - min(L1) = $\{ \lambda \}$ (string of length 0) Regular
  - $L2 = \{ a^i b^j c^k \mid k > i \text{ or } k > j \}$ CFL
    - max(L2) = $\{ \}$ (empty) Regular
    - min(L2) = $\{ a^i b^j c^k \mid k = \min(i, j)+1 \}$ Non-CFL
- max(L1) shows CFL not closed under max
- min(L2) shows CFL not closed under min

Complement of ww

- Let L = { ww | w ∈ $\{a,b\}^+$ }. L is not a CFL
- Consider L's complement, it must be of form xayx'by' or xbyx'ay', where |x|=|x'| and |y|=|y'|
- The above reflects that this language has one "transcription error"
- This seems really hard to write a CFG but it's all a matter of how you view it
- We don't care about what precedes or follows the errors so long as the lengths are right
- Thus, we can view above as xax'yby' or xbx'y'ay', where |x|=|x'| and |y|=|y'|
- The grammar for this has rules
  S → AB | BA ; A → XAX | a ; B → XBX | b
  X → a | b

Solvable CFL Problems

- Let L be an arbitrary CFL generated by CFG G with start symbol S then the following are all decidable
  - Is w in L? — Run CKY
If S in final cell then w∈L
  - Is L empty (non-empty)? — Reduce G
If no rules left then empty
  - Is L finite (infinite)? — Reduce G
Run DFS(S)
If no loops then finite

Formalization of PDA

- $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$
- Q is finite set of states
- Σ is finite input alphabet
- Γ is finite set of stack symbols
- $\delta : Q \times \Sigma_e \times \Gamma_e \rightarrow 2^{Q \times \Gamma^*}$ is transition function
  – Note: Can limit stack push to Γ_e but it's equivalent!!
- $Z_0 \in \Gamma$ is an optional initial symbol on stack
- $F \subseteq Q$ is final set of states and can be omitted for some notions of a PDA

Notion of ID for PDA

- An instantaneous description for a PDA is $[q, w, \gamma]$ where
  - q is current state
  - w is remaining input
  - γ is contents of stack (leftmost symbol is top)
- Single step derivation is defined by
  - $[q,ax,Z\alpha] \vdash [p,x,\beta\alpha]$ if $\delta(q,a,Z)$ contains (p,β)
- Multistep derivation ($\vdash^*$) is reflexive transitive closure of single step.

Language Recognized by PDA

- Given $A = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$
  there are three senses of recognition
- By final state
  $L(A) = \{w \mid [q_0,w,Z_0] \vdash^* [f,\lambda,\beta]\}$, where $f \in F$
- By empty stack
  $N(A) = \{w \mid [q_0,w,Z_0] \vdash^* [q,\lambda,\lambda]\}$
- By empty stack and final state
  $E(A) = \{w \mid [q_0,w,Z_0] \vdash^* [f,\lambda,\lambda]\}$, where $f \in F$

Top Down Parsing by PDA

- Given $G = (V, \Sigma, R, S)$, define
  $A = (\{q\}, \Sigma, \Sigma \cup V, \delta, q, S, \phi)$
- $\delta(q,a,a) = \{(q,\lambda)\}$ for all $a \in \Sigma$
- $\delta(q,\lambda,A) = \{(q,\alpha) \mid A \rightarrow \alpha \in R \text{ (guess)} \}$
- $N(A) = \mathscr{L}(G)$

- Give just one state, this is essentially stateless, except for stack

Top Down Parsing by PDA

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid Int$

- $\delta(q,+,+) = \{(q,\lambda)\}$, $\delta(q,*,*) = \{(q,\lambda)\}$,
- $\delta(q,Int,Int) = \{(q,\lambda)\}$,
- $\delta(q,(,() = \{(q,\lambda)\}$, $\delta(q,),)) = \{(q,\lambda)\}$
- $\delta(q,\lambda,E) = \{(q,E+T), (q,T)\}$
- $\delta(q,\lambda,T) = \{(q,T*F), (q,F)\}$
- $\delta(q,\lambda,F) = \{(q,(E)), (q,Int)\}$

Bottom Up Parsing by PDA

- Given $G = (V, \Sigma, R, S)$, define
  $A = (\{q,f\}, \Sigma, \Sigma \cup V \cup \{\$\}, \delta, q, \$, \{f\})$
- $\delta(q,a,\lambda) = \{(q,a)\}$ for all $a \in \Sigma$, SHIFT
- $\delta(q,\lambda,\alpha^R) \supseteq \{(q,A)\}$ if $A \rightarrow \alpha \in R$, REDUCE
  Cheat: looking at more than top of stack
- $\delta(q,\lambda,S) \supseteq \{(f,\lambda)\}$
- $\delta(f,\lambda,\$) = \{(f,\lambda)\}$, ACCEPT
- $E(A) = \mathscr{L}(G)$
- Could also do $\delta(q,\lambda,S\$) \supseteq \{(q,\lambda)\}$, $N(A) = \mathscr{L}(G)$

Bottom Up Parsing by PDA

$E \rightarrow E + T \mid T$

$T \rightarrow T * F \mid F$

$F \rightarrow (E) \mid Int$

- $\delta(q,+,\lambda)=\{(q,+)\}$, $\delta(q,*,\lambda)=\{(q,*)\}$, $\delta(q,Int,\lambda)=\{(q,Int)\}$, $\delta(q,(,\lambda)=\{(q,()\}$, $\delta(q,),\lambda)=\{(q,))\}$
- $\delta(q,\lambda,T{+}E) = \{(q,E)\}$, $\delta(q,\lambda,T) \supseteq \{(q,E)\}$
- $\delta(q,\lambda,F{*}T) \supseteq \{(q,T)\}$, $\delta(q,\lambda,F) \supseteq \{(q,T)\}$
- $\delta(q,\lambda,)E() \supseteq \{(q,F)\}$, $\delta(q,\lambda,Int) \supseteq \{(q,F)\}$
- $\delta(q,\lambda,E) \supseteq \{(f,\lambda)\}$
- $\delta(f,\lambda,\$) = \{(f,\lambda)\}$
- $E(A) = \mathscr{L}(G)$

Challenge

- Use the two recognizers on some sets of expressions like
  - 5 + 7 * 2
  - 5 * 7 + 2
  - (5 + 7) * 2

Converting a PDA to CFG

- Book has one approach; here is another
- Let A = (Q, Σ, Γ, δ, q_0, Z, F) accept L by empty stack and final state
- Define A' = ($Q \cup \{q_0', f\}$, Σ, $\Gamma \cup \{\$\}$, δ', q_0', \$, {f}) where
  - $\delta'(q_0', \lambda, \$) = \{(q_0, \text{PUSH}(Z))$ or in normal notation $\{(q_0, Z\$)\}$
  - δ' does what δ does but only uses PUSH and POP instructions, always reading top of stack
    Note1: we need to consider using the \$ for cases of the original machine looking at empty stack, when using λ for stack check. This guarantees we have top of stack until very end.
    Note2: If original adds stuff to stack, we do pop, followed by a bunch of pushes.
  - We add (f, λ) = (f, POP) to $\delta'(q_f, \lambda, \$)$ whenever q_f is in F, so we jump to a fixed final state.
- Now, wlog, we can assume our PDA uses only POP and PUSH, has just one final state and accepts by empty stack and final state. We will assume the original machine is of this form and that its bottom of stack is \$.
- Define G = (V, Σ, R, S) where
  - V = {S} $\cup$ { <q, X, p> | q,p $\in$ Q, X $\in$ Γ }
  - R on next page

Rules for PDA to CFG

- R contains rules as follows:
  $S \to <q_0,\$,f>$ where $F = \{f\}$
  meaning: want to generate w whenever
  $[q_0,w,\$] \vdash^* [f,\lambda,\lambda]$
- Remaining rules are:
  $<q,X,p> \to a<s,Y,t><t,X,p>$
  whenever $\delta(q,a,X) \supseteq \{(s,PUSH(Y))\}$
  $<q,X,p> \to a$
  whenever $\delta(q,a,X) \supseteq \{(p,POP)\}$
- Want $<q,X,p> \Rightarrow^* w$ when $[q,w,X] \vdash^* [p,\lambda,\lambda]$

Greibach Normal Form

- Each rule of a GNF is constrained to be of form:
  $A \to a\alpha, \quad A \in V,\ a \in \Sigma,\ \alpha \in V^*$
- If the language contains λ then we allow
  $S \to \lambda$
  and constrain S to not be on right hand side of any rule
- The beauty of this form is that, in a bottom up parse, every step consumes an input character and so parse is linear (if we guess right)
- We will not show details of conversion but it is dependent on starting in CNF and then removing left recursion, both of which we have already shown

Closure Properties

Context Free Languages

Intersection with Regular

- CFLs are closed under intersection with Regular sets
  - To show this we use the equivalence of CFGs generative power with the recognition power of PDAs.
  - Let $A_0 = (Q_0, \Sigma, \Gamma, \delta_0, q_0, \$, F_0)$ be an arbitrary PDA
  - Let $A_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ be an arbitrary DFA
  - Define $A_2 = (Q_0 \times Q_1, \Sigma, \Gamma, \delta_2, <q_0,q_1> \$, F_0 \times F_1)$ where
    - $\delta_2(<q,s>, a, X) \supseteq \{(<q',s'>, \alpha)\}$, $a \in \Sigma \cup \{\lambda\}$, $X \in \Gamma$ iff
      $\delta_0(q, a, X) \supseteq \{(q', \alpha)\}$ and
      $\delta_1(s,a) = s'$ (if $a=\lambda$ then $s' = s$).
  - Using the definition of derivations we see that
    $[<q_0,q_1>, w, \$] \vdash^* [<t,s>, \lambda, \beta]$ in A_2 iff
    $[q_0, w, \$] \vdash^* [t, \lambda, \beta]$ in A_0 and
    $[q_1, w] \vdash^* [s, \lambda]$ in A_1
    But then $w \in \mathcal{F}(A_2)$ iff $t \in F_0$ and $s \in F_1$ iff $w \in \mathcal{F}(A_0)$ and $w \in \mathcal{F}(A_1)$

Substitution

- CFLs are closed under CFL substitution
  - Let $G=(V,\Sigma,R,S)$ be a CFG.
  - Let f be a substitution over Σ such that
    - $f(a) = L_a$ for $a \in \Sigma$
    - $G_a = (V_a,\Sigma_a,R_a,S_a)$ is a CFG that produces L_a.
    - No symbol appears in more than one of V or any V_a
  - Define $G_f = (V \cup_{a\in\Sigma} V_a, \cup_{a\in\Sigma}\Sigma_a, R' \cup_{a\in\Sigma} R_a, S)$
    - $R' = \{ A \rightarrow g(\alpha)$ where $A \rightarrow \alpha$ is in $R \}$
    - $g: (V\cup\Sigma)^* \rightarrow (V \cup_{a\in\Sigma} S_a)^*$
    - $g(\lambda) = \lambda$; $g(B) = B$, $B \in V$; $g(a) = S_a$, $a \in \Sigma$
    - $g(\alpha X) = g(\alpha)\, g(X)$, $|\alpha| > 0$, $X \in V\cup\Sigma$
  - Claim, $f(\mathscr{L}(G)) = \mathscr{L}(G_f)$, and so CFLs closed under substitution and homomorphism.

More on Substitution

- Consider G'_f. If we limit derivations to the rules $R' = \{ A \rightarrow g(\alpha)$ where $A \rightarrow \alpha$ is in $R \}$ and consider only sentential forms over the $\cup_{a \in \Sigma} S_a$, then $S \Rightarrow^* S_{a1}\ S_{a2} \ldots S_{an}$ in G' iff $S \Rightarrow^* a1\ a2 \ldots an$ iff $a1\ a2 \ldots an \in \mathscr{L}(G)$. But, then $w \in \mathscr{L}(G)$ iff $f(w) \in \mathscr{L}(G_f)$ and, thus, $f(\mathscr{L}(G)) = \mathscr{L}(G_f)$.

- Given that CFLs are closed under intersection, substitution, homomorphism and intersection with regular sets, we can recast previous proofs to show that CFLs are closed under
  - Prefix, Suffix, Substring, Quotient with Regular Sets

- Later we will show that CFLs are <u>not</u> closed under Quotient with CFLs.

Context Sensitive

Context Sensitive Grammar

$G = (V, \Sigma, R, S)$ is a PSG where

Each member of R is a rule whose right side is no shorter than its left side.

The essential idea is that rules are length preserving, although we do allow $S \rightarrow \lambda$ so long as S never appears on the right hand side of any rule.

A context sensitive grammar is denoted as a CSG and the language generated is a Context Sensitive Language (CSL).

The recognizer for a CSL is a Linear Bounded Automaton (LBA), a form of Turing Machine (soon to be discussed), but with the constraint that it is limited to moving along a tape that contains just the input surrounded by a start and end symbol.

CSG Example#1

$L = \{ a^n b^n c^n \mid n>0 \}$

$G = (\{A,B,C\}, \{a,b,c\}, R, A)$ where R is

$A \rightarrow aBbc \mid abc$

$B \rightarrow aBbC \mid abC$

Note: $A \Rightarrow aBbc \Rightarrow n\ a^{n+1}(bC)^n\ bc$ // n>0

$Cb \rightarrow bC$ // Shuttle C over to a c

$Cc \rightarrow cc$ // Change C to a c

Note: $a^{n+1}(bC)^n\ bc \Rightarrow^* a^{n+1}b^{n+1}c^{n+1}$

Thus, $A \Rightarrow^* a^n b^n c^n$, $n>0$

CSG Example#2

$L = \{ ww \mid w \in \{0,1\}^+ \}$

$G = (\{S,A,X,Z,<0>,<1>\}, \{0,1\}, R, S)$ where R is

$S \rightarrow 00 \mid 11 \mid 0A<0> \mid aA<1> \mid 1A<1>$

$A \rightarrow 0AZ \mid 1AX \mid 0Z \mid 1X$

$Z0 \rightarrow 0Z$	$Z1 \rightarrow 1Z$	// Shuttle Z (for owe zero)
$X0 \rightarrow 0X$	$X1 \rightarrow 1X$	// Shuttle X (for owe one)
$Z<0> \rightarrow 0<0>$	$Z<1> \rightarrow 1<0>$	// New 0 must be on rhs of old 0/1's
$X<0> \rightarrow 0<1>$	$X<1> \rightarrow 1<1>$	// New 1 must be on rhs of old 0/1's
$<0> \rightarrow 0$		// Guess we are done
$<1> \rightarrow 1$		// Guess we are done

Phrase Structured Grammar

We previously defined PSGs. The language generated by a PSG is a Phrase Structured Language (PSL) but is more commonly called a recursively enumerable (re) language. The reason for this will become evident a bit later in the course.

The recognizer for a PSL (re language) is a Turing Machine, a model of computation we will soon discuss.

HISTORY

The Quest for Mechanizing Mathematics

Hilbert, Russell and Whitehead

- Until 1800's there were no formal systems to reason about mathematical properties
- Major advances in late 1800's/early 1900's
- Axiomatic schemes
  - Axioms plus sound rules of inference
  - Much of focus on number theory
- First Order Predicate Calculus
  - $\forall x \exists y\ [y > x]$
- Second Order (Peano's Axiom)
  - $\forall P\ [[P(0)\ \&\&\ \forall x[P(x) \Rightarrow P(x+1)]] \Rightarrow \forall x P(x)]$

Hilbert

- In 1900 declared there were 23 really important problems in mathematics.
- Belief was that the solutions to these would help address math’s complexity.
- Hilbert’s Tenth asks for an algorithm to find the integral zeros of polynomial equations with integral coefficients. This is now known to be impossible (In 1972, Matiyacevič showed this undecidable).

Hilbert's Belief

- All mathematics could be developed within a formal system that allowed the mechanical creation and checking of proofs.

Gödel

- In 1931 he showed that any first order theory that embeds elementary arithmetic is either incomplete or inconsistent.
- He did this by showing that such a first order theory cannot reason about itself. That is, there is a first order expressible proposition that cannot be either proved or disproved, or the theory is inconsistent (some proposition and its complement are both provable).
- Gödel also developed the general notion of recursive functions but made no claims about their strength.

Turing (Post, Church, Kleene)

- In 1936, each presented a formalism for computability.
  - Turing and Post devised abstract machines and claimed these represented all mechanically computable functions.
  - Church developed the notion of lambda-computability from recursive functions (as previously defined by Gödel and Kleene) and claimed completeness for this model.
- Kleene demonstrated the computational equivalence of recursively defined functions to Post-Turing machines.
- Church's notation was the lambda calculus, which later gave birth to Lisp.

More on Emil Post

- In the 1920's, starting with notation developed by Frege and others in 1880s, Post devised the truth table form we all use now for Boolean expressions (propositional logic). This was a part of his PhD thesis in which he showed the axiomatic completeness of the propositional calculus (all tautologies can be deduced from a finite set of tautologies and a finite set of rules of inference).
- In the late 1930's and the 1940's, Post devised symbol manipulation systems in the form of rewriting rules (precursors to Chomsky's grammars). He showed their equivalence to Turing machines.
- In 1940s, Post showed the complexity (undecidability) of determining what is derivable from an arbitrary set of propositional axioms.

Computability

The study of what can/cannot be done via purely mechanical means

Basic Definitions

The Preliminaries

Goals of Computability

- Provide precise characterizations (computational models) of the class of effective procedures / algorithms.
- Study the boundaries between complete and incomplete models of computation.
- Study the properties of classes of solvable and unsolvable problems.
- Solve or prove unsolvable open problems.
- Determine reducibility and equivalence relations among unsolvable problems.
- Our added goal is to apply these techniques and results across multiple areas of Computer Science.

Effective Procedure

- *A process whose execution is clearly specified to the smallest detail*
- Such procedures have, among other properties, the following:
  - Processes must be finitely describable and the language used to describe them must be over a finite alphabet.
  - The current state of the machine model must be finitely presentable.
  - Given the current state, the choice of actions (steps) to move to the next state must be easily determinable from the procedure's description.
  - Each action (step) of the process must be capable of being carried out in a finite amount of time.
  - The semantics associated with each step must be clear and unambiguous.

Algorithm

- *An effective procedure that halts on all input*
- The key term here is "*halts on all input*"
- By contrast, an effective procedure may halt on all, none or some of its input.
- The domain of an algorithm is its entire domain of possible inputs.

Sets and Decision Problems

- Set -- A collection of atoms from some universe **U**. **Ø** denotes the empty set.
- (Decision) Problem -- A set of questions, each of which has answer "yes" or "no".

Categorizing Problems (Sets)

- Solvable or Decidable -- A problem P is said to be solvable (decidable) if there exists an algorithm F which, when applied to a question q in P, produces the correct answer ("yes" or "no").
- Solved -- A problem P is said to solved if P is solvable and we have produced its solution.
- Unsolved, Unsolvable (Undecidable) -- Complements of above

Categorizing Problems (Sets) # 2

- Recursively enumerable -- A set S is recursively enumerable (re) if S is empty (S = Ø) or there exists an algorithm F, over the natural numbers ***N***, whose range is exactly S. A problem is said to be re if the set associated with it is re.
- Semi-Decidable -- A problem is said to be semi-decidable if there is an effective procedure F which, when applied to a question q in P, produces the answer "yes" if and only if q has answer "yes". F need not halt if q has answer "no".
- Semi-decidable is the same as the notion of recognizable used in the text.

Immediate Implications

- **P** solved implies **P** solvable implies **P** semi-decidable (re, recognizable).
- **P** non-re implies **P** unsolvable implies **P** unsolved.
- **P** finite implies **P** solvable.

Slightly Harder Implications

- **P** enumerable iff **P** semi-decidable.
- **P** solvable iff both $\mathbf{S_P}$ and ($\mathbf{U} — \mathbf{S_P}$) are re (semi-decidable).

- We will prove these later.

Existence of Undecidables

- A counting argument
  - The number of mappings from *N* to *N* is at least as great as the number of subsets of *N*. But the number of subsets of *N* is uncountably infinite ($\aleph_1$). However, the number of programs in any model of computation is countably infinite ($\aleph_0$). This latter statement is a consequence of the fact that the descriptions must be finite and they must be written in a language with a finite alphabet. In fact, not only is the number of programs countable, it is also effectively enumerable; moreover, its membership is decidable.
- A diagonalization argument
  - Will be shown later in class

Hilbert's Tenth

Diophantine Equations are Unsolvable

One Variable Diophantine Equations are Solvable

Hilbert's 10^{th}

- In 1900 declared there were 23 really important problems in mathematics.
- Belief was that the solutions to these would help address math's complexity.
- Hilbert's Tenth asks for an algorithm to find the integral roots of polynomials with integral coefficients. For example
  $6x^3yz^2 + 3xy^2 – x^3 – 10 = 0$ has roots
  $x = 5; y = 3; z = 0$
- This is now known to be impossible (In 1970, Matiyacevič showed this undecidable).

Hilbert's 10^{th} is Semi-Decidable

- Consider over one variable: $P(x) = 0$
- Can semi-decide by plugging in 0, 1, -1, 2, -2, 3, -3, …
- This terminates and says "yes" if $P(x)$ evaluates to 0, eventually. Unfortunately, it never terminates if there is no x such that $P(x) =0$.
- Can easily extend to $P(x_1,x_2,..,x_k) = 0$.

$P(x) = 0$ is Decidable

- $c_n x^n + c_{n-1} x^{n-1} + \ldots + c_1 x + c_0 = 0$
- $x^n = -(c_{n-1} x^{n-1} + \ldots + c_1 x + c_0)/c_n$
- $|x^n| \le c_{max}(|x^{n-1}| + \ldots + |x| + 1|)/|c_n|$
- $|x^n| \le c_{max}(n |x^{n-1}|)/|c_n|$, since $|x|\ge 1$
- $|x| \le n \times c_{max}/|c_n|$

$P(x) = 0$ is Decidable

- Can bound the search to values of x in range [$\pm$ $n * (c_{max} / c_n)$], where
  n = highest order exponent in polynomial
  c_{max} = largest absolute value coefficient
  c_n = coefficient of highest order term
- Once we have a search bound and we are dealing with a countable set, we have an algorithm to decide if there is an x.
- Cannot find bound when more than one variable, so cannot extend to $P(x_1,x_2,..,x_k) = 0$.

Undecidability

We Can’t Do It All

Classic Unsolvable Problem

Given an arbitrary program ***P***, in some language ***L***, and an input ***x*** to ***P***, will ***P*** eventually stop when run with input ***x***?

The above problem is called the "**Halting Problem**." It is clearly an important and practical one – wouldn't it be nice to not be embarrassed by having your program run "forever" when you try to do a demo?

Unfortunately, there's a fly in the ointment as one can prove that no algorithm can be written in ***L*** that solves the halting problem for ***L***.

Some terminology

We will say that a procedure, ***f***, converges on input ***x*** if it eventually halts when it receives ***x*** as input. We denote this as ***f(x)***↓.

We will say that a procedure, ***f***, diverges on input ***x*** if it never halts when it receives ***x*** as input. We denote this as ***f(x)***↑.

Of course, if ***f(x)***↓ then ***f*** defines a value for ***x***. In fact we also say that ***f(x)*** is defined if ***f(x)***↓ and undefined if ***f(x)***↑.

Finally, we define the domain of ***f*** as **{*x* | *f(x)*↓}**.
The range of ***f*** is **{*y* | *f(x)*↓** and ***f(x)* = *y*** }.

Halting Problem

Assume we can decide the halting problem. Then there exists some total function **Halt** such that

$$
\textbf{Halt}(x,y) = \begin{cases} 1 & \text{if } \varphi_x(y)\downarrow \\ 0 & \text{if } \varphi_x(y)\uparrow \end{cases}
$$

Here, we have numbered all programs and φ_x refers to the **x**-th program in this ordering. Now we can view Halt as a mapping from $\aleph$ into $\aleph$ by treating its input as a single number representing the pairing of two numbers via the one-one onto function

$$\textbf{pair}(x,y) = \langle x,y\rangle = 2^x (2y + 1) - 1$$

with inverses

$$\langle z\rangle_1 = \log_2(z+1)$$

$$\langle z\rangle_2 = (((z + 1) \,//\, 2^{\langle z\rangle_1}) - 1) \,//\, 2$$

The Contradiction

Now if **Halt** exist, then so does **Disagree**, where

$$\textbf{Disagree}(x) = \begin{cases} \mathbf{0} & \text{if } \mathbf{Halt(x,x) = 0}\text{, i.e, if } \varphi_x\ \mathbf{(x)} \uparrow \\ \mu y\ (y == y+1) & \text{if } \mathbf{Halt(x,x) = 1}\text{, i.e, if } \varphi_x\ \mathbf{(x)} \downarrow \end{cases}$$

Since **Disagree** is a program from ℵ into ℵ , **Disagree** can be reasoned about by **Halt**. Let **d** be such that **Disagree** = φ_d, then

Disagree(d) is defined ⇔ **Halt(d,d) = 0**

⇔ φ_d **(d)** ↑

⇔ **Disagree(d)** is undefined

But this means that **Disagree** contradicts its own existence. Since every step we took was constructive, except for the original assumption, we must presume that the original assumption was in error. Thus, the **Halting Problem** is not solvable.

Halting is recognizable

While the **Halting Problem** is not solvable, it is re, recognizable or semi-decidable.

To see this, consider the following semi-decision procedure. Let ***P*** be an arbitrary procedure and let ***x*** be an arbitrary natural number. Run the procedure ***P*** on input ***x*** until it stops. If it stops, say "yes." If **P** does not stop, we will provide no answer. This semi-decides the **Halting Problem**. Here is a procedural description.

```
Semi_Decide_Halting() {
    Read P, x;
    P(x);
    Print "yes";
}
```

Why not just algorithms?

A question that might come to mind is why we could not just have a model of computation that involves only programs that halt for all input. Assume you have such a model – our claim is that this model must be incomplete!

Here's the logic. Any programming language needs to have an associated grammar that can be used to generate all legitimate programs. By ordering the rules of the grammar in a way that generates programs in some lexical or syntactic order, we have a means to recursively enumerate the set of all programs. Thus, the set of procedures (programs) is re. using this fact, we will employ the notation that φ_x is the **x**-th procedure and φ_x**(y)** is the **x**-th procedure with input **y**. We also refer to **x** as the procedure's index.

The universal machine

First, we can all agree that any complete model of computation must be able to simulate programs in its own language. We refer to such a simulator (interpreter) as the Universal machine, denote **Univ**. This program gets two inputs. The first is a description of the program to be simulated and the second of the input to that program. Since the set of programs in a model is re, we will assume both arguments are natural numbers; the first being the index of the program. Thus,

Univ(x,y) $= \varphi_x(y)$

Non-re Problems

- There are even “practical” problems that are worse than unsolvable -- they’re not even semi-decidable.
- The classic non-re problem is the **Uniform Halting Problem**, that is, the problem to decide of an arbitrary effective procedure **P**, whether or not **P** is an algorithm.
- Assume that the algorithms can be enumerated, and that **F** accomplishes this. Then

  $\mathbf{F(x) = F_x}$

  where $\mathbf{F_0}$, $\mathbf{F_1}$, $\mathbf{F_2}$, … is a list of indexes of all and only the algorithms

The Contradiction

- Define $\quad$ **G(x) = Univ (F(x) , x) + 1 =** $\varphi_{F(x)}$**(x) =** $F_x(x)$ **+ 1**
- But then **G** is itself an algorithm. Assume it is the **g**-th one

  $$F(g) = F_g = G$$

  Then, $\quad$ $$G(g) = F_g(g) + 1 = G(g) + 1$$

- But then **G** contradicts its own existence since **G** would need to be an algorithm.
- This cannot be used to show that the effective procedures are non-enumerable, since the above is not a contradiction when **G(g)** is undefined. In fact, we already have shown how to enumerate the (partial) recursive functions.

Consequences

- To capture all the algorithms, any model of computation must include some procedures that are not algorithms.

- Since the potential for non-termination is required, every complete model must have some for form of iteration that is potentially unbounded.

- This means that simple, well-behaved for-loops (the kind where you can predict the number of iterations on entry to the loop) are not sufficient. While type loops are needed, even if implicit rather than explicit.

Insights

Non-re nature of algorithms

- No generative system (e.g., grammar) can produce descriptions of all and only algorithms
- No parsing system (even one that rejects by divergence) can accept all and only algorithms

- Of course, if you buy Church’s Theorem, the set of all procedures can be generated. In fact, we can build an algorithmic acceptor of such programs.

Many unbounded ways

- How do you achieve divergence, i.e., what are the various means of unbounded computation in each of our models?
- GOTO: Turing Machines and Register Machines
- Minimization: Recursive Functions
  - Why not just simple finite iteration or recursion?
- Fixed Point: Ordered Petri Nets, (Ordered) Factor Replacement Systems

Non-determinism

- It sometimes doesn't matter
  - Turing Machines, Finite State Automata, Linear Bounded Automata

- It sometimes helps
  - Push Down Automata

- It sometimes hinders
  - Factor Replacement Systems, Petri Nets

Models of Computation

Turing Machines

Register Machines

Factor Replacement Systems

Recursive Functions

Turing Machines

1st Model

A Linear Memory Machine

Typical Textbook Description

- A Turing machine is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$
- Q is finite set of states
- Σ, is a finite input alphabet not containing the blank symbol $\sqcup$
- Γ is finite set of tape symbols that includes Σ and $\sqcup$ commonly $\Gamma = \Sigma \cup \{\sqcup\}$
- $\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{R,L\}$
- q_0 starts, q_{accept} accepts, q_{reject} rejects

Turing versus Post

- The Turing description just given requires you to write a new symbol and move off the current tape square
- Post had a variant where
  $\delta: Q \times \Gamma \rightarrow Q \times (\Gamma \cup \{R,L\})$
- Here, you either write or move, not both
- Also, Post did not have an accept or reject state – acceptance is giving an answer of 1; rejection is 0; this treats decision procedures as predicates (functions that map input into {0,1})
- The way we stop our machines from running is to omit actions for some discriminants making the transition function partial
- I tend to use Post's notation and to create macros so machines are easy to create
- I am not a fan of having you build Turing tables

Basic Description

- We will use a simplified form that is a variant of Post's models.
- Here, each machine is represented by a finite set of states Q, the simple alphabet {0,1}, where 0 is the blank symbol, and each state transition is defined by a 4-tuple of form

  q a X s

  where q a is the discriminant based on current state q, scanned symbol a; X can be one of {R, L, 0, 1}, signifying move right, move left, print 0, or 1; and s is the new state.
- Limiting the alphabet to {0,1} is not really a limitation. We can represent a k-letter alphabet by encoding the j-th letter via j 1's in succession. A 0 ends each letter, and two 0's ends a word.
- We rarely write quads. Rather, we typically will build machines from simple forms.

Base Machines

- R -- move right over any scanned symbol
- L -- move left over any scanned symbol
- 0 -- write a 0 in current scanned square
- 1 -- write a 1 in current scanned square
- We can then string these machines together with optionally labeled arc.
- A labeled arc signifies a transition from one part of the composite machine to another, if the scanned square's content matches the label. Unlabeled arcs are unconditional. We will put machines together without arcs, when the arcs are unlabeled.

Useful Composite Machines

$\mathcal{R}$ -- move right to next 0 (not including current square)

$\ldots\underline{?}11\ldots10\ldots \Rightarrow \ldots?11\ldots1\underline{0}\ldots$

$\mathcal{L}$ -- move left to next 0 (not including current square)

$\ldots011\ldots1\underline{?}\ldots \Rightarrow \ldots\underline{0}11\ldots1?\ldots$

Commentary on Machines

- These machines can be used to move over encodings of letters or encodings of unary based natural numbers.
- In fact, any effective computation can easily be viewed as being over natural numbers. We can get the negative integers by pairing two natural numbers. The first is the sign (0 for +, 1 for -). The second is the magnitude.

Computing with TMs

A reasonably standard definition of a Turing computation of some n-ary function F is to assume that the machine starts with a tape containing the n inputs, x1, … , xn in the form

$$\ldots 0 1^{x1} 0 1^{x2} 0 \ldots 0 1^{xn} \underline{0} \ldots$$

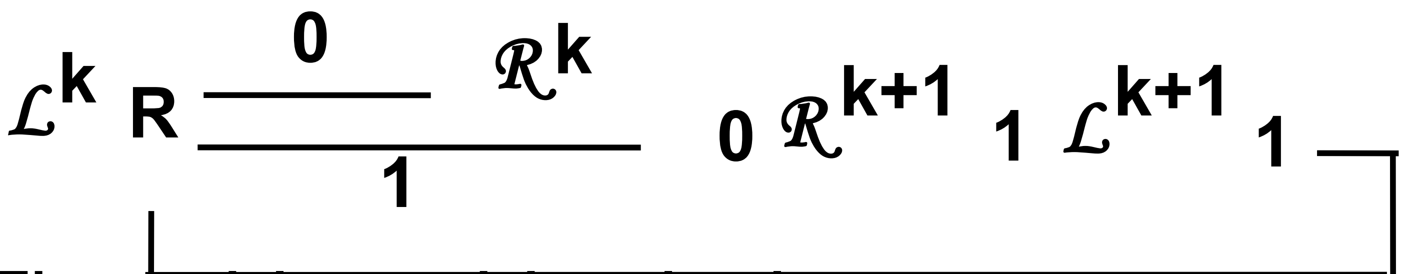
and ends with

$$\ldots 0 1^{x1} 0 1^{x2} 0 \ldots 0 1^{xn} 0 1^{y} \underline{0} \ldots$$

where y = F(x1, … , xn).

Addition by TM

Need the copy family of useful submachines, where C_k copies k-th preceding value.



The add machine is then

$$C_2\ C_2\ \mathcal{L}\ 1\ \mathcal{R}\ L\ 0$$

Turing Machine Variations

- Two tracks
- N tracks
- Non-deterministic *********
- Two-dimensional
- K dimensional
- Two stack machines
- Two counter machines

Register Machines

2nd Model

Feels Like Assembly Language

Register Machine Concepts

- A register machine consists of a finite length program, each of whose instructions is chosen from a small repertoire of simple commands.
- The instructions are labeled from **1** to **m**, where there are m instructions. Termination occurs as a result of an attempt to execute the **m+1**-st instruction.
- The storage medium of a register machine is a finite set of registers, each capable of storing an arbitrary natural number.
- Any given register machine has a finite, predetermined number of registers, independent of its input.

Computing by Register Machines

- A register machine partially computing some **n-**ary function **F** typically starts with its argument values in registers **1** to **n** and ends with the result in the **0-th** register.

- We extend this slightly to allow the computation to start with values in its **k+1**-st through **k+n**-th register, with the result appearing in the **k**-th register, for any **k**, such that there are at least k**+n+1** registers.

Register Instructions

- Each instruction of a register machine is of one of two forms:

  INC_r[i] –

  increment **r** and jump to **i**.

  DEC_r[p, z] –

  if register **r > 0**, decrement **r** and jump to **p**

  else jump to **z**
- Note, we do not use subscripts if obvious.

Addition by RM

Addition ($r0 \leftarrow r1 + r2$)

1. DEC0[1,2] : Zero result (r0) and work (r3) registers
2. DEC3[2,3]
3. DEC1[4,6] : Add r1 to r0, saving original r1 in r3
4. INC0[5]
5. INC3[3]
6. DEC3[7,8] : Restore r1
7. INC1[6]
8. DEC2[9,11] : Add r2 to r0, saving original r2 in r3
9. INC0[10]
10. INC3[8]
11. DEC3[12,13] : Restore r2
12. INC2[11]
13. : Halt by branching here

In many cases we just assume registers, other those with input, are zero at start. That would remove need instructions 1 and 2.

Limited Subtraction by RM

Subtraction ($r0 \leftarrow r1 - r2$, if $r1 \geq r2$; 0, otherwise)

1. **DEC0[1,2]** : **Zero result (r0) and work (r3) registers**
2. **DEC3[2,3]**
3. **DEC1[4,6]** : **Add r1 to r0, saving original r1 in r3**
4. **INC0[5]**
5. **INC3[3]**
6. **DEC3[7,8]** : **Restore r1**
7. **INC1[6]**
8. **DEC2[9,11]** : **Subtract r2 from r0, saving original r2 in r3**
9. **DEC0[10,10]** : **Note that decrementing 0 does nothing**
10. **INC3[8]**
11. **DEC3[12,13]** : **Restore r2**
12. **INC2[11]**
13. : **Halt by branching here**

Factor Replacement Systems

3rd Model

Deceptively Simple

Factor Replacement Concepts

- A factor replacement system (FRS) consists of a finite (ordered) sequence of fractions, and some starting natural number **x**.
- A fraction **a/b** is applicable to some natural number **x**, just in case **x** is divisible by **b**. We always chose the first applicable fraction (**a/b**), multiplying it times **x** to produce a new natural number **x*a/b**. The process is then applied to this new number.
- Termination occurs when no fraction is applicable.
- A factor replacement system partially computing **n**-ary function **F** typically starts with its argument encoded as powers of the first **n** odd primes. Thus, arguments **x1**,**x2**,…,**xn** are encoded as $\mathbf{3^{x1}5^{x2}\ldots p_n^{xn}}$. The result then appears as the power of the prime **2**.

Addition by FRS

Addition is $\mathbf{3^{x1}5^{x2}}$ becomes $\mathbf{2^{x1+x2}}$

or, in more details, $\mathbf{2^{0}3^{x1}5^{x2}}$ becomes $\mathbf{2^{x1+x2}\,3^{0}5^{0}}$

2 / 3

2 / 5

Note that these systems are sometimes presented as rewriting rules of the form

$\mathbf{bx} \rightarrow \mathbf{ax}$

meaning that a number that has can be factored as **bx** can have the factor **b** replaced by an **a**.

The previous rules would then be written

$\mathbf{3x} \rightarrow \mathbf{2x}$

$\mathbf{5x} \rightarrow \mathbf{2x}$

Limited Subtraction by FRS

Subtraction is $\mathbf{3^{x1}5^{x2}}$ becomes $\mathbf{2^{max(0,x1-x2)}}$

$$\mathbf{3 \cdot 5x \rightarrow x}$$
$$\mathbf{3x \rightarrow 2x}$$
$$\mathbf{5x \rightarrow x}$$

Ordering of Rules

- The ordering of rules are immaterial for the addition example but are critical to the workings of limited subtraction.
- In fact, if we ignore the order and just allow any applicable rule to be used, we get a form of non-determinism that makes these systems equivalent to Petri nets.
- The ordered kind are deterministic and are equivalent to a Petri net in which the transitions are prioritized.

Why Deterministic?

To see why determinism makes a difference, consider

$$3 \cdot 5x \rightarrow x$$
$$3x \rightarrow 2x$$
$$5x \rightarrow x$$

Starting with $\mathbf{135 = 3^3 5^1}$, deterministically we get

$$135 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2$$

Non-deterministically we get a larger, less selective set.

$$135 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2$$
$$135 \Rightarrow 90 \Rightarrow 60 \Rightarrow 40 \Rightarrow 8 = 2^3$$
$$135 \Rightarrow 45 \Rightarrow 3 \Rightarrow 2 = 2^1$$
$$135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 1 = 2^0$$
$$135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 5 \Rightarrow 1 = 2^0$$
$$135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 3 \Rightarrow 2 = 2^1$$
$$135 \Rightarrow 45 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2$$
$$135 \Rightarrow 90 \Rightarrow 60 \Rightarrow 40 \Rightarrow 8 = 2^3$$

…

This computes $\mathbf{2^z}$ where $\mathbf{0 \le z \le x_1}$. Think about it.

More on Determinism

In general, we might get an infinite set using non-determinism, whereas determinism might produce a finite set. To see this consider a system

$$\mathbf{2x} \rightarrow \mathbf{x}$$

$$\mathbf{2x} \rightarrow \mathbf{4x}$$

starting with the number **2**.

Sample RM and FRS

Present a Register Machine that computes IsOdd. Assume R1=x at starts; at termination, set R0=1 if x is odd; 0 otherwise. We assume R0=0 at start. We also are not concerned about destroying input.

1. DEC1[2, 4]
2. DEC1[1, 3]
3. INC0[4]
4.

Present a Factor Replacement System that computes IsOdd. Assume starting number is 3^x; at termination, result is 2=2^1 if x is odd; 1= 2^0 otherwise.

$3*3\ x \rightarrow x$

$3\ x \rightarrow 2\ x$

Sample FRS

Present a Factor Replacement System that computes IsPowerOf2. Assume starting number is 3^x 5; at termination, result is $2=2^1$ if x is a power of 2; $1= 2^0$ otherwise

$3^2*5\ x \rightarrow 5*7\ x$

$3*5*7\ x \rightarrow x$

$3*5\ x \rightarrow 2\ x$

$5*7\ x \rightarrow 7*11\ x$

$7*11\ x \rightarrow 3*11\ x$

$11\ x \rightarrow 5\ x$

$5\ x \rightarrow x$

$7\ x \rightarrow x$

Systems Related to FRS

- Petri Nets:
  - Unordered
  - Ordered
  - Negated Arcs
- Vector Addition Systems:
  - Unordered
  - Ordered
- Factors with Residues:
  - $a\,x + c \;\rightarrow\; b\,x + d$
- Finitely Presented Abelian Semi-Groups

Petri Net Operation

- Finite number of places, each of which can hold zero of more markers.
- Finite number of transitions, each of which has a finite number of input and output arcs, starting and ending, respectively, at places.
- A transition is enabled if all the nodes on its input arcs have at least as many markers as arcs leading from them to this transition.
- Progress is made whenever at least one transition is enabled. Among all enabled, one is chosen randomly to fire.
- Firing a transition removes one marker per arc from the incoming nodes and adds one marker per arc to the outgoing nodes.

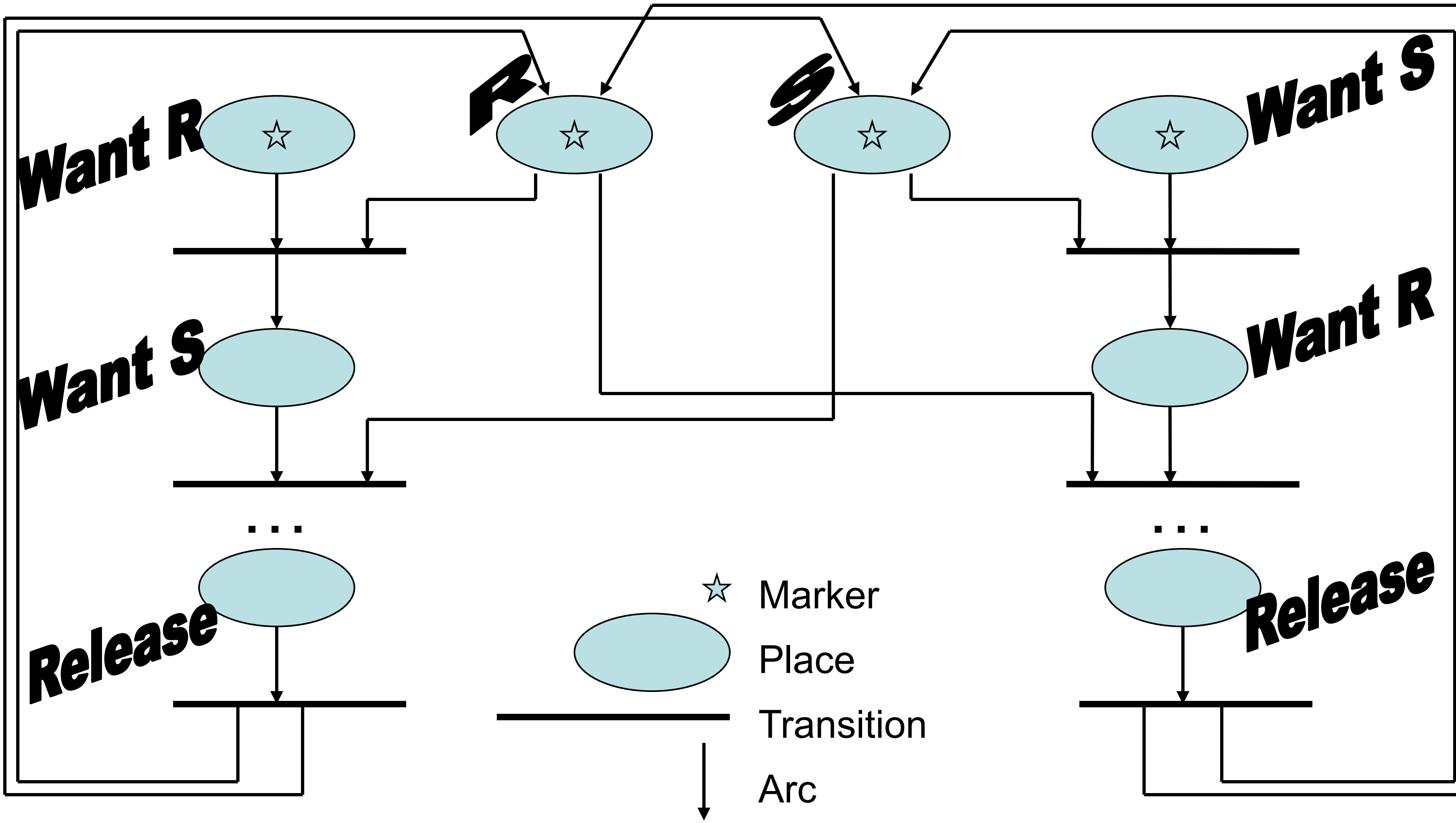
Petri Net Computation

- A Petri Net starts with some finite number of markers distributed throughout its **n** nodes.
- The state of the net is a vector of n natural numbers, with the **i**-th component's number indicating the contents of the **i**-th node. E.g., **<0,1,4,0,6>** could be the state of a Petri Net with **5** places, the 2nd, 3rd and 5th, having **1**, **4**, and **6** markers, resp., and the 1st and 4th being empty.
- Computation progresses by selecting and firing enabled transitions. Non-determinism is typical as many transitions can be simultaneously enabled.
- Petri nets are often used to model coordination algorithms, especially for computer networks.

Variants of Petri Nets

- A Petri Net is not computationally complete. In fact, its halting and word problems are decidable. However, its containment problem (are the markings of one net contained in those of another?) is not decidable.
- A Petri net with prioritized transitions, such that the highest priority transitions is fired when multiple are enabled is equivalent to an FRS. (Think about it).
- A Petri Net with negated input arcs is one where any arc with a slash through it contributes to enabling its associated transition only if the node is empty. These are computationally complete. They can simulate register machines. (Think about this also).

Petri Net Example



Vector Addition

- Start with a finite set of vectors in integer n-space.
- Start with a single point with non-negative integral coefficients.
- Can apply a vector only if the resultant point has non-negative coefficients.
- Choose randomly among acceptable vectors.
- This generates the set of reachable points.
- Vector addition systems are equivalent to Petri Nets.
- If order vectors, these are equivalent to FRS.

Vectors as Resource Models

- Each component of a point in **n**-space represents the quantity of a particular resource.
- The vectors represent processes that consume and produce resources.
- The issues are safety (do we avoid bad states) and liveness (do we attain a desired state).
- Issues are deadlock, starvation, etc.

Factors with Residues

- Rules are of form
  - $\mathbf{a_i\ x + c_i \rightarrow b_i\ x + d_i}$
  - There are **n** such rules
  - Can apply if number is such that you get a residue (remainder) $\mathbf{c_i}$ when you divide by $\mathbf{a_i}$
  - Take quotient **x** and produce a new number $\mathbf{b_i\ x + d_i}$
  - Can apply any applicable one (no order)
- These systems are equivalent to Register Machines.

Abelian Semi-Group

S = (G, •) is a semi-group if

G is a set, • is a binary operator, and

1. Closure: If $\mathbf{x,y} \in \mathbf{G}$ then $\mathbf{x \bullet y} \in \mathbf{G}$
2. Associativity: $\mathbf{x \bullet (y \bullet z) = (x \bullet y) \bullet z}$

S is a monoid if

3. Identity: $\exists \mathbf{e} \in \mathbf{G}\ \forall \mathbf{x} \in \mathbf{G}\ [\mathbf{e \bullet x = x \bullet e = x}]$

S is a group if

4. Inverse: $\forall \mathbf{x} \in \mathbf{G}\ \exists \mathbf{x^{-1}} \in \mathbf{G}\ [\mathbf{x^{-1} \bullet x = x \bullet x^{-1} = e}]$

S is Abelian if • is commutative

Finitely Presented

- **S = (G, •)**, a semi-group (monoid, group), is finitely presented if there is a finite set of symbols, Σ, called the alphabet or generators, and a finite set of equalities ($\alpha_i = \beta_i$), the reflexive transitive closure of which determines equivalence classes over **G**.
- Note, the set **G** is the closure of the generators under the semi-group's operator •.
- The problem of determining membership in equivalence classes for finitely presented Abelian semi-groups is equivalent to that of determining mutual derivability in an unordered FRS or Vector Addition System with inverses for each rule.

Recursive Functions

Primitive and μ-Recursive

Primitive Recursive

An Incomplete Model

Basis of PRFs

- The primitive recursive functions are defined by starting with some base set of functions and then expanding this set via rules that create new primitive recursive functions from old ones.
- The **base functions** are:

$\mathbf{C_a(x_1,\ldots,x_n) = a}$	: constant functions
$\mathbf{I^n_i(x_1,\ldots,x_n) = x_i}$	: identity functions
	: aka projection
$\mathbf{S(x) = x+1}$	: an increment function

Building New Functions

- **Composition:**

  If **G**, $\mathbf{H_1}$, … , $\mathbf{H_k}$ are already known to be primitive recursive, then so is **F**, where

  $$F(x_1,\ldots,x_n) = G(H_1(x_1,\ldots,x_n), \ldots , H_k(x_1,\ldots,x_n))$$

- **Iteration (aka primitive recursion):**

  If **G**, **H** are already known to be primitive recursive, then so is **F**, where

  $$F(0, x_1,\ldots,x_n) = G(x_1,\ldots,x_n)$$
  $$F(y+1, x_1,\ldots,x_n) = H(y, x_1,\ldots,x_n, F(y, x_1,\ldots,x_n))$$

  We also allow definitions like the above, except iterating on y as the last, rather than first argument.

Addition & Multiplication

Example: Addition

$+(0,y) = I_1^1(y)$

$+(x+1,y) = H(x,y,+(x,y))$

where $H(a,b,c) = S(I_3^3(a,b,c))$

Example: Multiplication

$*(0,y) = C_0(y)$

$*(x+1,y) = H(x,y,*(x,y))$

where $H(a,b,c) = +(I_2^3(a,b,c), I_3^3(a,b,c))$

$= b+c = y + *(x,y) = (x+1)*y$

Basic Arithmetic

x + 1:

$x + 1 = S(x)$

x – 1:

$0 - 1 = 0$

$(x+1) - 1 = x$

x + y:

$x + 0 = x$

$x + (y+1) = (x+y) + 1$

x – y: // limited subtraction

$x - 0 = x$

$x - (y+1) = (x-y) - 1$

2nd Grade Arithmetic

$x * y$:

$x * 0 = 0$

$x * (y+1) = x*y + x$

$x!$:

$0! = 1$

$(x+1)! = (x+1) * x!$

Basic Relations

x == 0:

0 == 0 = 1

(y+1) == 0 = 0

x == y:

x==y = ((x – y) + (y – x)) == 0

x ≤y :

x≤y = (x – y) == 0

x ≥ y:

x≥y = y≤x

x > y :

x>y = ~(x≤y) /* See ~ on next page */

x < y :

x<y = ~(x≥y)

Basic Boolean Operations

~x:

~x = 1 – x or (x==0)

signum(x): 1 if **x>0**; **0** if **x==0**

~(x==0)

x && y:

x&&y = signum(x*y)

x || y:

x||y = ~((x==0) && (y==0))

Definition by Cases

One case

$$f(x) = \begin{array}{ll} g(x) & \text{if } P(x) \\ h(x) & \text{otherwise} \end{array}$$

$$f(x) = P(x) * g(x) + (1-P(x)) * h(x)$$

Can use induction to prove this is true for all **k>0**, where

$$f(x) = \begin{array}{ll} g_1(x) & \text{if } P_1(x) \\ g_2(x) & \text{if } P_2(x) \ \&\& \ {\sim}P_1(x) \\ \dots & \\ g_k(x) & \text{if } P_k(x) \ \&\& \ {\sim}(P_1(x) \ || \ \dots \ || \ {\sim}P_{k-1}(x)) \\ h(x) & \text{otherwise} \end{array}$$

Bounded Minimization 1

$\mathbf{f(x) = \mu\, z\, (z \le x)\, [\, P(z)\,]}$ if $\exists$ such a **z**,
$\quad\quad$ $\mathbf{= x+1}$, otherwise
where **P(z)** is primitive recursive.

Can show **f** is primitive recursive by

f(0)	**=**	**1-P(0)**	
f(x+1)	**=**	**f(x)**	if **f(x) ≤ x**
	=	**x+2-P(x+1)**	otherwise

Bounded Minimization 2

f(x) = μ **z (z < x) [P(z)]** if $\exists$ such a **z,**
= x, otherwise
where **P(z)** is primitive recursive.

Can show **f** is primitive recursive by
f(0) = 0
f(x+1) = μ **z (z ≤ x) [P(z)]**

Intermediate Arithmetic

x // y:

$x//0 = 0$: silly, but want a value

$x//(y+1) = \mu z\ (z<x)\ [\ (z+1)*(y+1) > x\ ]$

x | y: x is a divisor of **y**

$x|y = ((y//x) * x) == y$

Primality

firstFactor(x): first non-zero, non-one factor of **x**.

firstfactor(x) = $\mu\, z\ (2 \le z \le x)\ [\, z|x\,]$,

0 if none

isPrime(x):

isPrime(x) = firstFactor(x) == x && (x>1)

prime(i) = i-th prime:

prime(0) = 2

prime(x+1) = $\mu\, z(\text{prime}(x) < z \le \text{prime}(x)!+1)[\text{isPrime}(z)]$

We will abbreviate this as p_i for **prime(i)**

Exponents

x^y:

x^0 = 1

x^(y+1) = x * x^y

exp(x,i): the exponent of $\mathbf{p_i}$ in number **x.**

exp(x,i) = μ z (z<x) [~(p_i^(z+1) | x)]

Pairing Functions

- $\mathbf{pair(x,y) = \langle x,y \rangle = 2^x\ (2y + 1) - 1}$
- with inverses

  $\mathbf{\langle z \rangle_1 = exp(z+1,0)}$

  $\mathbf{\langle z \rangle_2 = (((\ z + 1\ )\ //\ 2^{\langle z \rangle_1}\ ) - 1\ )\ //\ 2}$
- These are very useful and can be extended to encode **n**-tuples

  $\mathbf{\langle x,y,z \rangle = \langle x, \langle y,z \rangle \rangle}$ (note: stack analogy)

Pairing Function is 1-1 Onto

Prove that the pairing function $\langle x,y\rangle = 2^x (2y + 1) - 1$ is 1-1 onto the natural numbers.

Approach 1:

We will look at two cases, where we use the following modification of the pairing function, $\langle x,y\rangle+1$, which implies the problem of mapping the pairing function to Z^+.

Case 1 (x=0)

Case 1:

For x = 0, <0,y>+1 = $2^0(2y+1)$ = 2y+1. But every odd number is by definition one of the form 2y+1, where y≥0; moreover, a particular value of y is uniquely associated with each such odd number and no odd number is produced when x=0. Thus, <0,y>+1 is 1-1 onto the odd natural numbers.

Case 2 (x > 0)

Case 2:

For $x > 0$, $<x,y>+1 = 2^x(2y+1)$, where $2y+1$ ranges over all odd number and is uniquely associated with one based on the value of y (we saw that in case 1). 2^x must be even, since it has a factor of 2 and hence $2^x(2y+1)$ is also even. Moreover, from elementary number theory, we know that every even number except zero is of the form 2^xz, where $x>0$, z is an odd number and this pair x,y is unique. Thus, $<x,y>+1$ is 1-1 onto the even natural numbers, when $x>0$.

The above shows that $<x,y>+1$ is 1-1 onto Z^+, but then $<x,y>$ is 1-1 onto $\aleph$, as was desired.

Pairing Function is 1-1 Onto

Approach 2:

Another approach to show a function f over S is 1-1 onto T is to show that
$f^{-1}(f(x)) = x$, for arbitrary $x \in S$ and that
$f(f^{-1}(z)) = z$, for arbitrary $z \in T$.

Thus, we need to show that
$(\langle x,y \rangle_1, \langle x,y \rangle_2) = (x,y)$ for arbitrary $(x,y) \in \aleph \times \aleph$ and
$\langle \langle z \rangle_1, \langle z \rangle_2 \rangle = z$ for arbitrary $z \in \aleph$.

Alternate Proof

Let x,y be arbitrary natural number, then $\langle x,y\rangle = 2^x(2y+1)-1$.

Moreover, $\langle 2^x(2y+1)-1\rangle_1$ = Factor($2^x(2y+1)$,0) = x, since 2y+1 must be odd, and

$\langle 2^x(2y+1)-1\rangle_2$ = (($2^x(2y+1)$/2^Factor($2^x(2y+1)$,0))-1)/2 = 2y/2 = y.

Thus, $(\langle x,y\rangle_1,\langle x,y\rangle_2) = (x,y)$, as was desired.

Let z be an arbitrary natural number, then the inverse of the pairing is $(\langle z\rangle_1,\langle z\rangle_2)$

Moreover, $\langle\langle z\rangle_1,\langle z\rangle_2\rangle$ = 2^$\langle z\rangle_1$ *(2$\langle z\rangle_2$+1)-1

= 2^Factor(z+1,0)*(2*((z+1)/ 2^Factor(z+1,0))/2-1+1)-1

= 2^Factor(z+1,0)*((z+1)/ 2^Factor(z+1,0))-1

= (z+1) – 1

= z, as was desired.

Application of Pairing

Show that prfs are closed under Fibonacci induction. Fibonacci induction means that each induction step after calculating the base is computed using the previous two values, where the previous values for f(1) are f(0) and 0; and for x>1, f(x) is based on f(x-1) and f(x-2).

The formal hypothesis is:
Assume g and h are already known to be prf, then so is f, where
f(0,x) = g(x);

f(1,x) = h(f(0,x), 0); and

f(y+2,x) = h(f(y+1,x), f(y,x))

Proof is by construction

Fibonacci Recursion

Let K be the following primitive recursive function, defined by induction on the primitive recursive functions, g, h, and the pairing function.

$K(0,x) = B(x)$

$B(x) = < g(x), C_0(x) >$ // this is just $<g(x), 0>$

$K(y+1, x) = J(y, x, K(y,x))$

$J(y,x,z) = < h(<z>_1, <z>_2), <z>_1 >$

// this is $< f(y+1,x), f(y,x)>$, even though f is not yet shown to be prf!!

This shows K is prf.

f is then defined from K as follows:

$f(y,x) = <K(y,x)>_1$ // extract first value from pair encoded in K(y,x)

This shows it is also a prf, as was desired.

μ Recursive

4th Model

A Simple Extension to Primitive Recursive

μ Recursive Concepts

- All primitive recursive functions are algorithms since the only iterator is bounded. That's a clear limitation.
- There are algorithms like Ackerman's function that cannot be represented by the class of primitive recursive functions.
- The class of recursive functions adds one more iterator, the minimization operator (μ), read "the least value such that."

Ackermann's Function

- **$A(1, j)=2j$ for $j \geq 1$**
- **$A(i, 1)=A(i-1, 2)$** for **$i \geq 2$**
- **$A(i, j)=A(i-1, A(i, j-1))$** for **$i, j \geq 2$**
- Wilhelm Ackermann observed in 1928 that this is not a primitive recursive function.
- Ackermann's function grows too fast to have a for-loop implementation.
- The inverse of Ackermann's function is important to analyze Union/Find algorithm. Note: A(4,4) is a supper exponential number involving six levels of exponentiation. $\alpha(n) = A^{-1}(n, n)$ grows so slowly that it is less than 5 for any value of n that can be written.

Union/Find

- Start with a collection **S** of unrelated elements – singleton equivalence classes
- **Union(x,y)**, **x** and **y** are in **S**, merges the class containing **x** (**[x]**) with that containing **y** (**[y]**)
- **Find(x)** returns the canonical element of **[x]**
- Can see if **x≡y**, by seeing if **Find(x)==Find(y)**
- How do we represent the classes?

The μ Operator

- Minimization:

  If **G** is already known to be recursive, then so is **F**, where

  $$\mathbf{F(x1,\ldots,xn) = \mu y\ (G(y,x1,\ldots,xn) == 1)}$$

- We also allow other predicates besides testing for one. In fact any predicate that is recursive can be used as the stopping condition.

Equivalence of Models

Equivalency of computation by
Turing machines,
register machines,
factor replacement systems,
recursive functions

Proving Equivalence

- Constructions do not, by themselves, prove equivalence.
- To do so, we need to develop a notion of an “instantaneous description” (id) of each model of computation (well, almost as recursive functions are a bit different).
- We then show a mapping of id’s between the models.

Instantaneous Descriptions

- An instantaneous description (id) is a finite description of a state achievable by a computational machine, ***M***.
- Each machine starts in some initial id, id_0.
- The semantics of the instructions of *M* define a relation $\Rightarrow_M$ such that, $\mathbf{id_i} \Rightarrow_M \mathbf{id_{i+1}}$, $\mathbf{i \geq 0}$, if the execution of a single instruction of ***M*** would alter ***M***'s state from $\mathbf{id_i}$ to $\mathbf{id_{i+1}}$ or if ***M*** halts in state $\mathbf{id_i}$ and $\mathbf{id_{i+1}=id_i}$.
- $\Rightarrow^{+}_{M}$ is the transitive closure of $\Rightarrow_M$
- $\Rightarrow^{*}_{M}$ is the reflexive transitive closure of $\Rightarrow_M$

id Definitions

- For a register machine, **M**, an id is an **s+1** tuple of the form $\mathbf{(i, r_1,\ldots,r_s)_M}$ specifying the number of the next instruction to be executed and the values of all registers prior to its execution.
- For a factor replacement system, an id is just a natural number.
- For a Turing machine, **M**, an id is some finite representation of the tape, the position of the read/write head and the current state. This is usually represented as a string $\alpha\mathbf{qx}\beta$, where α (β) is the shortest string representing all non-blank squares to the left (right) of the scanned square, **x** is the symbol at the scanned square and **q** is the current state.
- Recursive functions do not have id's, so we will handle their simulation by an inductive argument, using the primitive functions are the basis and composition, induction and minimization in the inductive step.

Equivalence Steps

- Assume we have a machine ***M*** in one model of computation and a mapping of ***M*** into a machine ***M'*** in a second model.
- Assume the initial configuration of ***M*** is $\mathbf{id_0}$ and that of ***M'*** is $\mathbf{id'_0}$
- Define a mapping, **h**, from id's of ***M*** into those of ***M'***, such that, $\mathbf{R}_M$ **=** { **h(d) | d** is an instance of an id of ***M*** }, and
  - $\mathbf{id'_0 \Rightarrow^*_{M'} h(id_0)}$, and $\mathbf{h(id_0)}$ is the only member of $\mathbf{R}_M$ in the configurations encountered in this derivation.
  - $\mathbf{h(id_i) \Rightarrow^+_{M'} h(id_{i+1})}$, $\mathbf{i \geq 0}$, and $\mathbf{h(id_{i+1})}$ is the only member of $\mathbf{R}_M$ in this derivation.
- The above, in effect, provides an inductive proof that
  - $\mathbf{id_0 \Rightarrow^*_M id}$ implies $\mathbf{id'_0 \Rightarrow^*_{M'} h(id)}$, and
  - If $\mathbf{id'_0 \Rightarrow^*_{M'} id'}$ then either $\mathbf{id_0 \Rightarrow^*_M id}$, where $\mathbf{id' = h(id)}$, or $\mathbf{id' \notin R_M}$

All Models are Equivalent

Equivalency of computation by
Turing machines, register machines,
factor replacement systems,
recursive functions

Our Plan of Attack

- We will now show
  TURING ≤ REGISTER ≤ FACTOR ≤ RECURSIVE ≤ TURING
  where by **A ≤ B**, we mean that every instance of **A** can be replaced by an equivalent instance of **B**.
- The transitive closure will then get us the desired result.

TURING ≤ REGISTER

Encoding a TM's State

- Assume that we have an **n** state Turing machine. Let the states be numbered **0,…, n-1**.
- Assume our machine is in state **7**, with its tape containing
  … 0 0 1 0 1 0 0 1 1 q7 $\underline{0}$ 0 0 …
- The underscore indicates the square being read. We denote this by the finite id
  1 0 1 0 0 1 1 q7 $\underline{0}$
- In this notation, we always write down the scanned square, even if it and all symbols to its right are blank.

More on Encoding of TM

- An id can be represented by a triple of natural numbers, **(R,L,i)**, where **R** is the number denoted by the reversal of the binary sequence to the right of the **qi**, **L** is the number denoted by the binary sequence to the left, and **i** is the state index.
- So,
  … 0 0 1 0 1 0 0 1 1 q7 <u>0</u> 0 0 …
  is just (**0, 83, 7**).
  … 0 0 1 0 q5 <u>1</u> 0 1 1 0 0 …
  is represented as (**13, 2, 5**).
- We can store the **R** part in register **1**, the **L** part in register **2**, and the state index in register **3**.

Simulation by RM

1.	DEC3[2,q0]	: Go to simulate actions in state 0
2.	DEC3[3,q1]	: Go to simulate actions in state 1
…		
n.	DEC3[ERR,qn-1]	: Go to simulate actions in state n-1
…		
qj.	IF_r1_ODD[qj+2]	: Jump if scanning a 1
qj+1.	JUMP[set_k]	: If (qj 0 0 qk) is rule in TM
qj+1.	INC1[set_k]	: If (qj 0 1 qk) is rule in TM
qj+1.	DIV_r1_BY_2 MUL_r2__BY_2 JUMP[set_k]	: If (qj 0 R qk) is rule in TM
qj+1.	MUL_r1_BY_2 IF_r2_ODD then INC1 DIV_r2__BY_2[set_k]	: If (qj 0 L qk) is rule in TM
…		
set_n-1.	INC3[set_n-2]	: Set r3 to index n-1 for simulating state n-1
set_n-2.	INC3[set_n-3]	: Set r3 to index n-2 for simulating state n-2
…		
set_0.	JUMP[1]	: Set r3 to index 0 for simulating state 0

Fixups

- Need epilog so action for missing quad (halting) jumps beyond end of simulation to clean things up, placing result in **r0**.
- Can also have a prolog that starts with arguments in registers **r1** to **rn** and stores values in **r1**, **r2** and **r3** to represent Turing machines starting configuration.

Prolog

Example assuming **n** arguments (fix as needed)

1.	**MUL_rn+1_BY_2[2]**	**: Set rn+1 = 11…10$_2$, where, #1's = r1**
2.	**DEC1[3,4]**	**: r1 will be set to 0**
3.	**INCn+1[1]**	**:**
4.	**MUL_rn+1_BY_2[5]**	**: Set rn+1 = 11…1011…10$_2$, where, #1's = r1, then r2**
5.	**DEC2[6,7]**	**: r2 will be set to 0**
6.	**INCn+1[4]**	**:**
…		
3n-2.	**DECn[3n-1,3n+1]**	**: Set rn+1 = 11…1011…1011…1$_2$, where, #1's = r1, r2,…**
3n-1.	**MUL_rn+1_BY_2[3n]**	**: rn will be set to 0**
3n.	**INCn+1[3n-2]**	**:**
3n+1	**DECn+1[3n+2,3n+3]**	**: Copy rn+1 to r2, rn+1 is set to 0**
3n+2.	**INC2[3n+1]**	**:**
3n+3.		**: r2 = left tape, r1 = 0 (right), r3 = 0 (initial state)**

Epilog

1.	DEC3[1,2]	: Set r3 to 0 (just cleaning up)
2.	IF_r1_ODD[3,5]	: Are we done with answer?
3.	INC0[4]	: putting answer in r0
4.	DIV_r1_BY_2[2]	: strip a 1 from r1
5.		: Answer is now in r0

REGISTER $\leq$ FACTOR

Encoding a RM's State

- This is a really easy one based on the fact that every member of $\mathbf{Z^+}$ (the positive integers) has a unique prime factorization. Thus all such numbers can be uniquely written in the form

  $$p_{i_1}^{k_1} p_{i_2}^{k_2} \cdots p_{i_j}^{k_j}$$

  where the $\mathbf{p_i}$'s are distinct primes and the $\mathbf{k_i}$'s are non-zero values, except that the number **1** would be represented by $\mathbf{2^0}$.
- Let R be an arbitrary **n+1**-register machine, having m instructions.

  Encode the contents of registers **r0,…,rn** by the powers of $\mathbf{p_0,\ldots p_n}$.

  Encode rule number's **1,…,m** by primes $\mathbf{p_{n+1} ,\ldots, p_{n+m}}$

  Use $\mathbf{p_{n+m+1}}$ as prime factor that indicates simulation is done.
- This is, in essence, a Gödel number of the RM's state.

Simulation by FRS

- Now, the **j**-th instruction (**1≤j≤m**) of **R** has associated factor replacement rules as follows:

j. INCr[i]

$$\mathbf{p_{n+j}x \quad \rightarrow \quad p_{n+i}p_rx}$$

j. DECr[s, f]

$$\mathbf{p_{n+j}p_rx \quad \rightarrow \quad p_{n+s}x}$$

$$\mathbf{p_{n+j}x \quad \rightarrow \quad p_{n+f}x}$$

- We also add the halting rule associated with **m+1** of

$$\mathbf{p_{n+m+1}x \quad \rightarrow \quad x}$$

Importance of Order

- The relative order of the two rules to simulate a **DEC** are critical.
- To test if register **r** has a zero in it, we, in effect, make sure that we cannot execute the rule that is enabled when the **r**-th prime is a factor.
- If the rules were placed in the wrong order, or if they weren't prioritized, we would be non-deterministic.

Sample RM and FRS (repeat)

Present a Register Machine that computes IsOdd. Assume R1=x at starts; at termination, set R0=1 if x is odd; 0 otherwise. We assume R0=0 at start. We also are not concerned about destroying input.

1. DEC1[2, 4]
2. DEC1[1, 3]
3. INC0[4]
4.

Present a Factor Replacement System that computes IsOdd. Assume starting number is 3^x; at termination, result is 2=2^1 if x is odd; 1= 2^0 otherwise.

$3*3\ x \rightarrow x$

$3\ x \rightarrow 2\ x$

Example of Order

Consider the simple machine to compute **r0:=r1 – r2** (limited)

1. **DEC2[2,3]**
2. **DEC1[1,1]**
3. **DEC1[4,5]**
4. **INC0[3]**
5.

Subtraction Encoding

Start with $\mathbf{3^x 5^y 7}$

7 • 5 x	$\rightarrow$	**11 x**
7 x	$\rightarrow$	**13 x**
11 • 3 x	$\rightarrow$	**7 x**
11 x	$\rightarrow$	**7 x**
13 • 3 x	$\rightarrow$	**17 x**
13 x	$\rightarrow$	**19 x**
17 x	$\rightarrow$	**13 • 2 x**
19 x	$\rightarrow$	**x**

Analysis of Problem

- If we don't obey the ordering here, we could take an input like $\mathbf{3^5 5^2 7}$ and immediately apply the second rule (the one that mimics a failed decrement).
- We then have $\mathbf{3^5 5^2 13}$, signifying that we will mimic instruction number **3**, never having subtracted the **2** from **5**.
- Now, we mimic copying **r1** to **r0** and get $\mathbf{2^5 5^2 19}$.
- We then remove the **19** and have the wrong answer.

FACTOR $\leq$ RECURSIVE

Universal Machine

- In the process of doing this reduction, we will build a Universal Machine.
- This is a single recursive function with two arguments. The first specifies the factor system (encoded) and the second the argument to this factor system.
- The Universal Machine will then simulate the given machine on the selected input.

Encoding FRS

- Let **(n, ((a$_1$,b$_1$), (a$_2$,b$_2$), … ,(a$_n$,b$_n$))** be some factor replacement system, where **(a$_i$,b$_i$)** means that the **i**-th rule is

  a$_i$x $\rightarrow$ **b$_i$x**
- Encode this machine by the number **F**,

$$2^n 3^{a_1} 5^{b_1} 7^{a_2} 11^{b_2} \cdots p_{2n-1}^{a_n} p_{2n}^{b_n} p_{2n+1} p_{2n+2}$$

Simulation by Recursive # 1

- We can determine the rule of **F** that applies to **x** by

  $$\mathbf{RULE(F, x) = \mu\, z\, (1 \le z \le exp(F, 0)+1)\, [\, exp(F, 2^*z\text{-}1) \mid x\,]}$$
- Note: **exp(F,2*i-1) =** $\mathbf{a_i}$ where $\mathbf{a_i}$ is the exponent of the prime factor $\mathbf{p_{2i-1}}$ of **F**.

- If **x** is divisible by $\mathbf{a_i}$, and **i** is the least integer, **1≤i≤n**, for which this is true, then **RULE(F,x) = i**.

  If x is not divisible by any $\mathbf{a_i}$, **1≤i≤n**, then **x** is divisible by **1**, and **RULE(F,x)** returns **n+1**. That's why we added $\mathbf{p_{2n+1}\ p_{2n+2}}$.
- Given the function **RULE(F,x)**, we can determine **NEXT(F,x)**, the number that follows **x**, when using **F**, by

  $$\mathbf{NEXT(F, x) = (x\ //\ exp(F, 2^*RULE(F, x)\text{-}1)) * exp(F, 2^*RULE(F, x))}$$

Simulation by Recursive # 2

- The configurations listed by **F**, when started on **x**, are

CONFIG(F, x, 0) = x

CONFIG(F, x, y+1) = NEXT(F, CONFIG(F, x, y))

- The number of the configuration on which **F** halts is

HALT(F, x) = μ y [CONFIG(F, x, y) == CONFIG(F, x, y+1)]

This assumes we converge to a fixed point as our means of halting. Of course, no applicable rule meets this definition as the n+1-st rule divides and then multiplies the latest value by 1.

Simulation by Recursive # 3

- A Universal Machine that simulates an arbitrary Factor System, Turing Machine, Register Machine, Recursive Function can then be defined by

  Univ(F, x) = exp (CONFIG (F, x, HALT (F, x)), 0)

- This assumes that the answer will be returned as the exponent of the only even prime, **2**. We can fix **F** for any given Factor System that we wish to simulate. It is that ability that makes this function universal.

FRS Subtraction

- $2^0 3^a 5^b \Rightarrow 2^{a-b}$
  $3*5x \to x$ or $1/15$
  $5x \to x$ or $1/5$
  $3x \to 2x$ or $2/3$
- Encode $F = 2^3\ 3^{15}\ 5^1\ 7^5\ 11^1\ 13^3\ 17^2\ 19^1\ 23^1$
- Consider a=4, b=2
- $\text{RULE}(F, x) = \mu\, z\ (1 \le z \le 4)\ [\ \exp(F, 2*z-1) \mid x\ ]$
  $\text{RULE}\ (F, 3^4\ 5^2) = 1$, as 15 divides $3^4\ 5^2$
- $\text{NEXT}(F, x) = (x\ //\ \exp(F, 2*\text{RULE}(F, x)-1)) * \exp(F, 2*\text{RULE}(F, x))$
  $\text{NEXT}(F, 3^4\ 5^2) = (3^4\ 5^2\ //\ 15 * 1) = 3^3 5^1$
  $\text{NEXT}(F, 3^3\ 5^1) = (3^3\ 5^1\ //\ 15 * 1) = 3^2$
  $\text{NEXT}(F, 3^2) = (3^2\ //\ 3 * 2) = 2^1 3^1$
  $\text{NEXT}(F, 2^1 3^1) = (2^1 3^1\ //\ 3 * 2) = 2^2$
  $\text{NEXT}(F, 2^2) = (2^2\ //\ 1 * 1) = 2^2$

Rest of simulation

- CONFIG(F, x, 0) = x
  CONFIG(F, x, y+1) = NEXT(F, CONFIG(F, x, y))
- CONFIG(F,3^4 5^2,0) = 3^4 5^2
  CONFIG(F,3^4 5^2,1) = $3^3 5^1$
  CONFIG(F,3^4 5^2,2) = 3^2
  CONFIG(F,3^4 5^2,3) = $2^1 3^1$
  CONFIG(F,3^4 5^2,4) = 2^2
  CONFIG(F,3^4 5^2,5) = 2^2
- HALT(F, x)=μy[CONFIG(F,x,y)==CONFIG(F,x,y+1)] = 4
- Univ(F, x) = exp (CONFIG (F, x, HALT (F, x)), 0)
  = exp(2^2,0) = 2

Simplicity of Universal

- A side result is that every computable (recursive) function can be expressed in the form

$$\mathbf{F(x) = G(\mu\ y\ H(x, y))}$$

where **G** and **H** are primitive recursive.

RECURSIVE $\leq$ TURING

Standard Turing Computation

- Our notion of standard Turing computability of some **n**-ary function **F** assumes that the machine starts with a tape containing the **n** inputs, **x1, … , xn** in the form

  $\ldots 01^{x1}01^{x2}0\ldots 01^{xn}\underline{0}\ldots$

  and ends with

  $\ldots 01^{x1}01^{x2}0\ldots 01^{xn}01^{y}\underline{0}\ldots$

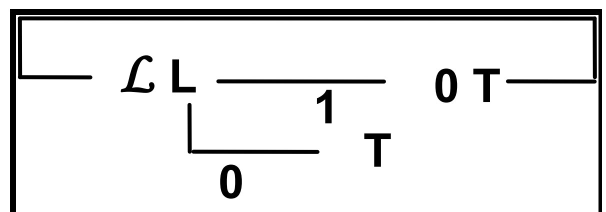
  where **y = F(x1, … , xn)**.

More Helpers

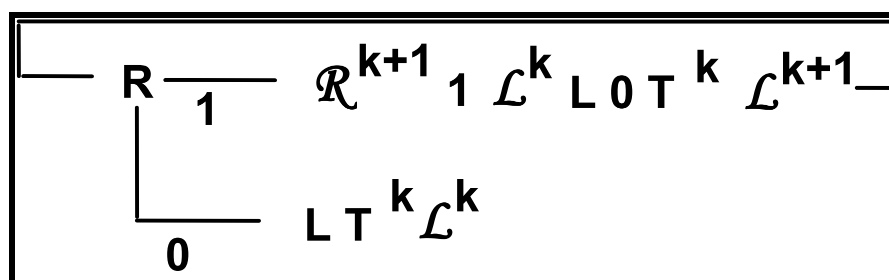
- To build our simulation we need to construct some useful submachines, in addition to the $\mathcal{R}$, $\mathcal{L}$, **R**, **L**, and $\mathbf{C_k}$ machines already defined.
- **T** -- translate moves a value left one tape square
  $\ldots\underline{?}01^x0\ldots \Rightarrow \ldots?1^x\underline{0}0\ldots$

$\boxed{\mathbf{R1\mathcal{R}L0}}$

- Shift -- shift a rightmost value left, destroying value to its left
  $\ldots01^{x1}01^{x2}\underline{0}\ldots \Rightarrow \ldots01^{x2}\underline{0}\ldots$



- $\mathbf{Rot_k}$ -- Rotate a **k** value sequence one slot to the left
  $\ldots\underline{0}1^{x1}01^{x2}0\ldots01^{xk}0\ldots$
  $\Rightarrow \ldots\underline{0}1^{x2}0\ldots01^{xk}01^{x1}0\ldots$



Basic Functions

All Basis Recursive Functions are Turing computable:

- $\mathbf{C_a^n(x_1,\ldots,x_n) = a}$

$\mathbf{(R1)^aR}$

- $\mathbf{I_i^n(x_1,\ldots,x_n) = x_i}$

$\mathbf{C_{n-i+1}}$

- $\mathbf{S(x) = x+1}$

$\mathbf{C_1 1R}$

Closure Under Composition

If **G, H_1, … , H_k** are already known to be Turing computable, then so is **F**, where

$$F(x_1,\ldots,x_n) = G(H1(x_1,\ldots,x_n), \ldots , Hk(x_1,\ldots,x_n))$$

To see this, we must first show that if **$E(x_1,\ldots,x_n)$** is Turing computable then so is

$$E\langle m\rangle(x_1,\ldots,x_n, y_1,\ldots,y_m) = E(x_1,\ldots,x_n)$$

This can be computed by the machine

$$\mathcal{L}^{n+m} (Rot_{n+m})^n \mathcal{R}^{n+m} E \ \ \mathcal{L}^{n+m+1} (Rot_{n+m})^m \mathcal{R}^{n+m+1}$$

Can now define **F** by

$$H_1\ H_2\langle 1\rangle\ H_3\langle 2\rangle \ldots H_k\langle k\text{-}1\rangle\ G\ Shift^k$$

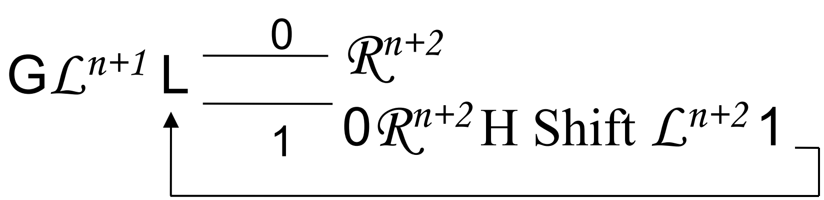
Closure Under Induction

To prove that Turing Machines are closed under induction (primitive recursion), we must simulate some arbitrary primitive recursive function $F(y,x_1,x_2, \ldots, x_n)$ on a Turing Machine, where

$F(0, x_1,x_2, \ldots, x_n) = G(x_1,x_2, \ldots, x_n)$

$F(y+1, x_1,x_2, \ldots, x_n) = H(y, x_1,x_2, \ldots, x_n, F(y,x_1,x_2, \ldots, x_n))$

Where, G and H are Standard Turing Computable. We define the function F for the Turing Machine as follows:



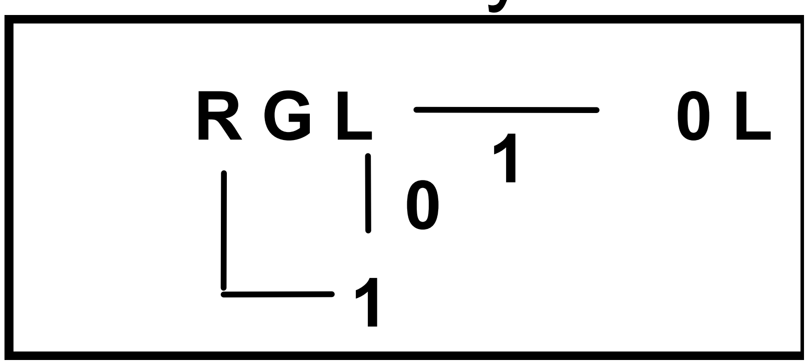
Since our Turing Machine simulator can produce the same value for any arbitrary PRF, F, we show that Turing Machines are closed under induction (primitive recursion).

Closure Under Minimization

If **G** is already known to be Turing computable, then so is **F**, where

$$\mathbf{F(x_1,\ldots,x_n) = \mu y\ (G(x_1,\ldots,x_n,\ y) == 1)}$$

This can be done by



Consequences of Equivalence

- Theorem: The computational power of Recursive Functions, Turing Machines, Register Machine, and Factor Replacement Systems are all equivalent.
- Theorem: Every Recursive Function (Turing Computable Function, etc.) can be performed with just one unbounded type of iteration.
- Theorem: Universal machines can be constructed for each of our formal models of computation.

Additional Notations

Includes comment on our notation versus that of others

Universal Machine

- Others consider functions of ***n*** arguments, whereas we had just one. However, our input to the FRS was actually an encoding of n arguments.
- The fact that we can focus on just a single number that is the encoding of ***n*** arguments is easy to justify based on the pairing function.
- Some presentations order arguments differently, starting with the ***n*** arguments and then the Gödel number of the function, but closure under argument permutation follows from closure under substitution.

Universal Machine Mapping

- $\varphi^{(n)}(f, x_1,\ldots,x_n) = \text{Univ}\ (f, \prod_{i=1}^{n} p_i^{x_i})$
- We will sometimes adopt the above and also its common shorthand

  $\varphi_f^{(n)}(x_1,\ldots,x_n) = \varphi^{(n)}(f, x_1,\ldots,x_n)$

  and the even shorter version

  $\varphi_f(x_1,\ldots,x_n) = \varphi^{(n)}(f, x_1,\ldots,x_n)$

SNAP and TERM

- Our **CONFIG** is essentially a snapshot function as seen in other presentations of a universal function

  SNAP(f, x, t) = CONFIG(f, x, t)
- Termination in our notation occurs when we reach a fixed point, so

  TERM(f, x) = (NEXT(f, x) == x)
- Again, we used a single argument but that can be extended as we have already shown.

STP Predicate

- **STP(f, x1,…,xn, t)** is a predicate defined to be true iff φ_f **(x1,…,xn)** converges in at most **t** steps.
- **STP** is primitive recursive since it can be defined by

  STP(f, x, *s*) = TERM(f, CONFIG(f, x, *s*))

  Extending to many arguments is easily done as before.

VALUE PRF

- **VALUE(f, x1,…,xn, t)** is a primitive recursive function (algorithm) that returns φ_f **(x1,…,xn)** so long as **STP(f, x1,…,xn, t)** is true.
- **VALUE(f, x1,…,xn, t)** returns a value if **STP(f, x1,…,xn, t)** is false, but the returned value is meaningless.

Recursively Enumerable

Properties of re Sets

Definition of re

- Some texts define re in the same way as I have defined semi-decidable.

  S $\subseteq \aleph$ is semi-decidable iff there exists a partially computable function **g** where

  $$\mathbf{S = \{ x \in \aleph \mid g(x)\downarrow \}}$$

- I prefer the definition of re that says
  S $\subseteq \aleph$ is re iff **S =** $\emptyset$ or there exists a totally computable function f where

  $$\mathbf{S = \{ y \mid \exists x\ f(x) == y \}}$$

- We will prove these equivalent. Actually, **f** can be a primitive recursive function.

Semi-Decidable Implies re

Theorem: Let **S** be semi-decided by $\mathbf{G_S}$. Assume $\mathbf{G_S}$ is the $\mathbf{g_S}$–th function in our enumeration of effective procedures. If **S = Ø** then **S** is re by definition, so we will assume wlog that there is some ***a*** ∈ **S**. Define the enumerating algorithm $\mathbf{F_S}$ by

$$\mathbf{F_S(<x,t>) = x * STP(g_s, x, t) + \mathit{a} * (1\text{-}STP(g_s, x, t))}$$

Note: $\mathbf{F_S}$ is <u>primitive recursive</u> and it enumerates every value in **S** infinitely often.

re Implies Semi-Decidable

Theorem: By definition, **S** is re iff **S == Ø** or there exists an algorithm $\mathbf{F_S}$, over the natural numbers $\aleph$, whose range is exactly **S**. Define

$$\psi_S(x) = \begin{cases} \mu y\,[y == y+1] & \text{if } S == \emptyset \\ \text{signum}((\mu y[F_S(y)==x])+1), & \text{otherwise} \end{cases}$$

This achieves our result as the domain of ψ_S is the range of $\mathbf{F_S}$, or empty if **S == Ø**. Note that this is an existence proof in that we cannot test if **S == Ø**

Domain of a Procedure

Corollary: **S** is re/semi-decidable iff **S** is the domain / range of a partial recursive predicate $\mathbf{F_S}$.

Proof: The predicate ψ_S we defined earlier to semi-decide **S**, given its enumerating function, can be easily adapted to have this property.

$$\psi_S(x) = \begin{cases} \mu y\,[y == y+1] & \text{if } S == \emptyset \\ x * \mathrm{signum}((\mu y[F_S(y)==x])+1), & \text{otherwise} \end{cases}$$

Recursive Implies re

Theorem: Recursive implies re.

Proof: **S** is recursive implies there is a total recursive function $\mathbf{f_S}$ such that

$$\mathbf{S = \{\ x \in \aleph \mid f_s(x) == 1\ \}}$$

Define $\mathbf{g_s(x) = \mu y\ (f_s(x) == 1)}$

Clearly

$$\begin{aligned}\mathbf{dom(g_s)} &= \mathbf{\{x \in \aleph \mid g_s(x)\downarrow\}} \\ &= \mathbf{\{\ x \in \aleph \mid f_s(x) == 1\ \}} \\ &= \mathbf{S}\end{aligned}$$

Related Results

Theorem: **S** is re iff **S** is semi-decidable.

Proof: That's what we proved.

Theorem: **S** and **~S** are both re (semi-decidable) iff **S** (equivalently **~S**) is recursive (decidable).

Proof: Let $\mathbf{f_S}$ semi-decide **S** and $\mathbf{f_{S'}}$ semi-decide **~S**. We can decide **S** by $\mathbf{g_S}$

$\mathbf{g_S(x) = STP(f_S, x, \mu t\ (STP(f_S, x, t) \mathbin{||} STP(f_{S'}, x, t))}$

~S is decided by $\mathbf{g_{S'}(x) = {\sim}g_S(x) = 1 - g_S(x)}$.

The other direction is immediate since, if **S** is decidable then **~S** is decidable (just complement $\mathbf{g_S}$) and hence they are both re (semi-decidable).

Enumeration Theorem

- Define
  $\mathbf{W_n = \{ x} \in \aleph \mathbf{\mid \varphi(n,x)\downarrow \}}$
- Theorem: A set **B** is re iff there exists an **n** such that $\mathbf{B = W_n}$.
  Proof: Follows from definition of $\varphi\mathbf{(n,x)}$.
- This gives us a way to enumerate the recursively enumerable sets.
- Note: We will later show (again) that we cannot enumerate the recursive sets.

The Set K

- $\mathbf{K = \{ n \in \aleph \mid n \in W_n \}}$
- Note that
  $\mathbf{n \in W_n \Leftrightarrow \varphi(n,n)\downarrow \Leftrightarrow HALT(n,n)}$
- Thus, **K** is the set consisting of the indices of each program that halts when given its own index
- **K** can be semi-decided by the **HALT** predicate above, so it is re.

K is not Recursive

- Theorem: We can prove this by showing **~K** is not re.
- If **~K** is re then $\mathbf{\sim K = W_i}$, for some **i**.
- However, this is a contradiction since
  $\mathbf{i} \in \mathbf{K} \Leftrightarrow \mathbf{i} \in \mathbf{W_i} \Leftrightarrow \mathbf{i} \in \mathbf{\sim K} \Leftrightarrow \mathbf{i} \notin \mathbf{K}$

re Characterizations

Theorem: If **S** $\neq \varnothing$ then the following are equivalent:

1. **S** is re
2. **S** is the range of a primitive rec. function
3. **S** is the range of a recursive function
4. **S** is the range of a partial rec. function
5. **S** is the domain of a partial rec. function
6. **S** is the range/domain of a partial rec. function whose domain is the same as its range and which acts as an identity when it converges. Below, assume $\mathbf{f_S}$ enumerates S.
   $\mathbf{g_S(x) = x*STP(f_S, x, \mu t\ (STP(f_S, x, t))}$ or
   $\mathbf{g_S(x) = x*\ \exists t\ STP(f_S, x, t)}$

S-m-n Theorem

Parameter (S-m-n) Theorem

- Theorem: For each **n,m>0**, there is a prf $\mathbf{S_m{}^n(y, u_1,\ldots,u_n)}$ such that

  $$\varphi^{(m+n)}(y, x_1,\ldots,x_m, u_1,\ldots,u_n) = \varphi^{(m)}(S_m{}^n(y,u_1,\ldots,u_n), x_1,\ldots, x_m)$$

- The proof of this is highly dependent on the system in which you proved universality and the encoding you chose.

S-m-n for FRS

- We would need to create a new FRS, from an existing one **F**, that fixes the value of $\mathbf{u_i}$ as the exponent of the prime $\mathbf{p_{m+i}}$.
- Sketch of proof:
  Assume we normally start with $\mathbf{p_1^{x1} \ldots p_m^{xm}\, p_1^{u1} \ldots p_{m+n}^{un}}\, \sigma$
  Here the first m are variable; the next **n** are fixed; σ denotes prime factors used to trigger first phase of computation.
  Assume that we use fixed point as convergence.
  We start with just $\mathbf{p_1^{x1} \ldots p_m^{xm}}$, with **q** the first unused prime.

$\mathbf{q}\ \alpha\ \mathbf{x} \rightarrow \mathbf{q}\ \beta\ \mathbf{x}$	replaces $\alpha\ \mathbf{x} \rightarrow \beta\ \mathbf{x}$ in **F**, for each rule in **F**
$\mathbf{q\ x} \rightarrow \mathbf{q\ x}$	ensures we loop at end
$\mathbf{x} \rightarrow \mathbf{q\ p_{m+1}^{u1} \ldots p_{m+n}^{un}}\ \sigma\ \mathbf{x}$	
	adds fixed input, start state and **q** this is selected once and never again

Note: **q = prime(max(n+m, lastFactor(Product[i=1 to r]** $\alpha_i\ \beta_i$ **))+1)**
where **r** is the number of rules in **F**.

Details of S-m-n for FRS

- The number of **F** (called **F**, also) is $\mathbf{2^r 3^{a_1} 5^{b_1} \ldots p_{2r-1}^{a_r} p_{2r}^{b_r}}$
- $\mathbf{S_{m,n}(F, u_1,\ldots u_n) = 2^{r+2} 3^{q\times a_1} 5^{q\times b_1} \ldots p_{2r-1}^{q\times a_r} p_{2r}^{q\times b_r}}$
  $\mathbf{p_{2r+1}^{q} p_{2r+2}^{q}\, p_{2r+3} p_{2r+4}^{\;q}\, p_{m+1}^{u_1} \ldots p_{m+n}^{u_n}\, \sigma}$
- This represents the rules we just talked about. The first added rule pair means that if the algorithm does not use fixed point, we force it to do so. The last rule pair is the only one initially enabled and it adds the prime **q**, the fixed arguments $\mathbf{u_1,\ldots u_n}$, the enabling prime **q**, and the **σ** needed to kick start computation. Note that **σ** could be a **1**, if no kick start is required.
- $\mathbf{S_{m,n} = S_m^{\,n}}$ is clearly primitive recursive. I'll leave the precise proof of that as a challenge to you.

Quantification 1 &2

Quantification#1

- **S** is decidable iff there exists an algorithm χ_S (called **S**'s characteristic function) such that
  $\mathbf{x} \in \mathbf{S} \Leftrightarrow \chi_S(\mathbf{x})$
  This is just the definition of decidable.
- **S** is re iff there exists an algorithm $\mathbf{A_S}$ where
  $\mathbf{x} \in \mathbf{S} \Leftrightarrow \exists \mathbf{t}\ \mathbf{A_S(x,t)}$
  This is clear since, if $\mathbf{g_S}$ is the index of the procedure ψ_S that semi-decides **S** then
  $\mathbf{x} \in \mathbf{S} \Leftrightarrow \exists \mathbf{t}\ \mathbf{STP(g_S, x, t)}$
  So, $\mathbf{A_S(x,t) = STP_{g_S}(\ x, t\ )}$, where $\mathbf{STP_{g_S}}$ is the **STP** function with its first argument fixed.
- Creating new functions by setting some one or more arguments to constants is an application of $\mathbf{S_m^n}$.

Quantification#2

- **S** is re iff there exists an algorithm $\mathbf{A_S}$ such that
  $\mathbf{x \notin S \Leftrightarrow \forall t\ A_S(x,t)}$
  This is clear since, if g_S is the index of the procedure ψ_S that semi-decides **S**, then
  $\mathbf{x \notin S \Leftrightarrow \sim\exists t\ STP(g_S, x, t) \Leftrightarrow \forall t \sim STP(g_S, x, t)}$
  So, $\mathbf{A_S(x,t) = \sim STP_{g_S}(\ x, t\ )}$, where $\mathbf{STP_{g_S}}$ is the **STP** function with its first argument fixed.
- Note that this works even if **S** is recursive (decidable). The important thing there is that if **S** is recursive then it may be viewed in two normal forms, one with existential quantification and the other with universal quantification.
- The complement of an re set is co-re. A set is recursive (decidable) iff it is both re and co-re.

Diagonalization and Reducibility

Non-re Problems

- There are even “practical” problems that are worse than unsolvable -- they’re not even semi-decidable.
- The classic non-re problem is the **Uniform Halting Problem**, that is, the problem to decide of an arbitrary effective procedure **P**, whether or not **P** is an algorithm.
- Assume that the algorithms can be enumerated, and that **F** accomplishes this. Then

  $\mathbf{F(x) = F_x}$

  where $\mathbf{F_0, F_1, F_2, \ldots}$ is a list of all the algorithms

The Contradiction

- Define $\quad$ $\mathbf{G(\,x\,) = Univ\,(\,F(x)\,,\,x\,) + 1 = \varphi(F(x), x)+1 = F_x(x) + 1}$
- But then **G** is itself an algorithm. Assume it is the **g**-th one

  $$\mathbf{F(g) = F_g = G}$$

  Then, $\quad$ $\mathbf{G(g) = F_g(g) + 1 = G(g) + 1}$
- But then **G** contradicts its own existence since **G** would need to be an algorithm.
- This cannot be used to show that the effective procedures are non-enumerable, since the above is not a contradiction when **G(g)** is undefined. In fact, we already have shown how to enumerate the (partial) recursive functions.

The Set TOT

- The listing of all algorithms can be viewed as

  $\textbf{TOT} = \{\, \mathbf{f} \in \aleph \mid \forall \mathbf{x}\ \varphi(\mathbf{f}, \mathbf{x})\!\downarrow \,\}$
- We can also note that

  $\textbf{TOT} = \{\, \mathbf{f} \in \aleph \mid \mathbf{W_f} = \aleph \,\}$
- Theorem: **TOT** is not re.

Reducibility

Reduction Concepts

- Proofs by contradiction are tedious after you've seen a few. We really would like proofs that build on known unsolvable problems to show other, open problems are unsolvable. The technique commonly used is called reduction. It starts with some known unsolvable problem and then shows that this problem is no harder than some open problem in which we are interested.

Diagonalization is a Bummer

- The issues with diagonalization are that it is tedious and is applicable as a proof of undecidability or non-re-ness for only a small subset of the problems that interest us.
- Thus, we will now seek to use reduction wherever possible.
- To show a set, **S**, is undecidable, we can show it is as least as hard as the set $\mathbf{K_0}$. That is, $\mathbf{K_0 \le S}$. Here the mapping used in the reduction does not need to run in polynomial time, it just needs to be an algorithm.
- To show a set, **S**, is not re, we can show it is as least as hard as the set **TOTAL (the set of algorithms)**. That is, **TOTAL ≤ S**.

Reduction to TOTAL

- We can show that the set $\mathbf{K_0}$ (Halting) is no harder than the set **TOTAL** (Uniform Halting). Since we already know that $\mathbf{K_0}$ is unsolvable, we would now know that **TOTAL** is also unsolvable. We cannot reduce in the other direction since **TOTAL** is in fact harder than $\mathbf{K_0}$.
- Let φ_F be some arbitrary effective procedure and let x be some arbitrary natural number.
- Define $\mathbf{F_x(y) = \varphi_F(x)}$, for all $\mathbf{y} \in \aleph$
- Then $\mathbf{F_x}$ is an algorithm if and only if φ_F halts on $\mathbf{x}$.
- Thus, $\mathbf{K_0 \leq TOTAL}$, and so a solution to membership in **TOTAL** would provide a solution to $\mathbf{K_0}$, which we know is not possible.

Reduction to ZERO

- We can show that the set **TOTAL** is no harder than the set **ZERO = { f | $\forall$x φ_f(x) = 0 }**. Since we already know that **TOTAL** is non-re, we would now know that **ZERO** is also non-re.
- Let φ_f be some arbitrary effective procedure.
- Define **F_f(y) = φ_f(x) – φ_f(x)**, for all **x** $\in \aleph$
- Then **F_f** is an algorithm that produces **0** for all input (is in the set **ZERO**) if and only if φ_f halts on all input **x**. Thus, **TOTAL ≤ ZERO.**
- Thus a semi-decision procedure for **ZERO** would provide one for **TOTAL**, a set already known to be non-re.

Classic Undecidable Sets

- The universal language
  $\mathbf{K_0 = L_u = \{ <f, x> \mid \varphi_f (x) \text{ is defined } \}}$

- Membership problem for $\mathbf{L_u}$ is the **Halting Problem**.
- The sets $\mathbf{L_{ne}}$ and $\mathbf{L_e}$, where

  $\mathbf{NON\text{-}EMPTY = L_{ne} = \{ f \mid \exists\, x\, \varphi_f (x) \downarrow \}}$

  $\mathbf{EMPTY = L_e = \{ f \mid \forall\, x\, \varphi_f (x) \uparrow \}}$

  are the next ones we will study.

L_{ne} is re

- $\mathbf{L_{ne}}$ is enumerated by

  $$\mathbf{F(\ <f,\ x,\ t>\ ) = f * STP(\ f,\ x,\ t\ )}$$

- This assumes that **0** is in $\mathbf{L_{ne}}$ since **0** probably encodes some trivial machine. If this isn't so, we'll just slightly vary our enumeration of the recursive functions so it is true.
- Thus, the range of this total function **F** is exactly the indices of functions that converge for some input, and that's $\mathbf{L_{ne}}$.

L_{ne} is Non-Recursive

- Note in the previous enumeration that **F** is a function of just one argument, as we are using an extended pairing function $\mathbf{<x,y,z> = <x,<y,z>>}$.
- Now $\mathbf{L_{ne}}$ cannot be recursive, for if it were then $\mathbf{L_u}$ $(K_0)$is recursive by the reduction we showed before.
- In particular, from any index **x** and input **y**, we created a new function which accepts all input just in case the **x**-th function accepts **y**. Recall $\mathbf{F_x(y) = \varphi_F(x)}$, for all $\mathbf{y} \in \aleph$**.**
- Hence, this new function's index is in $\mathbf{L_{ne}}$ just in case $<\mathbf{x, y}>$ is in $\mathbf{L_u}$ (K_0).
- Thus, a decision procedure for $\mathbf{L_{ne}}$ (equivalently for $\mathbf{L_e}$) implies one for $\mathbf{L_u}$ (K_0).

L_{ne} is re by Quantification

- Can do by observing that

  $$\mathbf{f \in L_{ne} \Leftrightarrow \exists \langle x,t \rangle\ STP(\ f,\ x,\ t)}$$

- By our earlier results, any set whose membership can be described by an existentially quantified recursive predicate is re (semi-decidable).

L_e is not re

- If $\mathbf{L_e}$ were re, then $\mathbf{L_{ne}}$ would be recursive since it and its complement would be re.
- Can also observe that $\mathbf{L_e}$ is the complement of an re set since

$$\begin{aligned}
\mathbf{f} \in \mathbf{L_e} \quad & \Leftrightarrow \forall \mathbf{<x,t> \sim STP(\, f,\, x,\, t)} \\
& \Leftrightarrow \sim\exists \mathbf{<x,t>\ STP(\, f,\, x,\, t)} \\
& \Leftrightarrow \mathbf{f} \notin \mathbf{L_{ne}}
\end{aligned}$$

Reduction and Equivalence

m-1, 1-1, Turing Degrees

Many-One Reduction

- Let **A** and **B** be two sets.
- We say **A** many-one reduces to **B**, $\mathbf{A} \le_{\mathbf{m}} \mathbf{B}$, if there exists a total recursive function **f** such that
  $\mathbf{x} \in \mathbf{A} \Leftrightarrow \mathbf{f(x)} \in \mathbf{B}$
- We say that **A** is many-one equivalent to **B**, $\mathbf{A} \equiv_{\mathbf{m}} \mathbf{B}$, if $\mathbf{A} \le_{\mathbf{m}} \mathbf{B}$ and $\mathbf{B} \le_{\mathbf{m}} \mathbf{A}$
- Sets that are many-one equivalent are in some sense equally hard or easy.

Many-One Degrees

- The relationship **A** $\equiv_m$ **B** is an equivalence relationship (why?)
- If **A** $\equiv_m$ **B**, we say **A** and **B** are of the same many-one degree (of unsolvability).
- Decidable problems occupy three m-1 degrees: $\varnothing$, $\aleph$, all others.
- The hierarchy of undecidable m-1 degrees is an infinite lattice (I'll discuss in class)

One-One Reduction

- Let **A** and **B** be two sets.
- We say **A** one-one reduces to **B**, $\mathbf{A} \leq_1 \mathbf{B}$,
  if there exists a total recursive 1-1 function **f** such that
  $\mathbf{x} \in \mathbf{A} \Leftrightarrow \mathbf{f(x)} \in \mathbf{B}$
- We say that **A** is one-one equivalent to **B**,
  $\mathbf{A} \equiv_1 \mathbf{B}$, if $\mathbf{A} \leq_1 \mathbf{B}$ and $\mathbf{B} \leq_1 \mathbf{A}$
- Sets that are one-one equivalent are in a strong sense equally hard or easy.

One-One Degrees

- The relationship **A** $\equiv_1$ **B** is an equivalence relationship (why?)
- If **A** $\equiv_1$ **B**, we say **A** and **B** are of the same one-one degree (of unsolvability).
- Decidable problems occupy infinitely many 1-1 degrees: each cardinality defines another 1-1 degree (think about it).
- The hierarchy of undecidable 1-1 degrees is an infinite lattice.

Turing (Oracle) Reduction

- Let **A** and **B** be two sets.
- We say **A** Turing reduces to **B**, **A** $\leq_t$ **B**, if the existence of an oracle for **B** would provide us with a decision procedure for **A**.
- We say that **A** is Turing equivalent to **B**, **A** $\equiv_t$ **B**, if **A** $\leq_t$ **B** and **B** $\leq_t$ **A**
- Sets that are Turing equivalent are in a very loose sense equally hard or easy.

Turing Degrees

- The relationship **A** $\equiv_t$ **B** is an equivalence relationship (why?)
- If **A** $\equiv_t$ **B**, we say **A** and **B** are of the same Turing degree (of unsolvability).
- Decidable problems occupy one Turing degree. We really don't even need the oracle.
- The hierarchy of undecidable Turing degrees is an infinite lattice.

Complete re Sets

- A set **C** is re 1-1 (m-1, Turing) complete if, for any re set **A**, **A** $\leq_1$ ($\leq_m$, $\leq_t$) **C**.
- The set **HALT** is an re complete set (in regard to 1-1, m-1 and Turing reducibility).
- The re complete degree (in each sense of degree) sits at the top of the lattice of re degrees.

The Set Halt = K_0 = L_u

- **Halt = K_0 = L_u = { <f, x> | φ_f (x) $\downarrow$}**
- Let A be an arbitrary re set. By definition, there exists an effective procedure φ_a, such that **dom(φ_a) = A**. Put equivalently, there exists an index, **a**, such that **A = W_a**.
- **x** $\in$ **A** iff **x** $\in$ **dom(φ_a)** iff φ_a(x)$\downarrow$ iff **<a,x>** $\in$ **K_0**
- The above provides a 1-1 function that reduces **A** to **K_0** (**A** $\leq_1$ **K_0**)
- Thus the universal set, **Halt = K_0 = L_u**, is an re (1-1, m-1, Turing) complete set.

The Set K

- $\mathbf{K = \{ f \mid \varphi_f(f) \text{ is defined } \}}$
- Define $\mathbf{f_x(y) = \varphi_f(x)}$, for all $\mathbf{y}$. The index for $\mathbf{f_x}$ can be computed from $\mathbf{f}$ and $\mathbf{x}$ using $\mathbf{S_{1,1}}$, where we add a dummy argument, $\mathbf{y}$, to $\varphi_\mathbf{f}$. Let that index be $\mathbf{f_x}$. (Yeah, that's overloading.)
- $\mathbf{\langle f,x \rangle} \in \mathbf{K_0}$ iff $\mathbf{x} \in \mathbf{dom(\varphi_f)}$ iff $\forall\mathbf{y[\varphi_{f_x}(y)\downarrow]}$ iff $\mathbf{f_x} \in \mathbf{K}$.
- The above provides a 1-1 function that reduces $\mathbf{K_0}$ to $\mathbf{K}$.
- Since $\mathbf{K_0}$ is an re (1-1, m-1, Turing) complete set and $\mathbf{K}$ is re, then $\mathbf{K}$ is also re (1-1, m-1, Turing) complete.

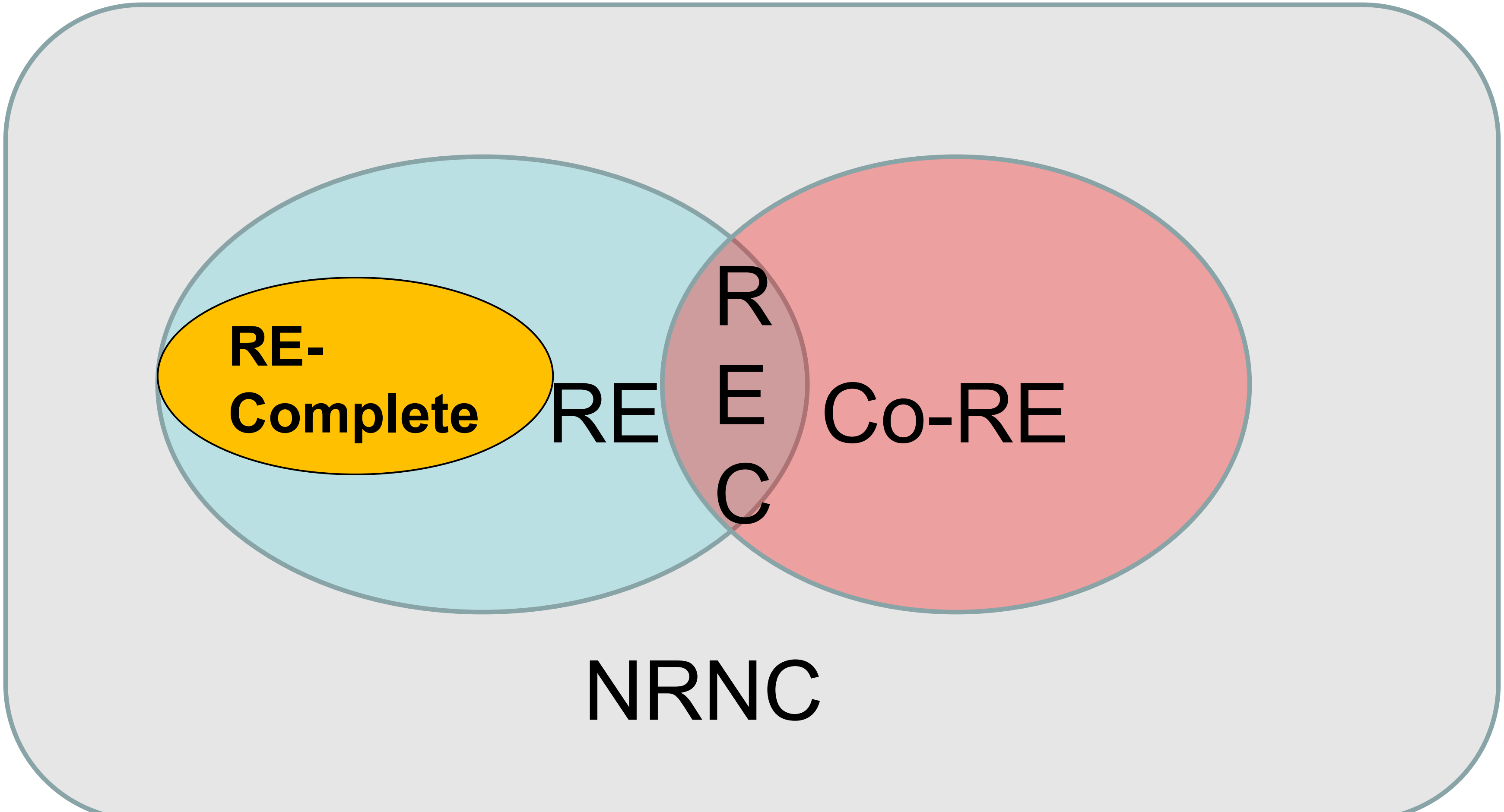
Quantification # 3 and the Overall Picture

Quantification#3

- The **Uniform Halting Problem** was already shown to be non-re. It turns out its complement is also not re. We'll cover that later. In fact, we will show that **TOT** requires an alternation of quantifiers. Specifically,

  f $\in$ TOT$\Leftrightarrow$ $\forall$x$\exists$t (STP(f, x, t))
  and this is the minimum quantification we can use, given that the quantified predicate is total recursive (actually primitive recursive here).

UNIVERSE OF SETS



NonRE = (NRNC ∪ Co-RE) - REC

Reduction and Rice's

Either Trivial or Undecidable

- Let **P** be some set of re languages, e.g. **P = {** **L** | **L** is infinite re }.
- We call **P** a property of re languages since it divides the class of all re languages into two subsets, those having property **P** and those not having property **P**.
- **P** is said to be trivial if it is empty (this is not the same as saying **P** contains the empty set) or contains all re languages.
- Trivial properties are not very discriminating in the way they divide up the re languages (all or nothing).

Rice's Theorem

Rice's Theorem: Let **P** be some non-trivial property of the re languages. Then

$$\mathbf{L_P = \{\ x \mid dom\ [x]\ is\ in\ P\ (has\ property\ P)\ \}}$$

is undecidable. Note that membership in $\mathbf{L_P}$ is based purely on the domain of a function, not on any aspect of its implementation.

Rice's Proof-1

Proof: We will assume, *wlog*, that **P** does not contain **Ø**. If it does we switch our attention to the complement of **P**. Now, since **P** is non-trivial, there exists some language **L** with property **P**. Let **[r]** be a recursive function whose domain is **L** (**r** is the index of a semi-decision procedure for **L**). Suppose **P** were decidable. We will use this decision procedure and the existence of **r** to decide $\mathbf{K_0}$.

Rice's Proof-2

First we define a function $\mathbf{F_{r,x,y}}$ for **r** and each function $\varphi_{\mathbf{x}}$ and input **y** as follows.

$$\mathbf{F_{r,x,y}(z) = \varphi(x , y) + \varphi(r , z)}$$

The domain of this function is **L** if $\varphi_{\mathbf{x}}$ **(y)** converges, otherwise it's **Ø**. Now if we can determine membership in $\mathbf{L_P}$, we can use this algorithm to decide $\mathbf{K_0}$ merely by applying it to $\mathbf{F_{r,x,y}}$. An answer as to whether or not $\mathbf{F_{r,x,y}}$ has property **P** is also the correct answer as to whether or not $\varphi_{\mathbf{x}}$ **(y)** converges.

Rice's Proof-3

Thus, there can be no decision procedure for **P**. And consequently, there can be no decision procedure for any non-trivial property of re languages.

Note: This does not apply if **P** is trivial, nor does it apply if **P** can differentiate indices that converge for precisely the same values.

I/O Property

- An I/O property, $\mathscr{P}$, of indices of recursive function is one that cannot differentiate indices of functions that produce precisely the same value for each input.
- This means that if two indices, **f** and **g**, are such that $\varphi_{\mathbf{f}}$ and $\varphi_{\mathbf{g}}$ converge on the same inputs and, when they converge, produce precisely the same result, then both **f** and **g** must have property $\mathscr{P}$, or neither one has this property.
- Note that any I/O property of recursive function indices also defines a property of re languages, since the domains of functions with the same I/O behavior are equal. However, not all properties of re languages are I/O properties.

Strong Rice's Theorem

Rice's Theorem: Let $\mathscr{P}$ be some non-trivial I/O property of the indices of recursive functions. Then

$$\mathbf{S}_{\mathscr{P}} = \{ \mathbf{x} \mid \varphi_{\mathbf{x}} \textbf{ has property } \mathscr{P}) \}$$

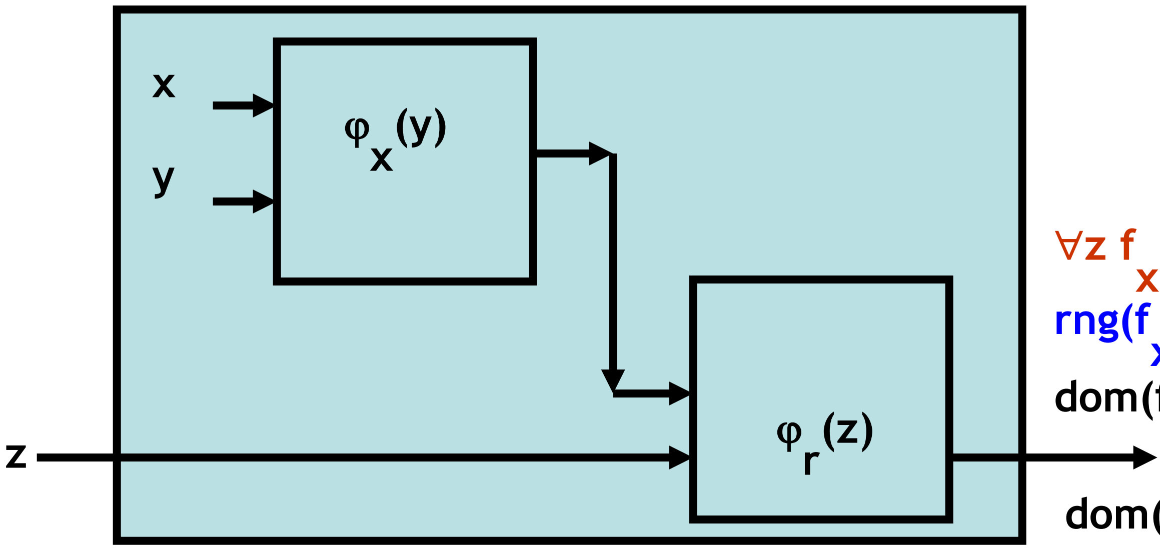
is undecidable. Note that membership in $\mathbf{S}_{\mathscr{P}}$ is based purely on the input/output behavior of a function, not on any aspect of its implementation.

Strong Rice's Proof

- Given **x**, **y**, **r**, where **r** is in the set
  $\mathbf{S}_{\mathcal{P}} \mathbf{.= \{f \mid} \varphi_{\mathbf{f}}$ has property $\mathcal{P}$},
  define the function
  $\mathbf{f_{x,y,r}(z) = \varphi_x(y) - \varphi_x(y) + \varphi_r(z)}$.

- $\mathbf{f_{x,y,r}(z) = \varphi_r(z)}$ **if** $\varphi_{\mathbf{x}}\mathbf{(y)}\downarrow$ **; =** ϕ **if** $\varphi_{\mathbf{x}}\mathbf{(y)}\uparrow$.
  Thus, $\varphi_{\mathbf{x}}\mathbf{(y)}\downarrow$ iff $\mathbf{f_{x,y,r}}$ has property $\mathcal{P}$, and so
  $\mathbf{K_0} \leq \mathbf{S}_{\mathcal{P}}$**.**

Picture Proof



$\forall z\ f_{x,y,r}(z)=\varphi_r(z)$ If $\varphi_x(y)\downarrow$

$rng(f_{x,y,r})=rng(\varphi_r)$ If $\varphi_x(y)\downarrow$

$dom(f_{x,y,r})=dom(\varphi_r)$ If $\varphi_x(y)\downarrow$

$dom(f_{x,y,r})=\phi$ If $\varphi_x(y)\uparrow$

$rng(f_{x,y,r})=\phi$ If $\varphi_x(y)\uparrow$

$\exists z\ f_{x,y,r}(z)\neq\varphi_r(z)$ If $\varphi_x(y)\uparrow$

Black is for standard Rice's Theorem;
Black and Red are needed for Strong Version
Blue is just another version based on range

Corollaries to Rice's

Corollary: The following properties of re sets are undecidable

a) **L** $= \emptyset$

b) **L** is finite

c) **L** is a regular set

d) **L** is a context-free set

Practice

Known Results:

HALT = { <f,x> | f(x)↓ } is re (semi-decidable) but undecidable

TOTAL = { f | ∀x f(x)↓ } is non-re (not even semi-decidable)

1. Use reduction from **HALT** to show that one cannot decide **NonTrivial**, where **NonTrivial = { f | for some x, y, x ≠ y, f(x)↓ and f(y)↓ and f(x) ≠ f(y) }**
2. Show that **Non-Trivial** reduces to **HALT**. (1 plus 2 show they are equally hard)
3. Use Reduction from **TOTAL** to show that **NoRepeats** is not even re, where **NoRepeats = { f | for all x, y, f(x)↓ and f(y)↓, and x ≠ y ⇒ f(x) ≠ f(y) }**
4. Show **NoRepeats** reduces to **TOTAL**. (3 plus 4 show they are equally hard)
5. Use Rice’s Theorem to show that **NonTrivial** is undecidable
6. Use Rice’s Theorem to show that **NoRepeats** is undecidable

Practice Classifications

1. Use quantification of an algorithmic predicate to estimate the complexity (decidable, re, co-re, non-re) of each of the following, (a)-(d):
   a) **NonTrivial = { f | for some x, y, x ≠ y, f(x)↓ and f(y)↓ and f(x) ≠ f(y) }**
   b) **NoRepeats = { f | for all x, y, f(x)↓ and f(y)↓, and x ≠ y ⇒ f(x) ≠ f(y) }**
   c) **FIN = { f | domain(f) is finite }**
2. Let set **A** be non-empty recursive, and let **B** be re non-recursive. Consider **C = { z | z = x * y,** where **x** ∈ **A** and **y** ∈ **B }**. . For (a)-(c), either show sets **A** and **B** with the specified property or demonstrate that this property cannot hold.
   a) **Can C be recursive?**
   b) **Can C be re non-recursive (undecidable)?**
   c) **Can C be non-re?**

Sample Question#1

1. Given that the predicate **STP** and the function **VALUE** are algorithms, show that we can semi-decide

   HZ = { f | φ_f evaluates to 0 for some input}

   Note: **STP(f, x, s)** is true iff φ_f**(x)** converges in **s** or fewer steps and, if so, **VALUE(f, x, s) =** φ_f**(x)**.

Sample Questions#2,3

2. Use Rice's Theorem to show that **HZ** is undecidable, where **HZ** is

   HZ = { f | φ_f evaluates to 0 for some input}

3. Redo using Reduction from **HALT**.

Sample Question#4

4. Let **P = { f | ∃ x [STP(f, x, x)] }**. Why does Rice's theorem not tell us anything about the undecidability of **P**?

Sample Question#5

5. Let **S** be an re (recursively enumerable), non-recursive set, and **T** be an re, possibly recursive non-empty set. Let

   $\mathbf{E = \{ z \mid z = x + y, \text{ where } x \in S \text{ and } y \in T \}}$.

   Answer with proofs, algorithms or counterexamples, as appropriate, each of the following questions:

   (a) Can **E** be non re?

   (b) Can **E** be re non-recursive?

   (c) Can **E** be recursive?

Constant time: Not amenable to Rice's

Constant Time

- **CTime = { M | ∃K [M** halts in at most **K** steps independent of its starting configuration **] }**
- **RT** cannot be shown undecidable by Rice's Theorem as it breaks property 2
  - Choose **M1** and **M2** to each Standard Turing Compute (STC) **ZERO**
  - **M1** is **R** (move right to end on a zero)
  - **M2** is $\mathcal{L}\ \mathcal{R}$ **R** (time is dependent on argument)
  - **M1** is in **CTime; M2** is not , but they have same I/O behavior, so **CTime** does not adhere to property 2

Quantifier Analysis

- **CTime = { M | $\exists$K $\forall$C [STP(M, C, K)] }**
- This would appear to imply that **CTime** is not even re. However, a TM that only runs for **K** steps can only scan at most **K** distinct tape symbols. Thus, if we use unary notation, **CTime** can be expressed
- **CTime = { M | $\exists$K $\forall$C$_{|C|\leq K}$ [STP(M, C, K)] }**
- We can dovetail over the set of all TMs, **M**, and all **K**, listing those **M** that halt in constant time.

Complexity of CTime

- Can show it is equivalent to the **Halting Problem** for TM's with **Infinite Tapes** (not unbounded but truly infinite)
- This was shown in 1966 to be undecidable.
- It was also shown to be re, just as we have done so for **CTime**.
- Details Later

What We've Done in Computability

List Minus Some Tedious Stuff

- A question with multiple parts that uses quantification (STP/VALUE)
- Various re and recursive equivalent definitions
- Proofs of equivalence of definitions
- Consequences of recursiveness or re-ness of a problem
- Closure of recursive/re sets
- Gödel numbering (pairing functions and inverses)
- Models of computation/equivalences (not details but understanding)
- Primitive recursion and its limitation; bounded versus unbounded μ
- Notion of universal machine
- A proof by diagonalization (there are just two possibilities)
- A question about K and/or K_0
- Many-one reduction(s)
- Rice's Theorem (its proof and its variants)
- Applications of Rice's Theorem and when it cannot be applied

More Practice Problems

Sample Question#1

1. Prove that the following are equivalent
   a) **S is an infinite recursive (decidable) set.**
   b) **S is the range of a monotonically increasing total recursive function. Note: f is monotonically increasing means that $\forall x\ f(x+1) > f(x)$.**

Sample Question#2

2. Let A and B be re sets. For each of the following, either prove that the set is re, or give a counterexample that results in some known non-re set.
   a) $\mathbf{A \cup B}$
   b) $\mathbf{A \cap B}$
   c) $\mathbf{\sim A}$

Sample Question#3

3. Present a demonstration that the *even* function is primitive recursive.
   even(x) = 1 if x is even
   even(x) = 0 if x is odd
   You may assume only that the base functions are prf and that prf's are closed under a finite number of applications of composition and primitive recursion.

Sample Question#4

4. Given that the predicate **STP** and the function **VALUE** are prf's, show that we can semi-decide

   { f | φ_f evaluates to 0 for some input}

   Note: **STP(f, x, s)** is true iff $\varphi_f(x)$ converges in **s** or fewer steps and, if so, **VALUE(f, x, s) = $\varphi_f(x)$**.

Sample Question#5

5. Let **S** be an re (recursively enumerable), non-recursive set, and **T** be an re, possibly recursive set. Let

   $\mathbf{E = \{ z \mid z = x + y, \text{ where } x \in S \text{ and } y \in T \}}$.

   Answer with proofs, algorithms or counterexamples, as appropriate, each of the following questions:

   (a) Can **E** be non re?

   (b) Can **E** be re non-recursive?

   (c) Can **E** be recursive?

Sample Question#6

6. Assuming that the Uniform Halting Problem (**TOTAL**) is undecidable (it's actually not even re), use reduction to show the undecidability of

$$\{ f \mid \forall x\ \varphi_f(x+1) > \varphi_f(x) \}$$

Sample Question#7

7. Let **Incr = { f | $\forall$x, φ_f(x+1)>φ_f(x) }**.
   Let **TOT = { f | $\forall$x, φ_f(x)$\downarrow$ }**.
   Prove that **Incr** $\equiv_m$ **TOT**. Note Q#6 starts this one.

Sample Question#8

8. Let **Incr = { f | ∀x $\varphi_f(x+1)>\varphi_f(x)$ }**. Use Rice's theorem to show **Incr** is not recursive.

Sample Question#9

9. Let **S** be a recursive (decidable set), what can we say about the complexity (recursive, re non-recursive, non-re) of **T**, where **T** ⊂ **S**?

Sample Question#10

10. Define the pairing function $\mathbf{<x,y>}$ and its two inverses $\mathbf{<z>_1}$ and $\mathbf{<z>_2}$, where if $\mathbf{z = <x,y>}$, then $\mathbf{x = <z>_1}$ and $\mathbf{y = <z>_2}$.

Sample Question#11

11. Assume **A** $\leq_{\mathbf{m}}$ **B** and **B** $\leq_{\mathbf{m}}$ **C**.
    Prove **A** $\leq_{\mathbf{m}}$ **C**.

Sample Question#12

12. Let **P = { f | ∃ x [STP(f, x, x)] }**. Why does Rice's theorem not tell us anything about the undecidability of **P**?

Post Systems

Thue Systems

- Devised by Axel Thue
- Just a string rewriting view of finitely presented monoids
- T = (Σ, R), where Σ is a finite alphabet and R is a finite set of bi-directional rules of form $\alpha_i \leftrightarrow \beta_i$, α_i, $\beta_i \in \Sigma^*$
- We define $\Leftrightarrow^*$ as the reflexive, transitive closure of $\Leftrightarrow$, where w $\Leftrightarrow$ x iff w=yαz and x=yβz, where $\alpha \leftrightarrow \beta$

Semi-Thue Systems

- Devised by Emil Post
- A one-directional version of Thue systems
- S = (Σ, R), where Σ is a finite alphabet and R is a finite set of rules of form $\alpha_i \rightarrow \beta_i$, α_i, $\beta_i \in \Sigma^*$
- We define $\Rightarrow^*$ as the reflexive, transitive closure of $\Rightarrow$, where w $\Rightarrow$ x iff w=yαz and x=yβz, where $\alpha \rightarrow \beta$

Word Problems

- Let $S = (\Sigma, R)$ be some Thue (Semi-Thue) system, then the word problem for S is the problem to determine of arbitrary words w and x over S, whether or not $w \Leftrightarrow^* x$ ($w \Rightarrow^* x$)
- The Thue system word problem is the problem of determining membership in equivalence classes. This is not true for Semi-Thue systems.
- We can always consider just the relation $\Rightarrow^*$ since the symmetric property of $\Leftrightarrow^*$ comes directly from the rules of Thue systems.

Post Canonical Systems

- These are a generalization of Semi-Thue systems.
- P = (Σ, V, R), where Σ is a finite alphabet, V is a finite set of "variables", and R is a finite set of rules.
- Here the premise part (left side) of a rule can have many premise forms, e.g, a rule appears as
  $P_{1,1}\alpha_{1,1}\ P_{1,2}\ldots\ \alpha_{1,n_1}P_{1,n_1}\alpha_{1,n_1+1}$,
  $P_{2,1}\alpha_{2,1}\ P_{2,2}\ldots\ \alpha_{2,n_2}P_{2,n_2}\alpha_{2,n_2+1}$,
  …
  $P_{k,1}\alpha_{k,1}\ P_{k,2}\ldots\ \alpha_{k,n_k}P_{k,n_k}\alpha_{k,n_k+1}$,
  $\rightarrow Q_1\beta_1\ Q_2\ldots\ \beta_{n_{k+1}}Q_{n_{k+1}}\beta_{n_{k+1}+1}$
- In the above, the P's and Q's are variables, the α's and β's are strings over Σ, and each Q must appear in at least one premise.
- We can extend the notion of $\Rightarrow^*$ to these systems considering sets of words that derive conclusions. Think of the original set as axioms, the rules as inferences and the final word as a theorem to be proved.

Examples of Canonical Forms

- Propositional rules
  $P, P \supset Q \rightarrow Q$
  $\sim P, P \cup Q \rightarrow Q$
  $P \cap Q \rightarrow P$
  $P \cap Q \rightarrow Q$
  $(P \cap Q) \cap R \leftrightarrow P \cap (Q \cap R)$
  $(P \cup Q) \cup R \leftrightarrow P \cup (Q \cup R)$
  $\sim(\sim P) \leftrightarrow P$
  $P \cup Q \rightarrow Q \cup P$
  $P \cap Q \rightarrow Q \cap P$

oh, oh $a \cap (b \cap c) \Rightarrow a \cap (b$

- Some proofs over $\{a,b,(,),\sim,\supset,\cup,\cap\}$
  $\{a \cup c, b \supset \sim c, b\} \Rightarrow \{a \cup c, b \supset \sim c, b, \sim c\} \Rightarrow$
  $\{a \cup c, b \supset \sim c, b, \sim c, c \cup a\} \Rightarrow$
  $\{a \cup c, b \supset \sim c, b, \sim c, c \cup a, a\}$ which proves "a"

Simplified Canonical Forms

- Each rule of a Semi-Thue system is a canonical rule of the form
  $P\alpha Q \rightarrow P\beta Q$
- Each rule of a Thue system is a canonical rule of the form
  $P\alpha Q \leftrightarrow P\beta Q$
- Each rule of a Post Normal system is a canonical rule of the form
  $\alpha P \rightarrow P\beta$
- Tag systems are just Normal systems where all premises are of the same length (the deletion number), and at most one can begin with any given letter in Σ. That makes Tag systems deterministic.

Examples of Post Systems

- Alphabet Σ = {a,b,#}. Semi-Thue rules:
  aba $\rightarrow$ b
  #b# $\rightarrow \lambda$
  For above, $\#a^nba^m\# \Rightarrow^* \lambda$ iff n=m
- Alphabet Σ = {0,1,c,#}. Normal rules:
  0c $\rightarrow$ 1
  1c $\rightarrow$ c0
  #c $\rightarrow$ #1
  0 $\rightarrow$ 0
  1 $\rightarrow$ 1
  # $\rightarrow$ #
  For above, *binaryc#* $\Rightarrow^*$ *binary+1#* where *binary* is some binary number.

Simulating Turing Machines

- Basically, we need at least one rule for each 4-tuple in the Turing machine's description.
- The rules lead from one instantaneous description to another.
- The Turing ID α**qa**β is represented by the string hα**qa**βh, **a** being the scanned symbol.
- The tuple **q a b s** leads to
  qa $\rightarrow$ **sb**
- Moving right and left can be harder due to blanks.

Details of Halt(TM) $\leq$ Word(ST)

- Let M = (Q, {0,1}, T), T is Turing table.
- If qabs $\in$ T, add rule qa $\rightarrow$ sb
- If qaRs $\in$ T, add rules
  - q1b $\rightarrow$ 1sb a=1, $\forall$b$\in${0,1}
  - q1h $\rightarrow$ 1s0h a=1
  - cq0b $\rightarrow$ c0sb a=0, $\forall$b,c$\in${0,1}
  - hq0b $\rightarrow$ hsb a=0, $\forall$b$\in${0,1}
  - cq0h $\rightarrow$ c0s0h a=0, $\forall$c$\in${0,1}
  - hq0h $\rightarrow$ hs0h a=0
- If qaLs $\in$ T, add rules
  - bqac $\rightarrow$ sbac $\forall$a,b,c$\in${0,1}
  - hqac $\rightarrow$ hs0ac $\forall$a,c$\in${0,1}
  - bq1h $\rightarrow$ sb1h a=1, $\forall$b$\in${0,1}
  - hq1h $\rightarrow$ hs01h a=1
  - bq0h $\rightarrow$ sbh a=0, $\forall$b$\in${0,1}
  - hq0h $\rightarrow$ hs0h a=0

Clean-Up

- Assume q_1 is start state and only one accepting state exists q_0
- We will start in $h1^xq_10h$, seeking to accept x (enter q_0) or reject (run forever).
- Add rules
  - $q_0a \to q_0$ $\quad\quad\quad\quad \forall a \in \{0,1\}$
  - $bq_0 \to q_0$ $\quad\quad\quad\quad \forall b \in \{0,1\}$

- The added rule allows us to “erase” the tape if we accept x.
- This means that acceptance can be changed to generating hq_0h.

- The next slide shows the consequences.

Semi-Thue Word Problem

- Construction from TM, M, gets:
- $h1^xq_10h \Rightarrow_{\Sigma(M)}{}^* hq_0h$ iff $x \in \mathcal{L}(M)$.
- $hq_0h \Rightarrow_{\Pi(M)}{}^* h1^xq_10h$ iff $x \in \mathcal{L}(M)$.
- $hq_0h \Leftrightarrow_{\Sigma(M)}{}^* h1^xq_10h$ iff $x \in \mathcal{L}(M)$.
- Can recast both Semi-Thue and Thue Systems to ones over alphabet {a,b} or {0,1}. That is, a binary alphabet is sufficient for undecidability.

Formal Language

Undecidability Continued

PCP and Traces

Post Correspondence Problem

- Many problems related to grammars can be shown to be no more complex than the Post Correspondence Problem (PCP).
- Each instance of PCP is denoted: Given n>0, Σ a finite alphabet, and two n-tuples of words
  $(x_1, \dots , x_n), (y_1, \dots , y_n)$ over Σ,
  does there exist a sequence $i_1, \dots , i_k$, $k>0$, $1 \le i_j \le n$, such that
  $x_{i_1} \dots x_{i_k} = y_{i_1} \dots y_{i_k}$?
- Example of PCP:
  $n = 3$, $\Sigma = \{ a , b \}$, (a b a , b b , a), (b a b , b , b a a).
  Solution 2 , 3, 1 , 2
  b b a a b a b b = b b a a b a b b

PCP Example#2

- Start with Semi-Thue System
  - aba $\rightarrow$ ab; a $\rightarrow$ aa; b $\rightarrow$ a
  - Instance of word problem: bbbb $\Rightarrow^*$? aa
- Convert to PCP

–	[bbbb*	ab	$\underline{\text{ab}}$	aa	$\underline{\text{aa}}$	a	$\underline{\text{a}}$	]
	[	$\underline{\text{aba}}$	aba	$\underline{\text{a}}$	a	$\underline{\text{b}}$	b	$\underline{*}$aa]
–	And	*	$\underline{*}$	a	$\underline{\text{a}}$	b	$\underline{\text{b}}$	
		$\underline{*}$	*	$\underline{\text{a}}$	a	$\underline{\text{b}}$	b	

How PCP Construction Works?

- Using underscored letters avoids solutions that don’t relate to word problem instance. E.g.,

    aba a
    ab   aa

- Top row insures start with $[W_0*$
- Bottom row insures end with $\underline{*}W_f]$
- Bottom row matches W_i, while top matches W_{i+1} (one is underscored)

Ambiguity of CFG

- Problem to determine if an arbitrary CFG is ambiguous

  $S \to A \mid B$

  $A \to x_i\ A\ [i] \mid x_i\ [i] \qquad 1 \le i \le n$

  $B \to y_i\ B\ [i] \mid y_i\ [i] \qquad 1 \le i \le n$

  $A \Rightarrow^* x_{i_1} \ldots x_{i_k}\ [i_k] \ldots [i_1] \qquad k > 0$

  $B \Rightarrow^* y_{i_1} \ldots y_{i_k}\ [i_k] \ldots [i_1] \qquad k > 0$

- Ambiguous if and only if there is a solution to this PCP instance.

Intersection of CFLs

- Problem to determine if arbitrary CFG's define overlapping languages
- Just take the grammar consisting of all the A-rules from previous, and a second grammar consisting of all the B-rules. Call the languages generated by these grammars, L_A and L_B.
  $L_A \cap L_B \neq \emptyset$, if and only there is a solution to this PCP instance.

CSG Produces Something

S	$\rightarrow x_i\ S\ y_i^R \mid x_i\ T\ y_i^R \quad 1 \le i \le n$
a T a	$\rightarrow$ * T *
* a	$\rightarrow$ a *
a *	$\rightarrow$ * a
T	$\rightarrow$ *

- Our only terminal is *. We get strings of form $*^{2j+1}$, for some j's if and only if there is a solution to this PCP instance.

Traces and Grammars

Traces

- A valid trace
  - $\# C_1 \# C_2 \# C_3 \# C_4 \; \ldots \# C_{k-1} \# C_k \#$,
    where $k \geq 1$ and $C_i \Rightarrow_M C_{i+1}$, for $1 \leq i < k$.
    Here, $\Rightarrow_M$ means derive in **M**, and C is a valid ID (Instantaneous Description)
- An invalid trace
  - $\# C_1 \# C_2 \# C_3 \# C_4 \; \ldots \# C_{k-1} \# C_k \#$,
    where $k \geq 1$ and for some i, it is false that $C_i \Rightarrow_M C_{i+1}$.

Traces (Valid Computations)

- A terminating trace of a machine **M**, is a word of the form
$\# \ C_0 \ \# \ C_1 \ \# \ C_2 \ \# \ C_3 \ \# \ \ldots \ \# \ C_{k-1} \ \# \ C_k \ \#$

  where $C_i \Rightarrow C_{i+1}$ $0 \le i < k$, X_0 is a starting configuration and C_k is a terminating configuration.
- We allow some laxness, where the configurations might be encoded in a manner appropriate to the machine model. Now, a context free grammar can be devised which approximates traces by either getting the even-odd pairs right, or the odd-even pairs right. The goal is to then intersect the two languages, so the result is a trace. This then allows us to create CFLs L1 and L2, where $L1 \cap L2 \neq \emptyset$, just in case the machine has an element in its domain. Since this is undecidable, the non-emptiness of the intersection problem is also undecidable. This is an alternate proof to one we already showed based on PCP.
- Additionally, if $L1 \cap L2 = \emptyset$, the complement (bad traces + non-traces) is Σ^*. As this can be shown to be a CFL, determining if a CFG generates Σ^* is undecidable as well.

What’s Undecidable?

- We cannot decide if the set of valid terminating traces of an arbitrary machine **M** is non-empty.
- We cannot decide if the complement of the set of valid terminating traces of an arbitrary machine **M** is everything. In fact, this is not even semi-decidable.

What's a CSL or CFL?

- Given some machine **M** (I'll talk about specific models later)
  - The set of valid traces of **M** is Context Sensitive (can prove by fact that intersection of two CFLs is a CSG or by direct construction)
  - The complement of the valid traces of **M** is Context Free; that is, the set of invalid traces of **M** is Context Free (just one mistake required)
  - The set of valid terminating traces of **M** is Context Sensitive (same as above)
  - The complement of the valid terminating traces of **M** is Context Free; again, this requires just one mistake

$L = \Sigma^*?$

- If L is regular, then $L = \Sigma^*?$ is decidable
  - Easy – Reduce to minimal deterministic FSA, $\mathcal{A}_L$ accepting L. $L = \Sigma^*$ iff $\mathcal{A}_L$ is a one-state machine, whose only state is accepting
- If L is context free, then $L = \Sigma^*?$ is undecidable
  - Just produce the complement of a machine's valid terminating traces; if it's Σ^* then the original machine accepted nothing

Traces are NOT CFLs

- In the previous, we assumed that a trace is NOT a CFL, but we never proved that.
- To show the trace language for a TM, M,
  $\{ \# C_1 \# C_2 \# C_3 \# C_4 \ \ldots \# C_{k-1} \# C_k \# \mid$
  $k \geq 1$ and $C_i \Rightarrow_M C_{i+1}$, for $1 \leq i < k \}$ is not a CFL, we can focus on a simple machine that has just one non-blank $\{1\}$ and one state $\{q\}$ and the rules
  q 0 0 q
  q 1 1 q
- This machine has traces of the form
  $\{ \# C \# C \# C \# C \ \ldots \# C \# C \# \}$ as it never changes the tape contents or its state.

Using Pumping Lemma

- From previous slide, assume that language of traces,
  L = { # C # C # C # C … # C # C # },
  involving no changes in the ID is Context Free
- Pumping Lemma gives me an N>0
- I choose the valid trace in L that is # q 1^N # q 1^N # q 1^N #
- PL breaks this up into uvwxy, |vwx| ≤ N, |vx|>0 and
  $\forall i \geq 0\ uv^iwx^iy \in L$
- Case 1: vx contains some 1's. Due to fact that |vwx| ≤ N, the 1's can come from at most two consecutive sequences of 1's. If i=0, then we reduce 1's in at most two subsequences, but not in the third, leading to an imbalance, and so the result is not in L.
- Case 2: vx contains no 1's, then it must be either 'q', '#', or '#q'. In any case, if i=1 then we remove a state or a divider or both and the result is not a sequence of fixed configuration, so is not in L.
- By PL, L is not a CFL.

Language of Traces is a CSL

- The easiest way to show this for Turing machine traces is to describe an LBA that is given a string and wants to check if it is a valid trace.
- The LBA could make a pass over to be sure the string starts with a #, ends with a #, has no 0's immediately following a #, has a leading 0 immediately prior to a # only if the character preceding that 0 is a state, and has exactly one state between each pair of #'s.
- The LBA could then check each pair by copying the second member of a pair under the first (2 tracks) and then marching over the two one character at a time until a state is found in one or the other. It can then do checks that are based on the Turing machine rules with there being a need to look at only 4 characters in each track – state, character to immediate left of state and up to two characters to immediate right of state on each track (think about it). Of course, all parts of configuration that are not altered must be checked to be sure they match on both tracks.

Non-Traces is a CFL

- There are two ways that a string might not be a valid trace.
- First, it might be ill-formed, but we can easily check if a word looks like a trace. If not, it is in the complement of valid traces
- Second, we can check pairs of configurations, # C_i # C_{i+1} to see if there is a transcription error; that is, we can check to see if it is the case that C_{i+1} does not follow from C_i in a valid trace. This is a non-deterministic process where we "guess" which pair might be in error and then, if the guess is correct, we accept the string as a bad one that just looks like a trace.
- How hard is it to check for one bad transcription? Well, as noted above it starts with a guess, but then we must check. If it's a TM trace, we use alternating ID reversals, so such a pair is either # C_i # C_{i+1}^R or # C_i^R # C_{i+1}. Checking an error here is just looking as was described with the LBA single step check and can be done with a stack. What the stack cannot do is look at sequences longer than single pairs.

Traces of FRS with Residues

- I have chosen, once again to use the Factor Replacement Systems, but this time, Factor Systems with Residues.
  The rules are unordered and each is of the form
  $a\,x + b \rightarrow c\,x + d$
- These systems need to overcome the lack of ordering when simulating Register Machines. This is done by

j.	$INC_r[i]$	$p_{n+j}\ x$	$\rightarrow p_{n+i}\ p_r\ x$
j.	$DEC_r[s, f]$	$p_{n+j}\ p_r\ x$	$\rightarrow p_{n+s}\ x$
		$p_{n+j}\ p_r\ x + k\ p_{n+j}$	$\rightarrow p_{n+f}\ p_r\ x + k\ p_{n+f}$, $1 \le k < p_r$

  We also add the halting rule associated with m+1 of

$$p_{n+m+1}\ x \rightarrow 0$$

- Thus, halting is equivalent to producing 0. We can also add one more rule that guarantees we can reach 0 on both odd and even numbers of moves

$$0 \rightarrow 0$$

Intersection of CFLs

- Let (n, ((a1,b1,c1,d1) , … ,(ak,bk,ck,dk)) be some factor replacement system with residues. Define grammars G1 and G2 by using the 4k+2 rules

  G : $F_i \rightarrow 1^{ai}F_i1^{ci}$ | $1^{ai+bi}\#1^{ci+di}$ $1 \le i \le k$

  $S_1 \rightarrow \# F_i S_1$ | $\# F_i \#$ $1 \le i \le k$

  $S_2 \rightarrow \# 1^{x0}S_11^{z0}\#$ Z_0 is 0 for us

  G1 starts with S_1 and G2 with S_2
- Thus, using the notation of writing Y in place of 1^Y,

  L1 = L(G1) = { $\#Y_0 \# Y_1 \# Y_2 \# Y_3 \# \ldots \# Y_{2j} \# Y_{2j+1} \#$ }

  where $Y_{2i} \Rightarrow Y_{2i+1}$, $0 \le i \le j$.

  This checks the even/odd steps of an even length computation.

  But, L2 = L(G2) = { $\#X_0 \# X_1 \# X_2 \# X_3 \# X_4 \# \ldots \# X_{2k-1} \# X_{2k}\# Z_0 \#$ }

  where $X_{2i-1} \Rightarrow X_{2i}$, $1 \le i \le k$.

  This checks the odd/even steps of an even length computation.
- Given that the intersection of two CFLs is at worst a CSL, we now have an indirect way of showing that the valid terminating traces are a CSL.

Intersection Continued

Now, X_0 is chosen as some selected input value to the Factor System with Residues, and Z_0 is the unique value (0 in our case) on which the machine halts. But,

$L1 \cap L2 = \{\#X_0 \# X_1 \# X_2 \# X_3 \# X_4 \# \ldots \# X_{2k-1} \# X_{2k}\# Z_0 \# \}$

where $X_i \Rightarrow X_{i+1}$, $0 \le i < 2k$, and $X_{2k} \Rightarrow Z_0$. This checks <u>all</u> steps of an even length computation. But our original system halts if and only if it produces 0 (Z_0) in an even (also odd) number of steps. Thus the intersection is non-empty just in case the Factor System with residue eventually produces 0 when started on X_0, just in case the Register Machine halts when started on the register contents encoded by X_0.

This is an independent proof of the undecidability of the non-empty intersection problem for CFGs and the non-emptiness problem for CSGs.

What's a CSL or CFL?

- Given an FRS with Residue
  - The set of valid traces is Context Sensitive (can prove by fact that intersection of two CFLs is a CSG or by direct construction or by describing an LBA that accepts this language)
  - The complement of the valid traces is Context Free; that is, the set of invalid traces of M is Context Free (just one mistake required)
  - The set of valid terminating traces is Context Sensitive (same as above)
  - The complement of the valid terminating traces is Context Free; again, this requires just one mistake

Quotients of CFLs (concept)

Let L1 = L(G1) = { $ # Y_0 # Y_1 # Y_2 # Y_3 # … # Y_{2j} # Y_{2j+1} # }
where $Y_{2i} \Rightarrow Y_{2i+1}$, $0 \le i \le j$.
This checks the even/odd steps of an even length computation.

Now, let L2=L(G2)={X_0 $ # X_0 # X_1 # X_2 # X_3 # X_4 # … # X_{2k-1} # X_{2k}# Z_0 #}
where $X_{2i-1} \Rightarrow X_{2i}$, $1 \le i \le k$ and Z is a unique halting configuration.

This checks the odd/steps of an even length computation and includes an extra copy of the starting number prior to its $.

Now, consider the quotient of L2 / L1 . The only way a member of L1 can match a final substring in L2 is to line up the $ signs. But then they serve to check out the validity and termination of the computation. Moreover, the quotient leaves only the starting point (the one on which the machine halts.) Thus,

L2 / L1 = { X_0 | the system being traced halts}.

Since deciding the members of an re set is in general undecidable, we have shown that membership in the quotient of two CFLs is also undecidable.
Note: Intersection of two CFLs is a CSL but quotient of two CFLs is an re set and, in fact, all re sets can be specified by such quotients.

Quotients of CFLs (precise)

- Let (n, ((a1,b1,c1,d1) , … ,(ak,bk,ck,dk)) be some factor replacement system with residues. Define grammars G1 and G2 by using the 4k+4 rules

G : F_i → $1^{ai}F_i1^{ci}$ | $1^{ai+bi}\#1^{ci+di}$ $1 \le i \le k$

T_1 → # F_i T_1 | # F_i # $1 \le i \le k$

A → 1 A 1 | $ #

S_1 → $$T_1$

S_2 → A T_1 # 1^{z0} # Z_0 is 0 for us

G1 starts with S_1 and G2 with S_2

- Thus, using the notation of writing Y in place of 1^Y,

L1 = L(G1) = { $ #Y_0 # Y_1 # Y_2 # Y_3 # … # Y_{2j} # Y_{2j+1} # }
where $Y_{2i} \Rightarrow Y_{2i+1}$, $0 \le i \le j$.
This checks the even/odd steps of an even length computation.
But, L2 = L(G2) = { X $ #X_0 # X_1 # X_2 # X_3 # X_4 # … # X_{2k-1} # X_{2k}# Z_0 # }
where $X_{2i-1} \Rightarrow X_{2i}$, $1 \le i \le k$ and X = X_0
This checks the odd/steps of an even length computation, and includes an extra copy of the starting number prior to its $.

Summarizing Quotient

Now, consider the quotient L2 / L1 where L1 and L2 are the CFLs on prior slide. The only way a member of L1 can match a final substring in L2 is to line up the $ signs. But then they serve to check out the validity and termination of the computation. Moreover, the quotient leaves only the starting number (the one on which the machine halts.) Thus,

L2 / L1 = { X | the system F halts on zero }.

Since deciding the members of an re set is in general undecidable, we have shown that membership in the quotient of two CFLs is also undecidable.

Traces and Type 0

- Here, it is easier to show a simulation of a Turing machine than of an FRS.
- Assume we are given some machine M, with Turing table T (using Post notation). We assume a tape alphabet of Σ that includes a blank symbol B.
- Consider a starting configuration C0. Our rules will be

S	→	**# C0 #**	**where C0 = αq0aβ is initial ID**
q a	→	**s b**	**if q a b s ∈ T**
b q a x	→	**b a s x**	**if q a R s ∈ T, a,b,x ∈ Σ**
b q a #	→	**b a s B #**	**if q a R s ∈ T, a,b ∈ Σ**
# q a x	→	**# a s x**	**if q a R s ∈ T, a,x ∈ Σ, a≠B**
# q a #	→	**# a s B #**	**if q a R s ∈ T, a ∈ Σ, a≠B**
# q a x	→	**# s x #**	**if q a R s ∈ T, x ∈ Σ, a=B**
# q a #	→	**# s B #**	**if q a R s ∈ T, a=B**
b q a x	→	**s b a x**	**if q a L s ∈ T, a,b,x ∈ Σ**
# q a x	→	**# s B a x**	**if q a L s ∈ T, a,x ∈ Σ**
b q a #	→	**s b a #**	**if q a L s ∈ T, a,b ∈ Σ, a≠B**
# q a #	→	**# s B a #**	**if q a L s ∈ T, a ∈ Σ, a≠B**
b q a #	→	**s b #**	**if q a L s ∈ T, b ∈ Σ, a=B**
# q a #	→	**# s B #**	**if q a L s ∈ T, a=B**
f	→	**λ**	**if f is a final state**
#	→	**λ**	**just cleaning up the dirty linen**

CSG and Undecidability

- We can almost do anything with a CSG that can be done with a Type 0 grammar. The only thing lacking is the ability to reduce lengths, but we can throw in a character that we think of as meaning “deleted”. Let’s use the letter d as a deleted character and use the letter e to mark both ends of a word.
- Let G = (V, T, P , S) be an arbitrary Type 0 grammar.
- Define the CSG G’ = (V $\cup$ {S’, D}, T $\cup$ {d, e}, S’, P’), where P’ is

S’	$\rightarrow$	**e S e**	
D x	$\rightarrow$	**x D**	**when x $\in$ V $\cup$ T**
D e	$\rightarrow$	**e d**	**push the delete characters to far right**
α	$\rightarrow$	β	**where $\alpha \rightarrow \beta \in$ P and $\lvert\alpha\rvert \le \lvert\beta\rvert$**
α	$\rightarrow$	βD^k	**where $\alpha \rightarrow \beta \in$ P and $\lvert\alpha\rvert - \lvert\beta\rvert = k > 0$**

- Clearly, L(G’) = { e w e d^m \| w $\in$ L(G) and m≥0 is some integer }
- For each w $\in$ L(G), we cannot, in general, determine for which values of m, e w e $d^m \in$ L(G’). We would need to ask a potentially infinite number of questions of the form
  “does e w e $d^m \in$ L(G’)” for some m≥0 to determine if w $\in$ L(G).
  That’s a semi-decision procedure because m can be unbounded above.

Some Consequences

- CSGs are not closed under Init, Final, Mid, quotient with regular sets, substitution and homomorphism (okay for λ-free homomorphism and non-length reducing substitutions)
- We also have that the emptiness problem is undecidable from this result. That gives us two proofs of this one result.
- For Type 0, emptiness and even the membership problems are undecidable.

Summary of Grammar Results

Decidability

- Everything about regular
- Membership in CFLs and CSLs
  - CKY for CFLs
- Emptiness for CFLs

Undecidability

- Is $L = \varnothing$, for CSL, L?
- Is $L=\Sigma^*$, for CFL (CSL), L?
- Is $L_1=L_2$ for CFLs (CSLs), L_1, L_2?
- Is $L_1 \subseteq L_2$ for CFLs (CSLs), L_1, L_2?
- Is $L_1 \cap L_2=\varnothing$ for CFLs (CSLs), L_1, L_2?

More Undecidability

- Is CFL, L, ambiguous?
- Is $L=L^2$, L a CFL?
- Does there exist a finite n, $L^n=L^{N+1}$?
- Is L_1/L_2 finite, L_1 and L_2 CFLs?
- Membership in L_1/L_2, where L_1 and L_2 are CFLs?

Word to Grammar Problem

- Recast semi-Thue system making all symbols non-terminal, adding S and V to non-terminals and terminal set $\Sigma=\{a\}$

  G: $S \to h1^x q_1 0h$

  $hq_0h \to V$

  $V \to aV$

  $V \to \lambda$

- $x \in \mathcal{L}(M)$ iff $\mathcal{L}(G) \neq \emptyset$ iff $\mathcal{L}(G)$ infinite iff $\lambda \in \mathcal{L}(G)$ iff $a \in \mathcal{L}(G)$ iff $\mathcal{L}(G) = \Sigma^*$

Consequences for PSG

- Unsolvables
  - $\mathcal{L}(G) = \varnothing$
  - $\mathcal{L}(G) = \Sigma^*$
  - $\mathcal{L}(G)$ infinite
  - $w \in \mathcal{L}(G)$, for arbitrary w
  - $\mathcal{L}(G) \supseteq \mathcal{L}(G2)$
  - $\mathcal{L}(G) = \mathcal{L}(G2)$
- Latter two results follow when have
  - G2: $S \to aS \mid \lambda \quad a \in \Sigma$

Finite Convergence for Concatenation of Context-Free Languages

Relation to Real-Time
(Constant Time) Execution

Powers of CFLs

Let G be a context free grammar.

Consider $L(G)^n$

Question1: Is $L(G) = L(G)^2$?

Question2: Is $L(G)^n = L(G)^{n+1}$, for some finite $n>0$?

These questions are both undecidable.

Think about why question1 is as hard as whether or not L(G) is Σ^*.

Question2 requires much more thought.

$L(G) = L(G)^2$?

- **The problem to determine if $L = \Sigma^*$ is Turing reducible to the problem to decide if $L \bullet L \subseteq L$, so long as L is selected from a class of languages C over the alphabet Σ for which we can decide if $\Sigma \cup \{\lambda\} \subseteq L$.**
- **Corollary 1:**
  The problem "is $L \bullet L = L$, for L context free or context sensitive?" is undecidable

$L(G) = L(G)^2$? is undecidable

- Question: Does $L \bullet L$ get us anything new?
  - i.e., Is $L \bullet L = L$?
- Membership in a CFL is decidable.
- Claim is that $L = \Sigma^*$ iff

  (1) $\Sigma \cup \{\lambda\} \subseteq L$; and

  (2) $L \bullet L = L$
- Clearly, if $L = \Sigma^*$ then (1) and (2) trivially hold.
- Conversely, we have $\Sigma^* \subseteq L^* = \cup_{n\geq 0} L^n \subseteq L$
  - first inclusion follows from (1); second from (2)

Finite Power Problem

- The problem to determine, for an arbitrary context free language L, if there exist a finite n such that $L^n = L^{n+1}$ is undecidable.
- $L_1 = \{ C_1\# C_2^R \$ \mid$
  C_1, C_2 are configurations $\}$,
- $L_2 = \{ C_1\#C_2^R\$C_3\#C_4^R \ \dots \ \$C_{2k-1}\#C_{2k}^R\$ \mid$ where $k \geq 1$ and, for some i, $1 \leq i < 2k$, $C_i \Rightarrow_M C_{i+1}$ is false $\}$,
- $L = L_1 \cup L_2 \cup \{\lambda\}$.

Undecidability of $\exists n\ L^n = L^{n+1}$

- **L is context free.**
- **Any product of L_1 and L_2, which contains L_2 at least once, is L_2. For instance, $L_1 \bullet L_2 = L_2 \bullet L_1 = L_2 \bullet L_2 = L_2$.**
- **This shows that $(L_1 \cup L_2)^n = L_1^n \cup L_2$.**
- **Thus, $L^n = \{\lambda\} \cup L_1 \cup L_1^2 \ldots \cup L_1^n \cup L_2$.**
- **Analyzing L_1 and L_2 we see that $L_1^n \cup L_2 \neq L_2$ just in case there is a word $C_1 \# C_2^R \$ C_3 \# C_4^R \ldots \$ C_{2n-1} \# C_{2n}^R \$$ in L_1^n that is not also in L_2.**
- **But then there is some valid trace of length 2n.**
- **L has the finite power property iff M executes in constant time.**

Missing Step

- We have that **CT** (Constant-Time) is many-one reducible to Finite Power Problem (**FPC**) for CFLs
- This means that if **CT** is unsolvable, so is **FPC** for CFLs.
- However, we still lack a proof that **CT** is unsolvable. I am keeping that open as one of the problems that you folks can attack in your presentation. It takes two papers to get here. I'll document that.

Propositional Calculus

Axiomatizable Fragments

Propositional Calculus

- Mathematical of unquantified logical expressions
- Essentially Boolean algebra
- Goal is to reason about propositions
- Often interested in determining
  - Is a well-formed formula (wff) a tautology?
  - Is a wff refutable (unsatisfiable)?
  - Is a wff satisfiable? (classic NP-complete)

Tautology and Satisfiability

- The classic approaches are:
  - Truth Table
  - Axiomatic System (axioms and inferences)
- Truth Table
  - Clearly exponential in number of variables
- Axiomatic Systems Rules of Inference
  - Substitution and Modus Ponens
  - Resolution / Unification

Proving Consequences

- Start with a set of axioms (all tautologies)
- Using substitution and MP
  $(P, P \supset Q \Rightarrow Q)$
  derive consequences of axioms (also tautologies, but just a fragment of all)
- Can create complete sets of axioms
- Need 3 variables for associativity, e.g.,
  $(p1 \vee p2) \vee p3 \quad \supset \quad p1 \vee (p2 \vee p3)$

Some Undecidables

- Given a set of axioms,
  - Is this set complete?
  - Given a tautology T, is T a consequent?
- The above are even undecidable with one axiom and with only 2 variables. I will show this result shortly.

Refutation

- If we wish to prove that some wff, F, is a tautology, we could negate it and try to prove that the new formula is refutable (cannot be satisfied; contains a logical contradiction).
- This is often done using resolution.

Resolution

- Put formula in Conjunctive Normal Form (CNF)
- If have terms of conjunction
  $(P \vee Q)$, $(R \vee \sim Q)$
  then can determine that $(P \vee R)$
- If we ever get a null conclusion, we have refuted the proposition
- Resolution is not complete for derivation, but it is for refutation

Axioms

- Must be tautologies
- Can be incomplete
- Might have limitations on them and on WFFs, e.g.,
  - Just implication
  - Only n variables
  - Single axiom

Simulating Machines

- Linear representations require associativity, unless all operations can be performed on prefix only (or suffix only)
- Prefix and suffix-based operations are single stacks and limit us to CFLs
- Can simulate Post normal Forms with just 3 variables.

Diadic PIPC

- Diadic limits us to two variables
- PIPC means Partial Implicational Propositional Calculus, and limits us to implication as only connective
- Partial just means we get a fragment
- Problems
  - Is fragment complete?
  - Can F be derived by substitution and MP?

Living without Associativity

- Consider a two-stack model of a TM
- Could somehow use one variable for left stack and other for right
- Must find a way to encode a sequence as a composition of forms – that's the key to this simulation

Composition Encoding

- Consider $(p \supset p)$, $(p \supset (p \supset p))$, $(p \supset (p \supset (p \supset p)))$, …
  - No form is a substitution instance of any of the other, so they can't be confused
  - All are tautologies
- Consider $((X \supset Y) \supset Y)$
  - This is just $X \vee Y$

Encoding

- Use $(p \supset p)$ as form of bottom of stack
- Use $(p \supset (p \supset p))$ as form for letter 0
- Use $(p \supset (p \supset (p \supset p)))$ as form for 1
- Etc.
- String 01 (reading top to bottom of stack) is
  - $(\; ((p \supset p) \supset ((p \supset p) \supset ((p \supset p) \supset (p \supset p)))) \supset$
    $(((p \supset p) \supset ((p \supset p) \supset ((p \supset p) \supset (p \supset p)))) \supset$
    $((p \supset p) \supset ((p \supset p) \supset ((p \supset p) \supset (p \supset p))))))$

Encoding

$\mathcal{I}(p)$ abbreviates $[p \supset p]$

$\Phi_0(p)$ is $[p \supset \mathcal{I}(p)]$ which is $[p \supset [p \supset p]]$

$\Phi_1(p)$ is $[p \supset \Phi_0(p)]$

$\xi_1(p)$ is $[p \supset \Phi_1(p)]$

$\xi_2(p)$ is $[p \supset \xi_1 (p)]$

$\xi_3(p)$ is $[p \supset \xi_2 (p)]$

$\psi_1(p)$ is $[p \supset \xi_3 (p)]$

$\psi_2(p)$ is $[p \supset \psi_1 (p)]$

…

$\psi_m(p)$ is $[p \supset \psi_{m-1} (p)]$

Creating Terminal IDs

1. $[\xi_1 I(p_1) \vee I(p_1)]$.
2. $[\xi_1 I(p_1) \vee I(p_1)] \supset [\xi_1 I(p_1) \vee \Phi_1 I(p_1)]$.
3. $[\xi_1 I(p_1) \vee \Phi_i(p_2)] \supset [\xi_1 I(p_1) \vee \Phi_j \Phi_i(p_2)]$, $\forall i, j \in \{0, 1\}$.
4. $[\xi_1 I(p_1) \vee p_2] \supset [\xi_2 \Phi_1 I(p_1) \vee p_2]$.
5. $[\xi_1 I(p_1) \vee p_2] \supset [\xi_3 \Phi_i I(p_1) \vee p_2]$, $\forall i \in \{0, 1\}$.
6. $[\xi_2 \Phi_i(p_1) \vee p_2] \supset [\xi_2 \Phi_j \Phi_i(p_1) \vee p_2]$, $\forall i, j \in \{0, 1\}$.
7. $[\xi_2 \Phi_i(p_1) \vee p_2] \supset [\xi_3 \Phi_j \Phi_i(p_1) \vee p_2]$, $\forall i, j \in \{0, 1\}$.
8. $[\xi_3 \Phi_i(p_1) \vee p_2] \supset [\Psi_k \Phi_i(p_1) \vee p_2]$, whenever $q_k a_i$ is a terminal discriminant of M.

Reversing Print and Left

9. $[\Psi_k\Phi_i(p_1) \vee p_2] \supset [\Psi_h\Phi_j(p_1) \vee p_2]$, whenever $q_h a_j a_i q_k \in T$.

10a. $[\Psi_k\Phi_0 I(p_1) \vee I(p_1)] \supset [\Psi_h\Phi_0 I(p_1) \vee I(p_1)]$,

b. $[\Psi_k\Phi_1 I(p_1) \vee I(p_1)] \supset [\Psi_h\Phi_0 I(p_1) \vee \Phi_1(p_1)]$,

c. $[\Psi_k\Phi_i I(p_1) \vee \Phi_j(p_2)] \supset [\Psi_h\Phi_0 I(p_1) \vee \Phi_i\Phi_j(p_2)]$,

d. $[\Psi_k\Phi_0\Phi_0\Phi_i(p_1) \vee I(p_2)] \supset [\Psi_h\Phi_0\Phi_i(p_1) \vee I(p_2)]$,

e. $[\Psi_k\Phi_1\Phi_0\Phi_i(p_1) \vee I(p_2)] \supset [\Psi_h\Phi_0\Phi_i(p_1) \vee \Phi_1 I(p_2)]$,

f. $[\Psi_k\Phi_i\Phi_0\Phi_j(p_1) \vee \Phi_m(p_2)] \supset [\Psi_h\Phi_0\Phi_j(p_1) \vee \Phi_i\Phi_m(p_2)]$,

$\forall i, j, m \in \{0, 1\}$ whenever $q_h 0 L q_k \in T$.

11a. $[\Psi_k\Phi_0\Phi_1(p_1) \vee I(p_2)] \supset [\Psi_h\Phi_1(p_1) \vee I(p_2)]$,

b. $[\Psi_k\Phi_1\Phi_1(p_1) \vee I(p_2)] \supset [\Psi_h\Phi_1(p_1) \vee \Phi_1 I(p_2)]$,

c. $[\Psi_k\Phi_i\Phi_1(p_1) \vee \Phi_j(p_2)] \supset [\Psi_h\Phi_1(p_1) \vee \Phi_i\Phi_j(p_2)]$,

$\forall i, j \in \{0, 1\}$ whenever $q_k 1 L q_k \in T$.

Reversing Right

12a. $[\Psi'_k \Phi_0 I(p_1) \vee I(p_1)] \supset [\Psi'_h \Phi_0 I(p_1) \vee I(p_1)]$,
 b. $[\Psi'_k \Phi_0 I(p_1) \vee \Phi_0 \Phi_i(p_2)] \supset [\Psi'_h \Phi_0 I(p_1) \vee \Phi_i(p_2)]$,
 c. $[\Psi'_k \Phi_1(p_1) \vee I(p_2)] \supset [\Psi'_h \Phi_0 \Phi_1(p_1) \vee I(p_2)]$,
 d. $[\Psi'_k \Phi_0 \Phi_i(p_1) \vee I(p_2)] \supset [\Psi'_h \Phi_0 \Phi_0 \Phi_i(p_1) \vee I(p_2)]$,
 e. $[\Psi'_k \Phi_0 \Phi_i(p_1) \vee \Phi_0 \Phi_j(p_2)] \supset [\Psi'_h \Phi_0 \Phi_0 \Phi_i(p_1) \vee \Phi_j(p_2)]$,
 f. $[\Psi'_k \Phi_1(p_1) \vee \Phi_0 \Phi_i(p_2)] \supset [\Psi'_h \Phi_0 \Phi_1(p_1) \vee \Phi_i(p_2)]$,
 $\forall i, j \in \{0, 1\}$ whenever $q_h 0 R q_k \in T$.

13a. $[\Psi'_k \Phi_0 I(p_1) \vee \Phi_1(p_2)] \supset [\Psi'_h \Phi_1 I(p_1) \vee p_2]$,
 b. $[\Psi'_k \Phi_1(p_1) \vee \Phi_1(p_2)] \supset [\Psi'_h \Phi_1 \Phi_1(p_1) \vee p_2]$,
 c. $[\Psi'_k \Phi_0 \Phi_i(p_1) \vee \Phi_1(p_2)] \supset [\Psi'_h \Phi_1 \Phi_0 \Phi_i(p_1) \vee p_2]$
 $\forall i \in \{0, 1\}$ whenever $q_h 1 R q_k \in T$.

Exam Prep

Sample Question

Let **A** and **B** be re sets. For each of the following, either prove that the set is re, or give a counterexample that results in some known non-re set.

Let A be semi decided by f_A and B by f_B

a) $A \cup B$: must be re as it is semi-decided by

$f_{A \cup B}(x) = \exists t\, [stp(f_A, x, t) \,||\, stp(f_B, x, t)\,]$

b) $A \cap B$: must be re as it is semi-decided by

$f_{A \cap B}(x) = \exists t\, [stp(f_A, x, t) \,\&\&\, stp(f_B, x, t)\,]$

c) ~A: can be non-re. If ~A is always re, then all re are recursive as any set that is re and whose complement is re is decidable. However, A = K is a non-rec, re set and so ~A is not re.

Sample Question

Given that the predicate **STP** and the function **VALUE** are prf's, show that we can semi-decide

{ f | φ_f evaluates to 0 for some input}

This can be shown re by the predicate

{f | $\exists$<x,t> [stp(f,x,t) && value(f,x,t) = 0] }

Sample Question

Let **S** be an re (recursively enumerable), non-recursive set, and **T** be re, non-empty, possibly recursive set.
Let **E = { z | z = x + y, where x $\in$ S and y $\in$ T }.**

(a) Can **E** be non re? **No as we can let S and T be semi-decided by f_S and f_T, resp., E is then semi-dec. by**
$f_E(z) = \exists \langle x,y,t \rangle$ [stp(f_S, x, t) && stp(f_T, y, t) && (z = value(f_S, x, t) + value(f_T, y, t))]

(b) Can **E** be re non-recursive? **Yes, just let T = {0}, then E = S which is known to be re, non-rec.**

(c) Can **E** be recursive? **Yes, let T = $\aleph$, then E = { x | x ≥ min (S) } which is a co-finite set and hence rec.**

Sample Question

Assuming **TOTAL** is undecidable, use reduction to show the undecidability of

Incr = { f | $\forall x\ \varphi_f(x+1) > \varphi_f(x)$ }

Let f be arb.

Define $G_f(x) = \varphi_f(x) - \varphi_f(x) + x$

$f \in$ TOTAL iff $\forall x \varphi_f(x)\downarrow$ iff $\forall x\ G_f(x)\downarrow$ iff

$\forall x\ \varphi_f(x) - \varphi_f(x) + x = x$ iff $G_f \in$ Incr

Sample Question

Let **Incr = { f | ∀x, $\varphi_f(x+1) > \varphi_f(x)$ }**.
Let **TOTAL = { f | ∀x, $\varphi_f(x)\downarrow$ }**.
Prove that **Incr $\leq_m$ TOTAL**.

Let f be arb.

Define $G_f(x) = \exists t[stp(f,x,t)$ && $stp(f,x+1,t)$ && $(value(f,x+1,t) > value(f,x,t))]$

f $\in$ Incr iff ∀x $\varphi_f(x+1) > \varphi_f(x)$ iff ∀x $G_f(x)\downarrow$ iff $G_f \in$ TOT

Sample Question

Let **Incr = { f | $\forall$x φ_f(x+1)>φ_f(x) }**.
Use Rice's theorem to show **Incr** is not recursive.

Non-Trivial as
C_0(x)=0 $\notin$ Incr; S(x)=x+1 $\in$ Incr
Let f,g be arb. Such that $\forall$x φ_f(x)=φ_g(x)
f $\in$ Incr iff $\forall$x φ_f(x+1)>φ_f(x) iff
$\forall$x φ_g(x+1)>φ_g(x) iff g $\in$ Incr

Sample Question

Let **S** be a recursive (decidable set), what can we say about the complexity (recursive, re non-recursive, non-re) of **T**, where **T** $\subset$ **S**?

> **Nothing. Just let S = $\aleph$, then T could be any subset of $\aleph$. There are an uncountable number of such subsets and some are clearly in each of the categories above.**

Sample Question

Let **$P = \{ f \mid \exists x [STP(f, x, x)] \}$**. Why does Rice's theorem not tell us anything about the undecidability of **P**?

> **This is not an I/O property as we can have implementations of C_0 that are efficient and satisfy P and others that do not.**

ORDER ANALYSIS

Computability & Complexity Theory

Charles E. Hughes

COT 6410 – Spring 2019

Notes

Notion of “Order”

Throughout the complexity portion of this course, we will be interested in how long an algorithm takes on the instances of some arbitrary "size" **n**. Recognizing that different times can be recorded for two instance of size **n**, we only ask about the worst case.

We also understand that different languages, computers, and even skill of the implementer can alter the "running time."

Notion of “Order”

As a result, we really can never know "exactly" how long anything takes.

So, we usually settle for a substitute function, and say the function we are trying to measure is "of the order of" this new substitute function.

Notion of "Order"

"Order" is something we use to describe an upper bound upon something else (in our case, time, but it can apply to almost anything).

For example, let **f(n)** and **g(n)** be two functions. We say "**f(n)** is order **g(n)**" when there exists constants **c** and **N** such that **f(n) ≤ cg(n)** for all **n ≥ N**.

What this is saying is that when **n** is 'large enough,' **f(n)** is bounded above by a constant multiple of **g(n)**.

Notion of "Order"

This is particularly useful when **f(n)** is not known precisely, is complicated to compute, and/or difficult to use. We can, by this, replace **f(n)** by **g(n)** and know we aren't "off too far."

We say **f(n)** is "in the order of **g(n)**" or, simply, **f(n)** $\in$ **O(g(n))**.

Usually, **g(n)** is a simple function, like **nlog(n)**, $\mathbf{n^3}$, $\mathbf{2^n}$, etc., that's easy to understand and use.

Notion of “Order”

Order of an Algorithm: The maximum number of steps required to find the answer to any instance of size **n**, for any arbitrary value of **n**.

For example, if an algorithm requires at most $\mathbf{6n^2+3n-6}$ steps on any instance of size n, we say it is "order $\mathbf{n^2}$" or, simply, $\mathbf{O(n^2)}$.

Order

Let the order of algorithm **X** be in $\mathbf{O(f_x(n))}$.

Then, for algorithms **A** and **B** and their respective order functions, $\mathbf{f_A(n)}$ and $\mathbf{f_B(n)}$, consider the limit of $\mathbf{f_A(n)/f_B(n)}$ as **n** goes to infinity.

If this value is

0	**A** is faster than **B**
constant	**A** and **B** are "equally slow/fast"
infinity	**A** is slower than **B**.

Order of a Problem

Order of a Problem

The order of the fastest algorithm that can ever solve this problem. (Also known as the "Complexity" of the problem.)

Often difficult to determine, since this allows for algorithms not yet discovered.

Decision vs Optimization

Two types of problems are of particular interest:

Decision Problems ("Yes/No" answers)

Optimization problems ("best" answers)

(there are other types)

Vertex Cover (VC)

- Suppose we are in charge of a large network (a graph where edges are links between pairs of cities (vertices). Periodically, a line fails. To mend the line, we must call in a repair crew that goes over the line to fix it. To minimize down time, we station a repair crew at one end of every line. How many crews must you have and where should they be stationed?

- This is called the Vertex Cover Problem. (Yes, it sounds like it should be called the Edge Cover problem – something else already had that name.)

- An interesting problem – it is among the hardest problems, yet is one of the easiest of the hard problems.

VC Decision vs Optimization

- As a Decision Problem:

- Instances: A graph **G** and an integer **k**.
- Question: Does **G** possess a vertex Cover with at most **k** vertices?

- As an Optimization Problem:

- Instances: A graph **G**.
- Question: What is the smallest **k** for which **G** possesses a vertex cover?

Relation of VC Problems

- If we can (easily) solve either one of these problems, we can (easily) solve the other. (To solve the optimization version, just solve the decision version with several different values of **k**. Use a binary search on **k** between **1** and **n**. That is **log(n)** solutions of the decision problem solves the optimization problem. It's simple to solve the decision version if we can solve the optimization version.

- We say their time complexity differs by no more than a multiple of **log(n)**.
- If one is polynomial then so is the other.
- If one is exponential, then so is the other.

- We say they are equally difficult (both poly. or both exponential).

Smallest VC

- A "stranger version"

- Instances: A graph **G** and an integer **k**.
- Question: Does the smallest vertex cover of **G** have exactly **k** vertices?
- This is a decision problem. But, notice that it does not seem to be easy to verify either Yes or No instances!! (We can easily verify No instances for which the VC number is less than **k**, but not when it is actually greater than **k**.)

- So, it would seem to be in a different category than either of the other two. Yet, it also has the property that if we can easily solve either of the first two versions, we can easily solve this one.

Natural Pairs of Problems

Interestingly, these usually come in pairs

a *decision* problem, and

an *optimization* problem.

Equally easy, or equally difficult, to solve.

Both can be solved in polynomial time, or both require exponential time.

A Word about Time

An algorithm for a problem is said to be polynomial if there exists integers **k** and **N** such that **t(n)**, the maximum number of steps required on any instance of size **n**, is at most $\mathbf{n^k}$, for all $\mathbf{n \geq N}$.

Otherwise, we say the algorithm is exponential. Usually, this is interpreted to mean $\mathbf{t(n) \geq c^n}$ for an infinite set of size **n** instances, and some constant $\mathbf{c > 1}$ (often, we simply use $\mathbf{c = 2}$).

A Word about “Words”

Normally, when we say a problem is "easy" we mean that it has a polynomial algorithm.

But, when we say a problem is "hard" or “apparently hard" we usually mean no polynomial algorithm is known, and none seems likely.

It is possible a polynomial algorithm exists for "hard" problems, but the evidence seems to indicate otherwise.

A Word about Abstractions

Problems we will discuss are usually "abstractions" of real problems. That is, to the extent possible, non-essential features have been removed, others have been simplified and given variable names, relationships have been replaced with mathematical equations and/or inequalities, etc.

If an abstraction is hard, then the real problem is probably even harder!!

A Word about Toy Problems

This process, Mathematical Modeling, is a field of study in itself, and not our interest here.

On the other hand, we sometimes conjure up artificial problems to put a little "reality" into our work. This results in what some call "toy problems."

Again, if a toy problem is hard, then the real problem is probably harder.

Very Hard Problems

Some problems have no algorithm (e. g., Halting Problem.)

<u>No</u> mechanical/logical procedure will ever solve all instances of any such problem!!

Some problems have only exponential algorithms (provably so – they must take at least order $\mathbf{2^n}$ steps) So far, only a few have been proven, but there may be many. We suspect so.

Easy Problems

Many problems have polynomial algorithms (Fortunately).

Why fortunately? Because, most exponential algorithms are essentially useless for problem instances with **n** much larger than 50 or 60. We have algorithms for them, but the best of these will take 100's of years to run, even on much faster computers than we now envision.

Three Classes of Problems

Problems proven to be in these three groups (classes) are, respectively,

Undecidable, Exponential, and Polynomial.

Theoretically, all problems belong to exactly one of these three classes.

Unknown Complexity

Practically, there are a lot of problems (maybe, most) that <u>have not</u> been proven to be in any of the classes (Yet, maybe never will be).

Most currently "lie between" polynomial and exponential – we know of exponential algorithms, but have been unable to prove that exponential algorithms are necessary.

Some may have polynomial algorithms, but we have not yet been clever enough to discover them.

Why do we Care?

If an algorithm is $\mathbf{O(n^k)}$, increasing the size of an instance by one gives a running time that is $\mathbf{O((n+1)^k)}$

That's really not much more.

With an increase of one in an exponential algorithm, $\mathbf{O(2^n)}$ changes to $\mathbf{O(2^{n+1}) = O(2*2^n) = 2*O(2^n)}$ – that is, it takes about twice as long.

A Word about "Size"

Technically, the size of an instance is the minimum number of bits (information) needed to represent the instance – its "length."

This comes from early Formal Language researchers who were analyzing the time needed to 'recognize' a string of characters as a function of its length (number of characters).

When dealing with more general problems there is usually a parameter (number of vertices, processors, variables, etc.) that is polynomially related to the length of the instance. Then, we are justified in using the parameter as a measure of the length (size), since anything polynomially related to one will be polynomially related to the other.

The Subtlety of "Size"

But, be careful.

For instance, if the "value" (magnitude) of n is both the input and the parameter, the 'length' of the input (number of bits) is $\log_2(n)$. So, an algorithm that takes n time is running in $n = 2^{\log2(n)}$ time, which is exponential in terms of the length, $\log_2(n)$, but linear (hence, polynomial) in terms of the "value," or magnitude, of n.

It's a subtle, and usually unimportant difference, but it can bite you.

Subset Sum

- Problem – Subset Sum
- Instances: A list **L** of **n** integer values and an integer **B**.
- Question: Does **L** have a subset which sums exactly to **B**?
- No one knows of a polynomial (deterministic) solution to this problem.
- On the other hand, there is a very simple (dynamic programming) algorithm that runs in **O(nB)** time.
- Why isn't this "polynomial"?
- Because, the "length" of an instance is **nlog(B)** and
- **nB > (nlog(B))^k** for any fixed **k**.

Why do we Care?

When given a new problem to solve (design an algorithm for), if it's undecidable, or even exponential, you will waste a lot of time trying to write a polynomial solution for it!!

If the problem really is polynomial, it will be worthwhile spending some time and effort to find a polynomial solution.

You should know something about how hard a problem is before you try to solve it.

Research Territory

Decidable – vs – Undecidable

(area of Computability Theory)

Exponential – vs – polynomial

(area of Computational Complexity)

Algorithms for any of these

(area of Algorithm Design/Analysis)

Complexity Theory

Second Part of Course

Models of Computation

NonDeterminism

Since we can't seem to find a model of computation that is more powerful than a TM, can we find one that is 'faster'?

In particular, we want one that takes us from exponential time to polynomial time.

Our candidate will be the NonDeterministic Turing Machine (NDTM).

NDTM's

In the basic Deterministic Turing Machine (DTM) we make one major alteration (and take care of a few repercussions):

> The 'transition functon' in DTM's is allowed to become a 'transition mapping' in NDTM's.

This means that rather than the next action being totally specified (deterministic) by the current state and input character, we now can have many next actions - simultaneously. That is, a NDTM can be in many states at once. (That raises some interesting problems with writing on the tape, just where the tape head is, etc., but those little things can be explained away).

NDTM's

We also require that there be only one halt state - the 'accept' state. That also raises an interesting question - what if we give it an instance that is not 'acceptable'? The answer - it blows up (or goes into an infinite loop).

The solution is that we are only allowed to give it 'acceptable' input. That means

> **NDTM's are only defined for decision problems**
>
> **and, in particular, only for Yes instances.**

NDTM's

We want to determine how long it takes to get to the accept state - that's our only motive!!

So, what is a NDTM doing?

In a normal (deterministic) algorithm, we often have a loop where each time through the loop we are testing a different option to see if that "choice" leads to a correct solution. If one does, fine, we go on to another part of the problem. If one doesn't, we return to the same place and make a different choice, and test it, etc.

NDTM's

If this is a Yes instance, we are guaranteed that an acceptable choice will eventually be found and we go on.

In a NDTM, what we are doing is making, and testing, all of those choices at once by 'spawning' a different NDTM for each of them. Those that don't work out, simply die (or something).

This is kind of like the ultimate in parallel programming.

NDTM's

To allay concerns about not being able to write on the tape, we can allow each spawned NDTM to have its own copy of the tape with a read/write head.

The restriction is that nothing can be reported back except that the accept state was reached.

NDTM's

Another interpretation of nondeterminism:

From the basic definition, we notice that out of every state having a nondeterministic choice, at least one choice is valid and all the rest sort of die off. That is they really have no reason for being spawned (for this instance - maybe for another). So, we station at each such state, an 'oracle' (an all knowing being) who only allows the correct NDTM to be spawned.

An 'Oracle Machine.'

NDTM's

This is not totally unreasonable. We can look at a non deterministic decision as a deterministic algorithm in which, when an "option" is to be tested, it is very lucky, or clever, to make the correct choice the first time.

In this sense, the two machines would work identically, and we are just asking "How long does a DTM take if it always makes the correct decisions?"

NDTM's

As long as we are talking magic, we might as well talk about a 'super' oracle stationed at the start state (and get rid of the rest of the oracles) whose task is to examine the given instance and simply tell you what sequence of transitions needs to be executed to reach the accept state.

He/she will write them to the left of cell 0 (the instance is to the right).

NDTM's

Now, you simply write a DTM to run back and forth between the left of the tape to get the 'next action' and then go back to the right half to examine the NDTM and instance to verify that the provided transition is a valid next action. As predicted by the oracle, the DTM will see that the NDTM would reach the accept state and can report the number of steps required.

NDTM's

All of this was originally designed with Language Recognition problems in mind. It is not a far stretch to realize the Yes instances of any of our more real word-like decision problems defines a language, and that the same approach can be used to "solve" them.

Rather than the oracle placing the sequence of transitions on the tape, we ask him/her to provide a 'witness' to (a 'proof' of) the correctness of the instance.

NDTM's

For example, in the SubsetSum problem, we ask the oracle to write down the subset of objects whose sum is B (the desired sum). Then we ask "Can we write a deterministic polynomial algorithm to test the given witness."

The answer for SubsetSum is Yes, we can, i.e., the witness is verifiable in deterministic polynomial time.

NDTM's - Witnesses

Just what can we ask and expect of a "witness"?

The witness must be something that

(1) we can verify to be accurate (for the given problem and instance) and

(2) we must be able to "finish off" the solution.

All in polynomial time.

NDTM's - Witnesses

The witness can be nothing!

> Then, we are on our own. We have to "solve the instance in polynomial time."

The witness can be "Yes."

> Duh. We already knew that. We have to now verify the yes instance is a yes instance (same as above).

The witness has to be something other than nothing and Yes.

NDTM's - Witnesses

The information provided must be something we could have come up with ourselves, but probably at an exponential cost. And, it has to be enough so that we can conclude the final answer Yes from it.

Consider a witness for the graph coloring problem:

Given: A graph G = (V, E) and an integer k.

Question: Can the vertices of G be assigned colors so that adjacent vertices have different colors and use at most k colors?

NDTM's - Witnesses

The witness could be nothing, or Yes.

> But that's not good enough - we don't know of a polynomial algorithm for graph coloring.

It could be "vertex 10 is colored Red."

> That's not good enough either. Any single vertex can be colored any color we want.

It could be a color assigned to each vertex.

> That would work, because we can verify its validity in polynomial time, and we can conclude the correct answer of Yes.

NDTM's - Witnesses

What if it was a color for all vertices but one?

> **That also is enough. We can verify the correctness of the n-1 given to us, then we can verify that the one uncolored vertex can be colored with a color not on any neighbor, and that the total is not more than k.**

What if all but 2, 3, or 20 vertices are colored

> **All are valid witnesses.**

What if half the vertices are colored?

> **Usually, No. There's not enough information. Sure, we can check that what is given to us is properly colored, but we don't know how to "finish it off."**

NDTM's - Witnesses

An interesting question: For a given problem, what are the limits to what can be provided that still allows a polynomial verification?

NDTM's

A major question remains: Do we have, in NDTMs, a model of computation that solves all deterministic exponential (DE) problems in polynomial time (nondeterministic polynomial time)??

It definitely solves some problems we *think* are DE in nondeterministic polynomial time.

NDTM's

But, so far, all problems that have been <u>proven</u> to require deterministic exponential time also require nondeterministic exponential time.

So, the jury is still out. In the meantime, NDTMs are still valuable, because they might identify a larger class of problems than does a deterministic TM - the set of decision problems for which Yes instances can be verified in polynomial time.

Problem Classes

We now begin to discuss several different classes of problems. The first two will be:

NP	'Nondeterministic' Polynomial
P	'Deterministic' Polynomial,
	The 'easiest' problems in NP

Their definitions are rooted in the depths of Computability Theory as just described, but it is worth repeating some of it in the next few slides.

Problem Classes

We assume knowledge of Deterministic and Nondeterministic Turing Machines. (DTM's and NDTM's)

The only use in life of a NDTM is to scan a string of characters X and proceed by state transitions until an 'accept' state is entered.

X <u>must</u> be in the language the NDTM is designed to recognize. Otherwise, it blows up!!

Problem Classes

So, what good is it?

We can count the number of transitions on the shortest path (elapsed time) to the accept state!!!

If there is a constant k for which the number of transitions is at most $|X|^k$, then the language is said to be 'nondeterministic polynomial.'

Problem Classes

The subset of YES instances of the set of instances of a decision problem, as we have described them above, is a language.

When given an instance, we want to know that it is in the subset of Yes instances. (All answers to Yes instances look alike - we don't care which one we get or how it was obtained).

This begs the question "What about the No instances?"

The answer is that we will get to them later. (They will actually form another class of problems.)

Problem Classes

This actually defines our first Class, NP, the set of decision problems whose Yes instances can be solved by a Nondeterministic Turing Machine in polynomial time.

That knowledge is not of much use!! We still don't know how to tell (easily) if a problem is in NP. And, that's our goal.

Fortunately, all we are doing with a NDTM is tracing the correct path to the accept state. Since all we are interested in doing is counting its length, if someone just gave us the correct path and we followed it, we could learn the same thing - how long it is.

Problem Classes

It is even simpler than that (all this has been proven mathematically). Consider the following problem:

You have a big van that can carry 10,000 lbs. You also have a batch of objects with weights w_1, w_2, …, w_n lbs. Their total sum is more than 10,000 lbs, so you can't haul all of them.

Can you load the van with exactly 10,000 lbs?

(WOW. That's the SubsetSum problem.)

Problem Classes

Now, suppose it is possible (i.e., a Yes instance) and someone tells you exactly what objects to select.

We can add the weights of those selected objects and verify the correctness of the selection.

This is the same as following the correct path in a NDTM. (Well, not just the same, but it can be proven to be equivalent.)

Therefore, all we have to do is count how long it takes to verify that a "correct" answer" is in fact correct.

Class – NP

First Significant Class of Problems:

The Class NP

Class – NP

We have, already, an informal definition for the set NP. We will now try to get a better idea of what NP includes, what it does not include, and give a formal definition.

Class – NP

Consider two seemingly closely related statements (versions) of a single problem. We show they are actually very different. Let G = (V, E) be a graph.

Definition: $X \subseteq V(G)$ is a *vertex cover* if every edge in G has at least one endpoint in X.

Class – NP

Version 1. Given a graph G and an integer k.
Does G contain a vertex cover
with at most k vertices?

Version 2. Given a graph G and an integer k.
Does the smallest vertex cover of G
have exactly k vertices?

Class – NP

Suppose, for either version, we are given a graph G and an integer k for which the answer is "yes." Someone also gives us a set X of vertices and claims

"X satisfies the conditions."

Class – NP

In Version 1, we can fairly easily check that the claim is correct – in polynomial time.

That is, in polynomial time, we can check that X has k vertices, and that X is a vertex cover.

Class – NP

In Version 2, we can also easily check that X has exactly k vertices and that X is a vertex cover.

<u>But</u>, we don't know how to easily check that there is not a smaller vertex cover!!

That seems to require exponential time.

These are very similar *looking* "decision" problems (Yes/No answers), yet they are VERY different in this one important respect.

Class – NP

In the first: We can verify a correct answer in polynomial time.

In the second: We apparently can not verify a correct answer in polynomial time.

(At least, we don't know how to verify one in polynomial time.)

Class – NP

Could we have asked to be given something that would have allowed us to easily verify that X was the smallest such set?

No one knows what to ask for!!

To check all subsets of k or fewer vertices requires exponential time (there can be an exponential number of them).

Class – NP

Version 1 problems make up the class called NP

Definition: The *Class NP* is the set of all decision problems for which answers to Yes instances can be verified in polynomial time.

{Why not the NO instances? We'll answer that later.}

For historical reasons, NP means

"Nondeterministic Polynomial."

(Specifically, it does not mean "not polynomial").

Class – NP

Version 2 of the Vertex Cover problem is not unique. There are other versions that exhibit this same property. For example,

Version 3: Given: A graph G = (V, E) and an integer k.

Question: Do all vertex covers of G have more than k vertices?

What would/could a 'witness' for a Yes instance be?

Class – NP

Again, no one knows except to list all subsets of at most k vertices. Then we would have to check each of the possible exponential number of sets.

Further, this is not isolated to the Vertex Cover problem. Every decision problem has a 'Version 3,' also known as the 'complement' problem (we will discuss these further at a later point).

Class – NP

All problems in NP are *decidable*.

That means there is an algorithm.

And, the algorithm is no worse than $O(2^n)$.

Class – NP

Version 2 and 3 problems are apparently not in NP.

So, where are they??

We need more structure! {Again, later.}

First we look inward, within NP.

Class – P

Second Significant Class of Problems: The Class P

Class – P

Some decision problems in NP can be solved (without knowing the answer in advance) - in polynomial time. That is, not only can we verify a correct answer in polynomial time, but we can actually <u>compute</u> the correct answer in polynomial time - from "scratch."

These are the problems that make up the class P.

P is a subset of NP.

Class – P

Problems in P can also have a witness – we just don't need one. But, this line of thought leads to an interesting observation. Consider the problem of searching a list L for a key X.

> Given: A list L of n values and a key X.
>
> Question: Is X in L?

Class – P

We know this problem is in P. But, we can also envision a nondeterministic solution. An oracle can, in fact, provide a "witness" for a Yes instance by simply writing down the index of where X is located.

We can verify the correctness with one simple comparison and reporting, Yes the witness is correct.

Class – P

Now, consider the complement (Version 3) of this problem:

Given: A list L of n values and a key X.

Question: Is X <u>not</u> in L?

Here, for any Yes instance, no 'witness' seems to exist, but if the oracle simply writes down "Yes" we can verify the correctness in polynomial time by comparing X with each of the n values and report "Yes, X is not in the list."

Class – P

Therefore, both problems can be verified in polynomial time and, hence, both are in NP.

This is a characteristic of any problem in P - both it and its complement can be verified in polynomial time (of course, they can both be 'solved' in polynomial time, too.)

Therefore, we can again conclude P $\subseteq$ NP.

Class – P

There is a popular conjecture that if any problem and its complement are both in NP, then both are also in P.

This has been the case for several problems that for many years were not known to be in P, but both the problem and its complement were known to be in NP.

For example, Linear Programming (proven to be in P in the 1980's), and Prime Number (proven in 2006 to be in P).

A notable 'holdout' to date is Graph Isomorphism.

Class – P

There are a lot of problems in NP that we do not know how to solve in polynomial time. Why?

Because they really don't have polynomial algorithms?

Or, because we are not yet clever enough to have found a polynomial algorithm for them?

Class – P

At the moment, no one knows.

Some believe all problems in NP have polynomial algorithms. Many do not (believe that).

The fundamental question in theoretical computer science is: Does P = NP?

There is an award of one million dollars for a proof.

– Either way, True or False.

Other Classes

We now look at other classes of problems.

Hard appearing problems can turn out to be easy to solve. And, easy looking problems can actually be very hard (Graph Theory is rich with such examples).

We must deal with the concept of "as hard as," "no harder than," etc. in a more rigorous way.

"No harder than"

Problem A is said to be 'no harder than' problem B when the smallest class containing A is a subset of the smallest class containing B.

Recall that $f_X(n)$ is the order of the smallest complexity class containing problem X.

If, for some constant α,

$$f_A(n) \leq n^{\alpha} \ f_B(n),$$

the time to solve A is no more than some polynomial multiple of the time required to solve B, i.e., A is 'no harder than' B.

"No harder than"

The requirement for determining the relative difficulty of two problems A and B requires that we know, at least, the order of the fastest algorithm for problem B and the order of some algorithm for Problem A.

We may not know either!!

In the following we exhibit a technique that can allow us to determine this relationship without knowing anything about an algorithm for either problem.

The "Key" to Complexity Theory

'Reductions,'
'Reductions,'
'Reductions.'

Reductions

For any problem X, let X(I_X, $Answer_X$) represents an algorithm for problem X – even if none is known to exist.

I_X is an arbitrary instance given to the algorithm and $Answer_X$ is the returned answer determined by the algorithm.

Reductions

Definition: For problems A and B, a (*Polynomial) Turing Reduction* is an algorithm $A(I_A, Answer_A)$ for solving all instances of problem A and satisfies the following:

(1) Constructs zero or more instances of problem B and invokes algorithm $B(I_B, Answer_B)$, on each.

(2) Computes the result, $Answer_A$, for I_A.

(3) Except for the time required to execute algorithm B, the execution time of algorithm A must be polynomial with respect to the size of I_A.

Reductions

Proc A(I_A, Answer$_A$)

For i = 1 to alpha

- **Compute I_B**
-

B(I_B, Answer$_B$)

-

End For

Compute Answer$_A$

End proc

Reductions

We may <u>assume</u> a 'best' algorithm for problem B without actually knowing it.

If $A(I_A, Answer_A)$ can be written without algorithm B, then problem A is simply a polynomial problem.

Poly Turing Reductions

The existence of a Turing reduction is often stated as:

"Problem A *reduces* to problem B" or, simply,

"$A \leq_P B$"

PolyTime Reductions

Theorem. If $A \leq_P B$ and problem B is polynomial, then problem A is polynomial.

Corollary. If $A \leq_P B$ and problem A is exponential, then problem B is exponential.

PT Reductions

The previous theorem and its corollary do not capture the full implication of Turing reductions.

Regardless of the complexity class problem B is in, a Turing reduction implies problem A is in a subclass.

Regardless of the class problem A might be in, problem B is in a super class.

PT Reductions

Theorem. **If $A \le_P B$, then problem A is "no harder than" problem B.**

Proof: **Let $t_A(n)$ and $t_B(n)$ be the maximum times for algorithms A and B per the definition. Thus, $f_A(n) \le t_A(n)$. Further, since we assume the best algorithm for B, $t_B(n) = f_B(n)$. Since $A \le_P B$, there is a constant k such that $t_A(n) \le n^k t_B(n)$. Therefore, $f_A(n) \le t_A(n) \le n^k t_B(n) = n^k f_B(n)$. That is, A is no harder than B.**

PT Reductions

Theorem.

If A $\leq_P$ B and B $\leq_P$ C then A $\leq_P$ C.

Definition.

If A $\leq_P$ B and B $\leq_P$ A, then A and B are *polynomially equivalent*.

Polynomial Reductions

$A \leq_P B$ means:

'Problem A is *no harder within a polynomial factor than* problem B,' and

'Problem B is *as hard within a polynomial factor as* is problem A.'

An Aside

Without condition (3) of the definition, a simple Reduction results.

If problem B is decidable,
then so is problem A.
Equivalently,
If problem A is undecidable,
then problem B is undecidable.

NP–Complete

Third Significant Class of Problems:

The Class NP–Complete

NP–Complete

Polynomial Transformations enforce an equivalence relationship on all decision problems, particularly, those in the Class NP. Class P is one of those classes and is the "easiest" class of problems in NP.

Is there a class in NP that is the hardest class in NP?

A problem B in NP such that A $\leq_P$ B for every A in NP.

NP–Complete

In 1971, Stephen Cook proved there was. Specifically, a problem called

Satisfiability **(or, SAT).**

Satisfiability

$U = \{u_1, u_2, \ldots, u_n\}$, Boolean variables.

$C = \{c_1, c_2, \ldots, c_m\}$, "OR clauses"

For example:

$$c_i = (u_4 \vee u_{35} \vee \sim u_{18} \vee u_3 \ldots \vee \sim u_6)$$

Satisfiability

Can we assign Boolean values to the variables in U so that every clause is TRUE?

There is no known polynomial time algorithm!!

NP–Complete

Cooks Theorem:

1) SAT is in NP
2) For every problem A in NP,

$$A \leq_P SAT$$

Thus, SAT is as hard as every problem in NP.

NP–Complete

Since SAT is itself in NP, that means SAT is a hardest problem in NP (there can be more than one.).

A hardest problem in a class is called the "completion" of that class.

Therefore, SAT is NP–Complete.

NP–Complete

Today, there are 1,000’s of problems that have been proven to be NP–Complete. (See Garey and Johnson, *Computers and Intractability: A Guide to the Theory of NP–Completeness*, for a list of over 300 as of the early 1980's).

P = NP?

If P = NP then all problems in NP are polynomial problems.

If P ≠ NP then all NP–C problems are at least super-polynomial and perhaps exponential. That is, NP-C problems could require sub-exponential super-polynomial time. (Example of super-polynomial, sub-exponential is $\mathbf{o(2^{o(n)})}$, e.g., $\mathbf{2^{\sqrt[3]{n}}}$

P = NP?

Why should P equal NP?

- There seems to be a huge "gap" between the known problems in P and Exponential. That is, almost all known polynomial problems are no worse than n^3 or n^4.
- Where are the $O(n^{50})$ problems?? $O(n^{100})$? Maybe they are the ones in NP–Complete?
- It's awfully hard to envision a problem that would require n^{100}, but surely they exist?
- Some of the problems in NP–C just look like we should be able to find a polynomial solution (looks can be deceiving, though).

P ≠ NP?

Why should P not equal NP?

- P = NP would mean, for any problem in NP, that it is just as easy to solve an instance form "scratch," as it is to verify the answer if someone gives it to you. That seems a bit hard to believe.
- There simply are a lot of awfully hard looking problems in NP–Complete (and Co–NP-Complete) and some just don't seem to be solvable in polynomial time.
- Many smart people have tried for a long time to find polynomial algorithms for some of the problems in NP-Complete - with no luck.

NP-Complete; NP-Hard

A decision problem, *C*, is NP-complete if:

C is in NP and

C is NP-hard. That is, every problem in NP is polynomially reducible to C.

D polynomially reduces to *C* means that there is a deterministic polynomial-time many-one algorithm, *f*, that transforms each instance *x* of *D* into an instance f(x) of *C*, such that the answer to *f(x)* is YES if and only if the answer to *x* is YES.

To prove that an NP problem *A* is NP-complete, it is sufficient to show that an already known NP-complete problem polynomially reduces to *A*. By transitivity, this shows that *A* is NP-hard.

A consequence of this definition is that if we had a polynomial time algorithm for any NP-complete problem *C*, we could solve all problems in NP in polynomial time. That is, P = NP.

Note that NP-hard does not necessarily mean NP-complete, as a given NP-hard problem could be outside NP.

Returning to SAT

- SAT is the problem to decide of an arbitrary Boolean formula (wff in the propositional calculus) whether or not this formula is satisfiable (has a set of variable assignments that evaluate the expression to true).
- SAT clearly can be solved in time $k2^n$, where k is the length of the formula and n is the number of variables in the formula.
- What we now show is that SAT is NP-complete, providing us our first concrete example of an NP-complete decision problem.

Simulating NDTM

- Given a NDTM, **M**, and an input **w**, we need to create a formula, $\varphi_{\mathbf{M},\mathbf{w}}$, containing a polynomial number of terms that is satisfiable just in case **M** accepts **w** in polynomial time.
- The formula must encode within its terms a trace of configurations that includes
  - A term for the starting configuration of the TM
  - Terms for all accepting configurations of the TM
  - Terms that ensure the consistency of each configuration
  - Terms that ensure that each configuration after the first follows from the prior configuration by a single move

Tableaus

A **tableau** is an array of tape alphabet symbols.

- It represents a configuration history of **one branch** of our NDTM's nondeterminism.
- If the NDTM runs in n^k time, the tableau is an $(n^k \times n^k)$ tableau.
  - It's big enough downward because, well, the TM runs in n^k.
  - …and rightward because the TM can only *count* to n^k.
- We assume that every configuration starts and ends with a # symbol.
- We think of our tableau as looking like this in the "beginning": the starting configuration across the top, and the other configurations blank.
  - (We quote "beginning" because SAT isn't really a stateful algorithm, but just go with it for now.)
- But we've assumed that we can "represent" alphabet symbols. How do we do that, in *SAT*?

#	q_0	w_1	w_2	…	w_n	□	…	□	#	
#									#	
#									#	
#									#	
#									#	
#									#	$\uparrow n^k \downarrow$
#									#	
#									#	
#									#	
#									#	
$\leftarrow n^k \rightarrow$										

Encoding the Tableau: Basics

Consider a set comprised of:

The tape alphabet

The state set

The separator character

$$C = \Gamma \cup Q \cup \{ \# \}$$

Consider a cell variable:

$$x_{i,j,c}$$

Turning this variable on *corresponds to* ***setting cell (i, j) = c***, for some $c \in C$.

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	w_1	w_2	…	w_n	□	…	□	#
2	#									#
3	#									#
4	#									#
5	#									#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Encoding the Tableau: Cells

Consider our tableau alphabet:

$$C = \Gamma \cup Q \cup \{ \# \}$$

Consider a cell and corresponding variable:

$$x_{i,j,c}$$

Now we need to make sure the tableau is consistently encoded.

Create a clause for **each cell (*i*, *j*)**.

$$\phi_{\text{encode}}(i,j) = \left[\left(\bigvee_{c \in C} x_{i,j,c}\right) \wedge \left(\bigwedge_{\substack{c,d \in C \\ c \neq d}} \left(\overline{x_{i,j,c}} \vee \overline{x_{i,j,d}}\right)\right)\right]$$

The left demands $x_{i,j,c}$ be true for **some *c*.**
The right demands $x_{i,j,c}$ be true for **only one *c*.**

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	w_1	w_2	…	w_n	□	…	□	#
2	#									#
3	#									#
4	#									#
5	#									#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Encoding the Tableau: The Tableau

Tableau alphabet: $C = \Gamma \cup Q \cup \{\#\}$

Cell variable: $x_{i,j,c}$

Create an encoding clause for each cell (i, j).

$$\phi_{\text{encode}}(i,j) = \left[\left(\bigvee_{c \in C} x_{i,j,c}\right) \wedge \left(\bigwedge_{\substack{c,d \in C \\ c \neq d}} \left(\overline{x_{i,j,c}} \vee \overline{x_{i,j,d}}\right)\right)\right]$$

Now repeat the clause across the tableau.

$$\phi_{\text{cells}} = \bigwedge_{1 \le i,j \le n^k} \phi_{\text{encode}}(i,j)$$

This is our *cell formula*. It ensures that each cell in the tableau is assigned a single symbol.

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	w_1	w_2	…	w_n	□	…	□	#
2	#									#
3	#									#
4	#									#
5	#									#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Encoding the Tableau: Complexity

$$\phi_{\text{encode}}(i,j) = \left[\left(\bigvee_{c\in C} x_{i,j,c}\right) \wedge \left(\bigwedge_{\substack{c,d\in C\\ c\neq d}} \left(\overline{x_{i,j,c}} \vee \overline{x_{i,j,d}}\right)\right)\right]$$

We can create the single-cell encoding formula in polynomial time with a $|C|^2$ iteration.

$$\phi_{\text{cells}} = \bigwedge_{1\le i,j\le n^k} \phi_{\text{encode}}(i,j)$$

We can create the *entire* cell formula in polynomial time with an n^{2k} iteration around that.

So we can say that ϕ_{cells} **is satisfied by, and only by, a properly encoded tableau, and is created in polynomial time.**

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	w_1	w_2	…	w_n	□	…	□	#
2	#									#
3	#									#
4	#									#
5	#									#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Starting and Accepting

Starting and accepting are (comparatively) easy.

To start, take the start configuration padded to n^k length with blanks…

$S = \#q_0w_1w_2\ldots w_n\square\ldots\square\#$ so that $|S| = n^k$

…and **require the first row be equal to the start configuration**:

$$\phi_{\text{start}} = \bigwedge_{1\le j\le n^k} \left[x_{1,j,s_j}\right]$$

Then to accept, just **require an accept state somewhere in the tableau.**

$$\phi_{\text{accept}} = \bigvee_{1\le i,j\le n^k} \left[x_{i,j,q_A}\right]$$

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	w_1	w_2	…	w_n	□	…	□	#
2	#									#
3	#									#
4	#									#
5	#	w_1	w_2	…	q_A	…	□	…	□	#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Starting and Accepting

$$\phi_{\text{start}} = \bigwedge_{1 \le j \le n^k} \left[x_{1,j,s_j} \right]$$

$$\phi_{\text{accept}} = \bigvee_{1 \le i,j \le n^k} \left[x_{i,j,q_A} \right]$$

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	w_1	w_2	…	w_n	□	…	□	#
2	#									#
3	#									#
4	#									#
5	#	z_1	z_2	…	q_A	…	□	…	□	#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

We can generate the start and accept formulas in n^k and $(n^k)^2$ time, both polynomial.

So now we can say that:

ϕ_{start} is satisfied by, and only by, a tableau with the starting configuration of *M* on *w* encoded as its first row, and is created in polynomial time.

…and…

ϕ_{accept} is satisfied by, and only by, a tableau encoding an accepting configuration as one of its rows, and is created in polynomial time.

Transitions

Now, for transitions. Recall the discussions we had about ID changes being limited to three characters or six, when looking at transitions..

> A given 2x3 **window** is **legal** if it does not violate our machine's transition function.
>
> Given the linear sets of states and tape symbols, and the finite size of 2x3 windows, we can make a **polynomial-sized set of all legal windows**.

Let a sequence $A = (a_1, \ldots, a_6)$ be a 2x3 window, with a_1 the top left cell, a_2 the top middle, etc.

> We say that ***A* is *legal*** if it **represents a legal window**. Here we have $\mathbf{q_0}$ **a R** $\mathbf{q_1}$

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	*a*	*b*	c	*a*	□	□	□	#
2	#	*a*	q_1	*b*	c	*a*	□	□	□	#
3	#									#
4	#									#
5	#									#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Transitions

A given 2x3 **window** is **legal** if it does not violate our machine's transition function. We have a **polynomial-sized set of all legal windows**.

Let a sequence $A = (a_1, \ldots, a_6)$ be a 2x3 window. ***A* is *legal*** if it **represents a legal window**.

Now we can come up with a formula to say that the window top-centered at cell (i, j) is legal.

$$\phi_{\text{legal}}(i,j) = \bigvee_{\substack{A=(a_1,\ldots,a_6)\\ \text{is legal}}} \begin{bmatrix} x_{i,j-1,a_1} \land x_{i,j,a_2} \land x_{i,j+1,a_3} \land \\ x_{i+1,j-1,a_4} \land x_{i+1,j,a_5} \land x_{i+1,j+1,a_6} \end{bmatrix}$$

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	*a*	*b*	c	*a*	□	□	□	#
2	#	*a*	q_1	*b*	c	*a*	□	□	□	#
3	#									#
4	#									#
5	#									#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Don't be intimidated by this formula!

It's just **counting off the six cells of the window** and demanding that each be **equal to the corresponding cell** in **some legal window**.

Transitions

A given 2x3 **window** is **legal** if it does not violate our machine's transition function.

We have a **polynomial-sized set of all legal windows**.

Let a sequence $A = (a_1, \ldots, a_6)$ be a 2x3 window. ***A* is *legal*** if it **represents a legal window**.

$$\phi_{\text{legal}}(i,j) = \bigvee_{\substack{A=(a_1,\ldots,a_6) \\ \text{is legal}}} \begin{bmatrix} x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge \\ x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6} \end{bmatrix}$$

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	*a*	*b*	c	*a*	□	□	□	#
2	#	*a*	q_1	*b*	c	*a*	□	□	□	#
3	#									#
4	#									#
5	#									#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Since we have a polynomial number of legal windows, this formula is also polynomial. So we can say:

ϕ_{legal} (*i*, *j*) is satisfied by, and only by, a tableau whose window top-centered at (*i*, *j*) is legal; and is created in polynomial time.

Windows and Configurations

Consider any **upper** and **lower** configuration in the tableau, so that the lower configuration is the one immediately below – that is, following – the upper.

If all the windows top-centered on cells in the upper configuration are legal, then:

> The legality of the windows that don't involve the state symbol easily ensures the legality of the configuration below them.
>
> The window top-centered on the state symbol in the upper configuration is sufficient to ensure that the state symbol in the lower configuration makes a legal move.

The upper configuration yields the lower one if and only if all the windows top-centered on cells in the upper configuration are legal – and that holds all the way down the tableau.

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	*a*	*b*	c	*a*	□	□	□	#
2	#	*a*	q_1	*b*	c	*a*	□	□	□	#
3	#									#
4	#									#
5	#									#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Windows and Configurations

$$\phi_{\text{legal}}(i,j) = \bigvee_{\substack{A=(a_1,\dots,a_6)\\ \text{is legal}}} \begin{bmatrix} x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge \\ x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6} \end{bmatrix}$$

ϕ_{legal} (i, j) is satisfied by, and only by, a tableau whose window top-centered at (i, j) is legal; and is created in polynomial time.

An upper configuration yields a lower one iff all the windows top-centered within the upper are legal.

This holds all the way down the tableau.

Then we have:

$$\phi_{\text{move}} = \bigwedge_{\substack{1\le i<n^k,\\ 1<j<n^k}} \phi_{\text{legal}}(i,j)$$

And can say ϕ_{move} **is satisfied by, and only by, a tableau that does not violate the machine's transition function; and is created in polynomial time.**

	1	2	3	4	5	6	7	8	9	10
1	#	q_0	*a*	*b*	c	*a*	□	□	□	#
2	#	*a*	q_1	*b*	c	*a*	□	□	□	#
3	#									#
4	#									#
5	#									#
6	#									#
7	#									#
8	#									#
9	#									#
10	#									#

Pulling It Together

$$\phi_{\text{cells}} = \bigwedge_{1 \le i,j \le n^k} \phi_{\text{encode}}(i,j)$$

$$\phi_{\text{start}} = \bigwedge_{1 \le j \le n^k} \left[x_{1,j,s_j}\right]$$

$$\phi_{\text{accept}} = \bigvee_{1 \le i,j \le n^k} \left[x_{i,j,q_{\text{A}}}\right]$$

$$\phi_{\text{move}} = \bigwedge_{\substack{1 \le i < n^k, \\ 1 < j < n^k}} \phi_{\text{legal}}(i,j)$$

$$\phi_{\text{NDTM}} = (\phi_{\text{cells}} \wedge \phi_{\text{start}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}})$$

We have:

ϕ_{cells} is satisfied by, and only by, a properly encoded tableau.

ϕ_{start} is satisfied by, and only by, a tableau with the starting configuration of *M* on *w* encoded as its first row.

ϕ_{accept} is satisfied by, and only by, a tableau encoding an accepting configuration as one of its rows.

ϕ_{move} is satisfied by, and only by, a tableau that does not violate the machine's transition function.

All are created in polynomial time.

Then ϕ_{NDTM} **is satisfied by, and only by, a tableau encoding an accepting computation history of *M* on *w*, and is created in polynomial time.**

SAT is NP-Complete

$$\phi_{\text{NDTM}} = (\phi_{\text{cells}} \wedge \phi_{\text{start}} \wedge \phi_{\text{accept}} \wedge \phi_{\text{move}})$$

ϕ_{NDTM} created from NDTM *M* and input *w* is satisfied by, and only by, a tableau encoding an accepting computation history of *M* on *w*, and is created in polynomial time.

This means that:

SAT accepts ϕ_{NDTM} if and only if such a tableau exists…

…if and only if the NDTM we are encoding into ϕ_{NDTM} accepts *w*.

We've just polynomially reduced every possible NP language to *SAT*.

Let's convince ourselves of that a bit more.

By definition, any NP language has an NDTM *M* that decides it in polynomial time.

We can decide any NP language with a result from *SAT* using the following algorithm:

On input <*M*, *w*>:

Create ϕ_{NDTM} from *M* and *w*.

Run the decider for *SAT* on ϕ_{NDTM}.

Accept if *SAT* accepts, reject if it rejects.

***SAT* is NP-complete.**

Cook's Theorem

- $\varphi_{M,w} = \phi_{cells} \wedge \phi_{start} \wedge \phi_{accept} \wedge \phi_{move}$
- **See the following for another detailed description and discussion of the four terms that make up this formula.**
- http://www.cs.tau.ac.il/~safra/Complexity/Cook.ppt

NP–Complete

Within a year, Richard Karp added 22 problems to this special class.

We will focus on:

3-SAT

Integer Linear Programming

SubsetSum

Partition

Vertex Cover

Independent Set

K-Color

Multiprocessor Scheduling

Co-NP

- A problem is in co-NP if its complement is in NP – this is like co-RE, wrt RE problems.
- An example is the problem to determine if a Boolean expression is a tautology.
  - If the answer to the problem "is B in TAUT ?" is NO, then ¬A is in SAT.
- A more direct example of a co-NP problem is to determine if a Boolean expression is self-contradictory.
  - This is the complement of satisfiability.
- Both of the above are co-NP Complete

SAT to 3SAT

- 3-SAT means that each clause has exactly three terms
- If one term, e.g., (p), expand to $(p \vee p \vee p)$
- If two terms, e.g., $(p \vee q)$, expand to $(p \vee q \vee p)$
- Any clause with three terms is fine
- If n > three terms, can reduce to two clauses, one with three terms and one with n-1 terms, e.g., $(p1 \vee p2 \vee \ldots \vee pn)$ to
  $(p1 \vee p2 \vee z)$ & $(p3 \vee \ldots \vee pn \vee \sim z)$, where z is a new variable. If n=4, we are done, else apply this approach again with the clause having n-1 terms

Integer Linear Programming

- Show for 0-1 integer linear programming by constraining solution space. Start with an instance of SAT (or 3SAT), assuming variables $v1, \ldots, vn$ and clauses $c1, \ldots, cm$
- For each variable vi, have constraint that $0 \le vi \le 1$
- For each clause we provide a constraint that it must be satisfied (evaluate to at least 1). For example, if clause cj is $v2 \lor \sim v3 \lor v5 \lor v6$ then add the constraint
  $v2 + (1-v3) + v5 + v6 \ge 1$
- A solution to this set of integer linear constraints implies a solution to the instance of SAT and vice versa

SubsetSum

$S = \{s_1, s_2, \ldots, s_n\}$

set of positive integers

and an integer B.

Question: Does S have a subset whose values sum to B?

No one knows of a polynomial algorithm.

{No one has proven there isn't one, either!!}

SubsetSum $\equiv_p$ Partition

Theorem. SAT $\le_P$ 3SAT

Theorem. 3SAT $\le_P$ SubsetSum

Theorem. SubsetSum $\le_P$ Partition

Theorem. Partition $\le_P$ SubsetSum

Therefore, not only is Satisfiability in NP–Complete, but so is 3SAT, Partition, and SubsetSum.

3SAT $\leq_p$ SubsetSum

Assuming a **3SAT expression (a + ~b + c) (~a + b + ~c)**

	a	b	c	a+~b+c	~a+b+~c
a	1	0	0	1	0
~a	1	0	0	0	1
b	0	1	0	0	1
~b	0	1	0	1	0
c	0	0	1	1	0
~c	0	0	1	0	1
C1	0	0	0	1	0
C1’	0	0	0	1	0
C2	0	0	0	0	1
C2’	0	0	0	0	1
	1	1	1	3	3

SubsetSum$\equiv_p$Partition Details

- Partition is polynomial equivalent to SubsetSum
  - Let $i_1, i_2, .., i_n$, G be an instance of SubsetSum. This instance has answer "yes" iff
    $i_1, i_2, .., i_n$, 2*Sum($i_1, i_2, .., i_n$) – G,Sum($i_1, i_2, .., i_n$) + G
    has answer "yes" in Partition. Here we assume that G ≤ Sum($i_1, i_2, .., i_n$), for, if not, the answer is "no."
  - Let $i_1, i_2, .., i_n$ be an instance of Partition. This instance has answer "yes" iff
    $i_1, i_2, .., i_n$, Sum($i_1, i_2, .., i_n$)/2
    has answer "yes" in SubsetSum

SubsetSum $\equiv_p$ Partition

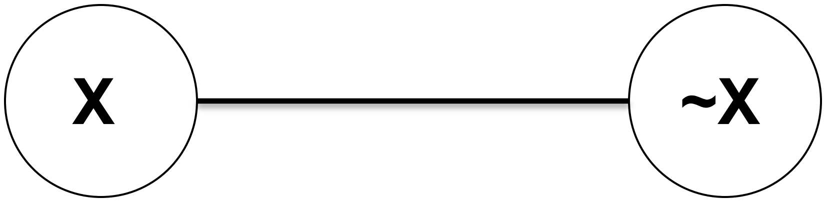
- [(15, 17, 27, 11, 4, 12, 33, 5, 6, 21, 2), 57]
- A solution is 15, 17, 11, 12, 2
- Sum of all is 153
- Mapping to Partition is
  - (15, 17, 27, 11, 4, 12, 33, 5, 6, 21, 2, 306-57, 153+57)
  - (15, 17, 27, 11, 4, 12, 33, 5, 6, 21, 2, 249, 210)
  - (15+17+11+12+2+249) = 306
  - (27+4+33+5+6+21+210) = 306
- Going other direction map above to
  - [(15, 17, 27, 11, 4, 12, 33, 5, 6, 21, 2, 249, 210), 306]

VERTEX COVERING (VC) DECISION PROBLEM IS NP-HARD

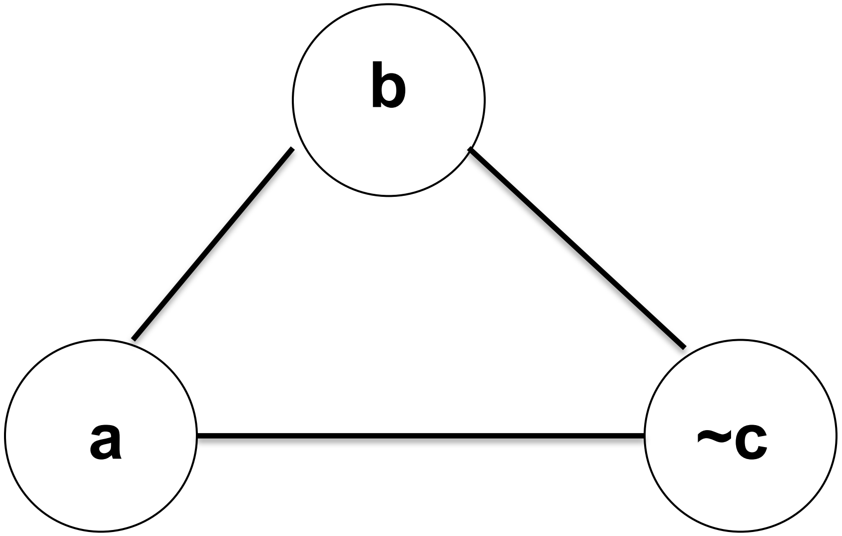
3SAT to Vertex Cover

- Vertex cover seeks a set of vertices that cover every edge in some graph
- Let $I_{3\text{-}SAT}$ be an arbitrary instance of 3-SAT. For integers n and m, $U = \{u_1, u_2, \ldots, u_n\}$ and $C_i = \{z_{i1}, z_{i2}, z_{i3}\}$ for $1 \le i \le m$, where each z_{ij} is either a u_k or u_k' for some k.
- Construct an instance of VC as follows.
- For each i, $1 \le i \le n$, construct two vertices, u_i and u_i' with an edge between them.
- For each clause $C_i = \{z_{i1}, z_{i2}, z_{i3}\}$, $1 \le i \le m$, construct three vertices z_{i1}, z_{i2}, and z_{i3} and form a "triangle on them. Each z_{ij} is one of the Boolean variables u_k or its complement u_k'. Draw an edge between z_{ij} and the Boolean variable (whichever it is). Each z_{ij} has degree 3. Finally, set k = n+2m.
- Theorem. The given instance of 3-SAT is satisfiable if and only if the constructed instance of VC has a vertex cover with at most k vertices.

VC Variable Gadget



VC Clause Gadget

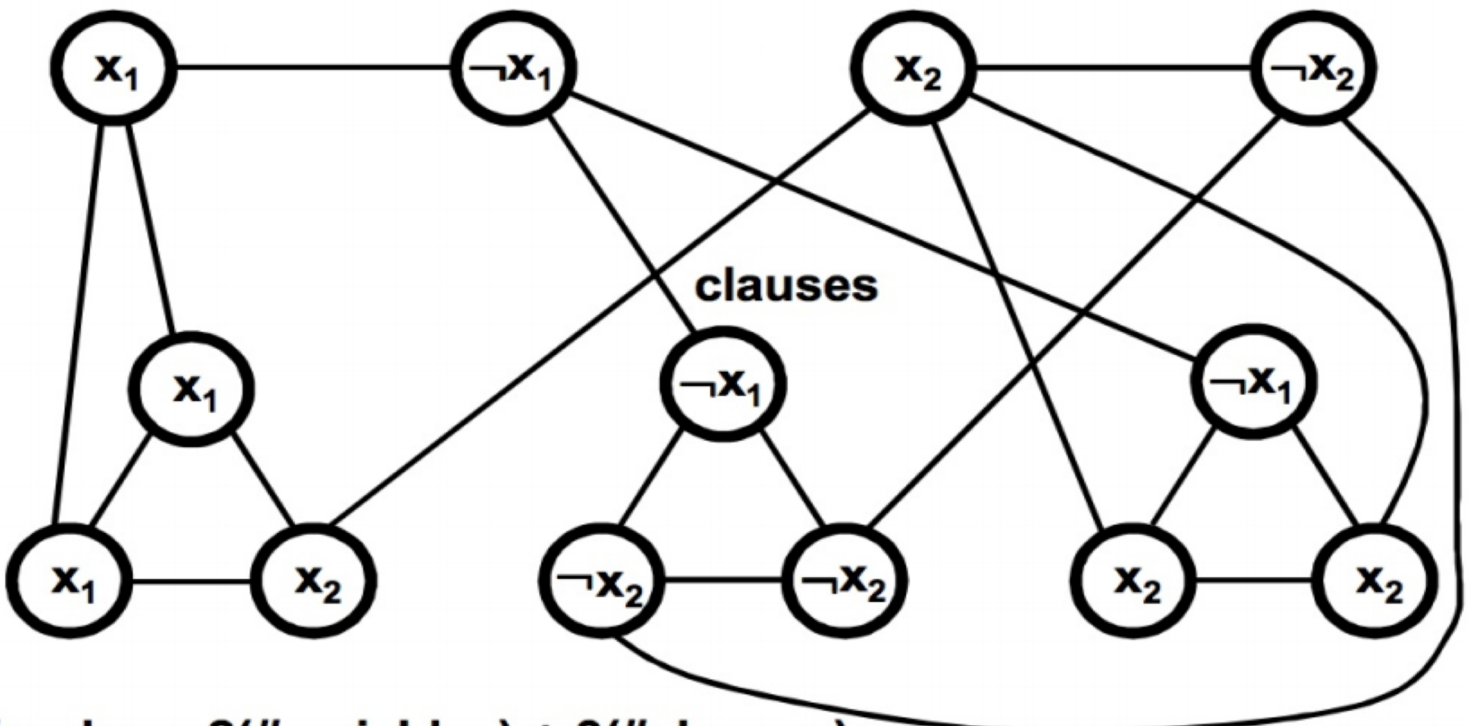


a + b + ~c

VC Gadgets Combined

$(x_1 \vee x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_2) \wedge (\neg x_1 \vee x_2 \vee x_2)$

Variables and negations of variables

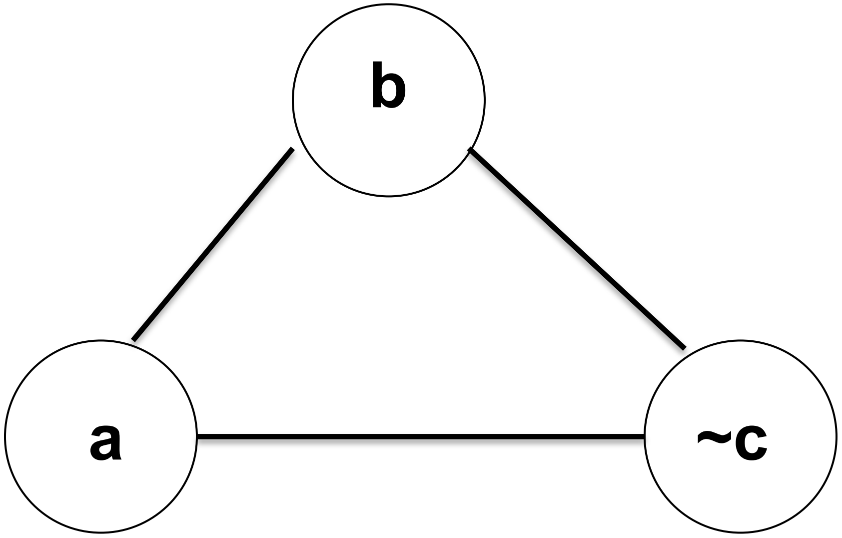


#nodes = 2(#variables) + 3(#clauses)

Independent Set

- Independent Set
  - Given Graph G = (V, E), a subset S of the vertices is independent if there are no edges between vertices in S
  - The k-IS problem is to determine for a k>0 and a graph G, whether or not G has an independent set of k nodes
- Note there is a related NP-Hard optimization problem to find a Maximum Independent Set. It is even hard to approximate a solution to the Maximum Independent Set Problem.

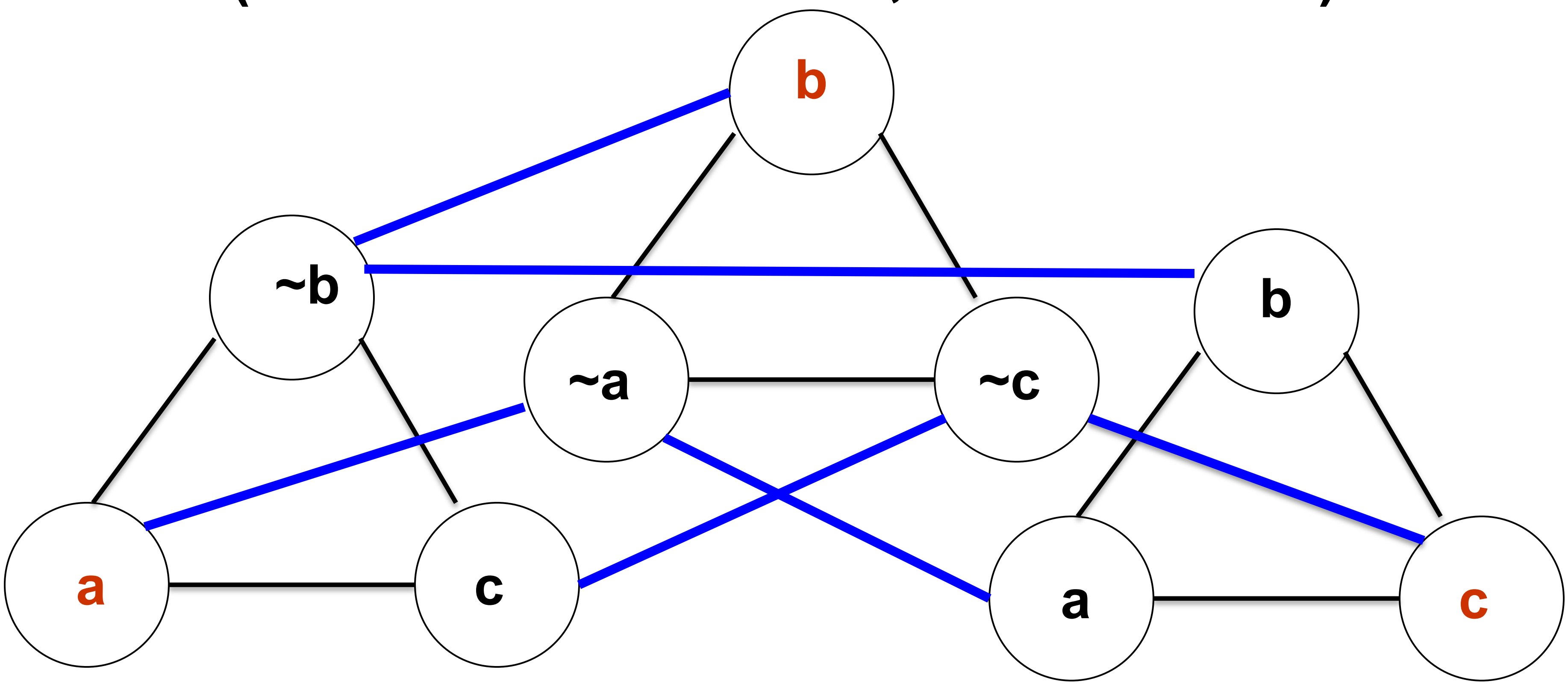
IS (VC) Clause Gadget



a + b + ~c

3SAT to IS

(a + ~b + c) (~a + b + ~c)(a + b + c), k=3
(k=number of clauses, not variables)



K-COLOR (KC) DECISION PROBLEM IS NP-HARD

K-Coloring

Given:

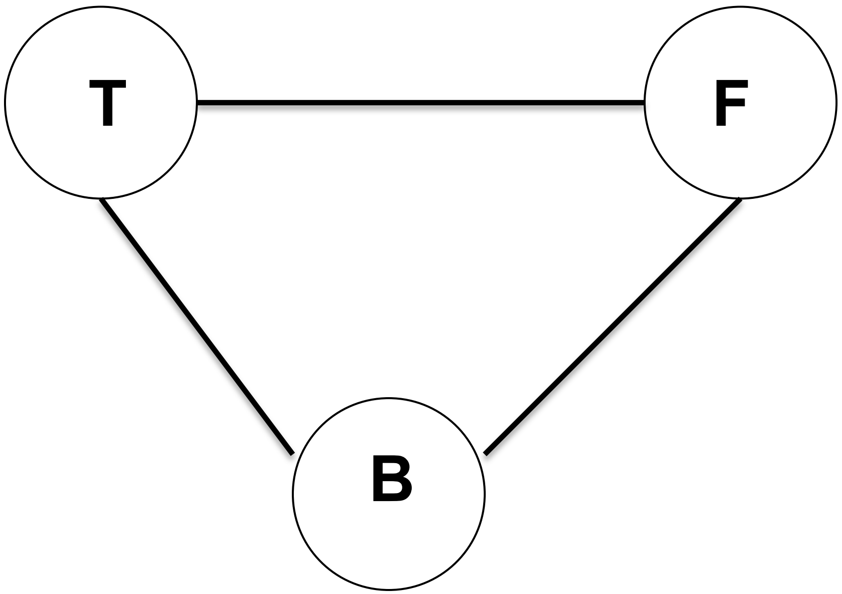
A graph G = (V, E) and an integer k.

Question:

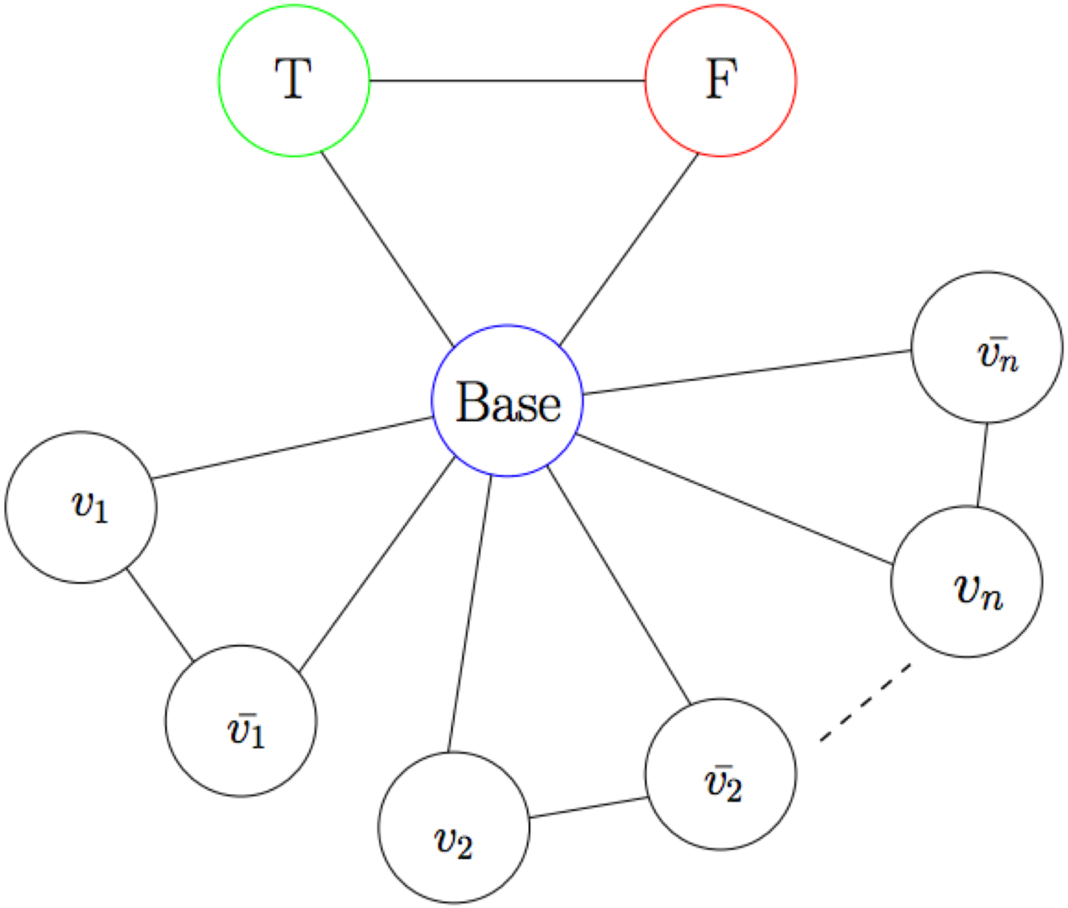
Can the vertices of G be assigned colors from a palette of size k, so that adjacent vertices have different colors and use at most k colors?

3Coloring (3C) uses k=3

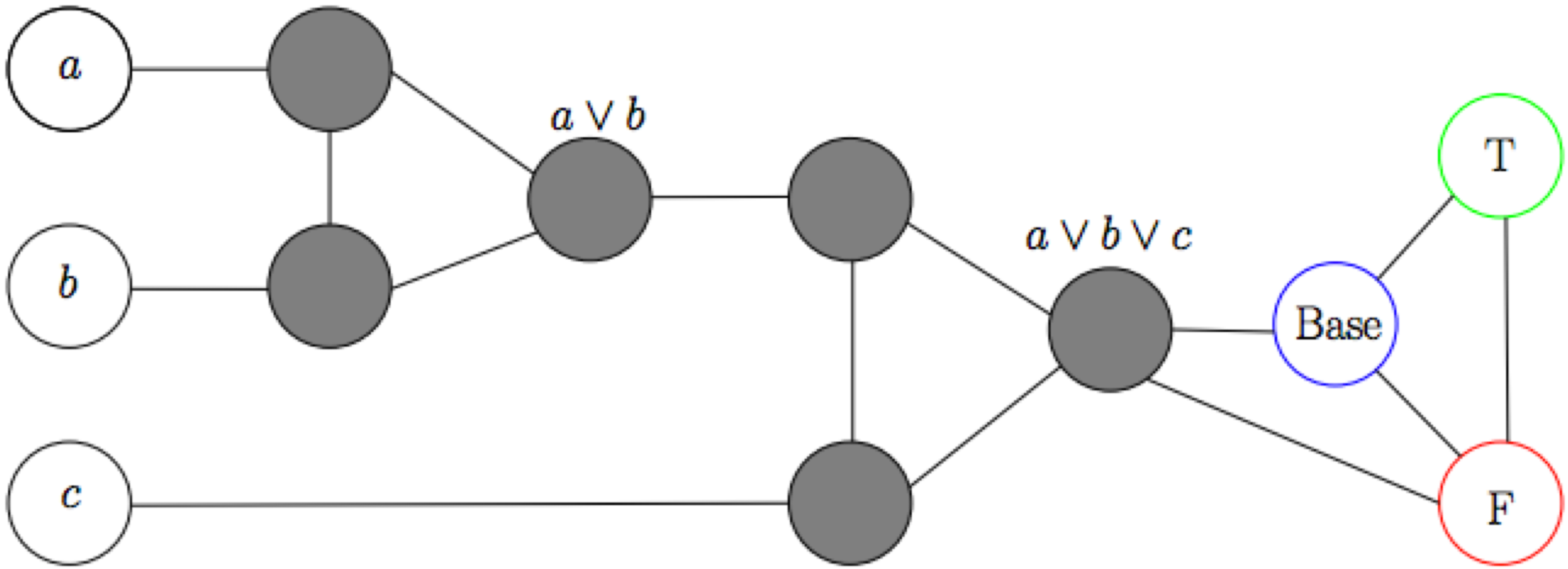
3C Super Gadget



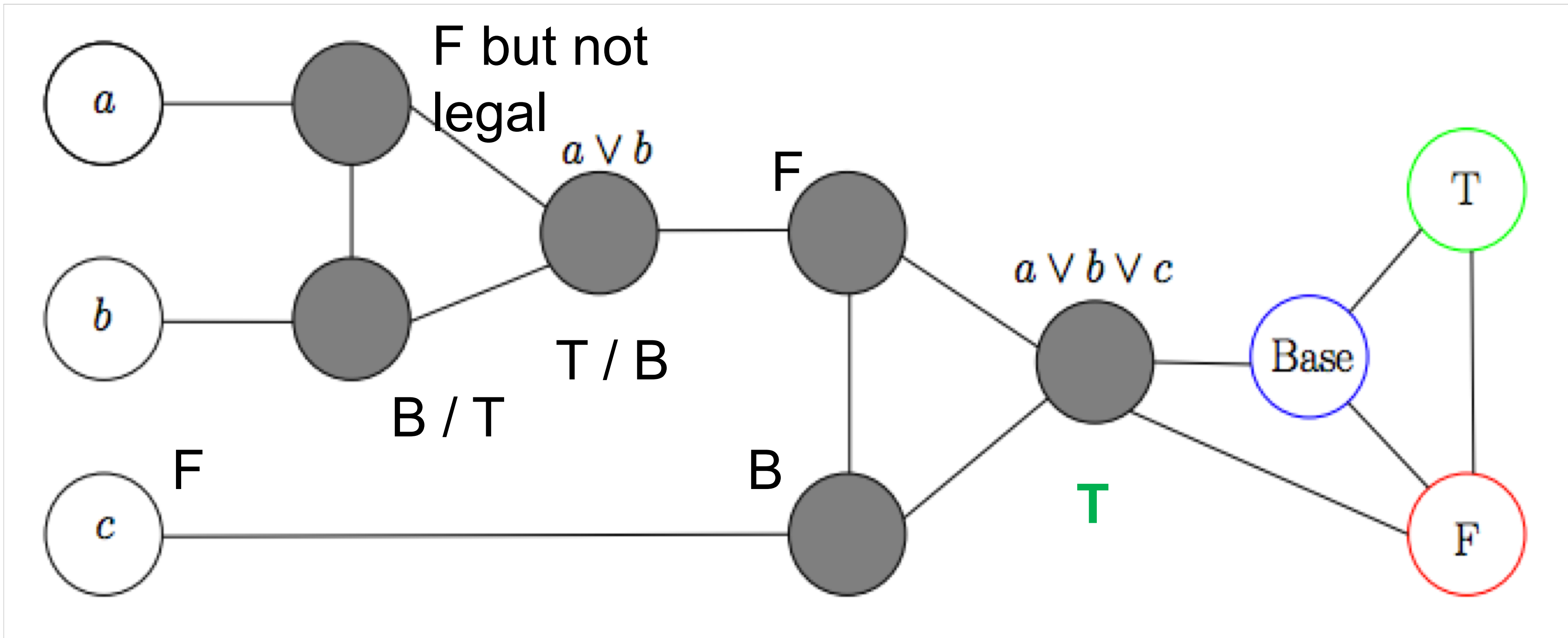
KC Super + Variables Gadget



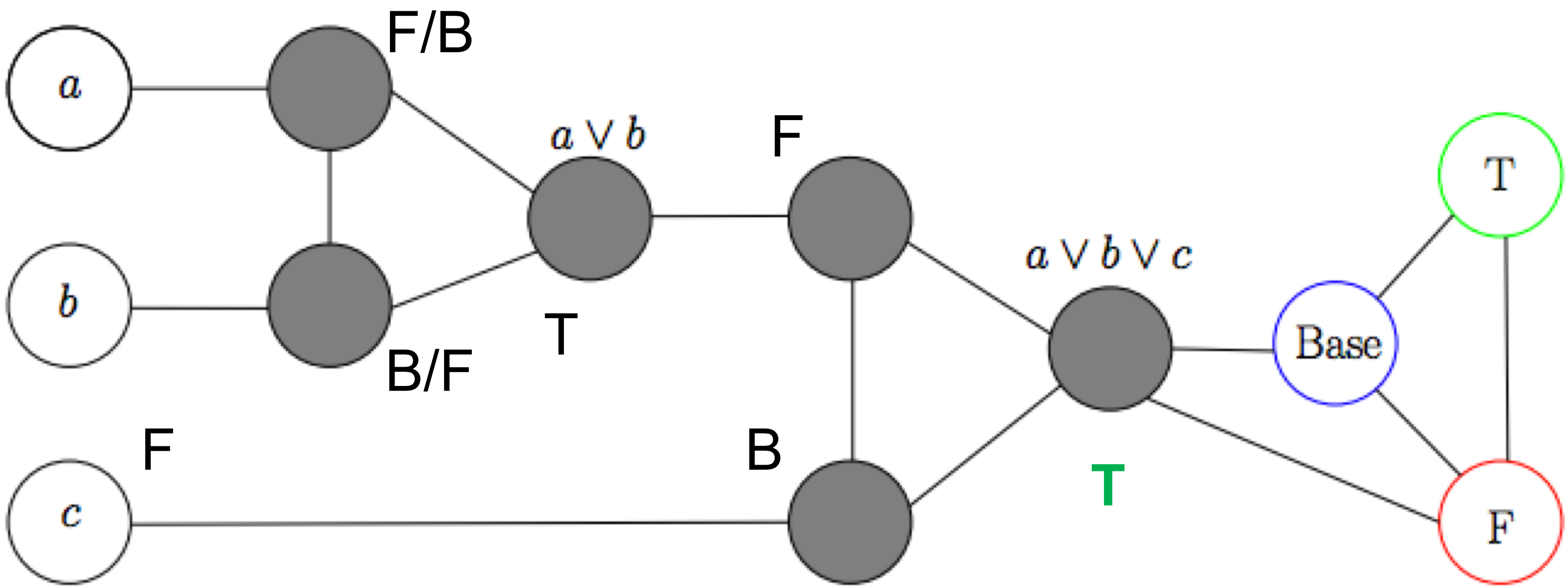
KC Clause Gadget



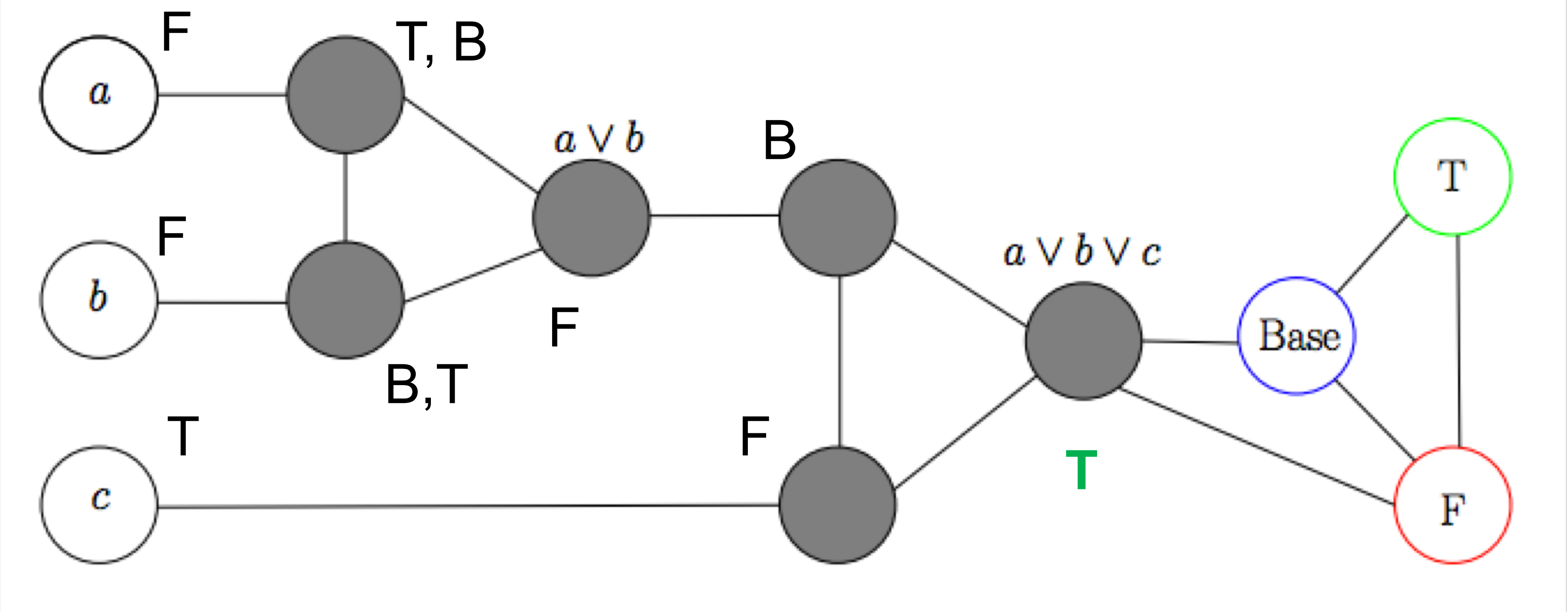
Consider ~a, ~b, ~c



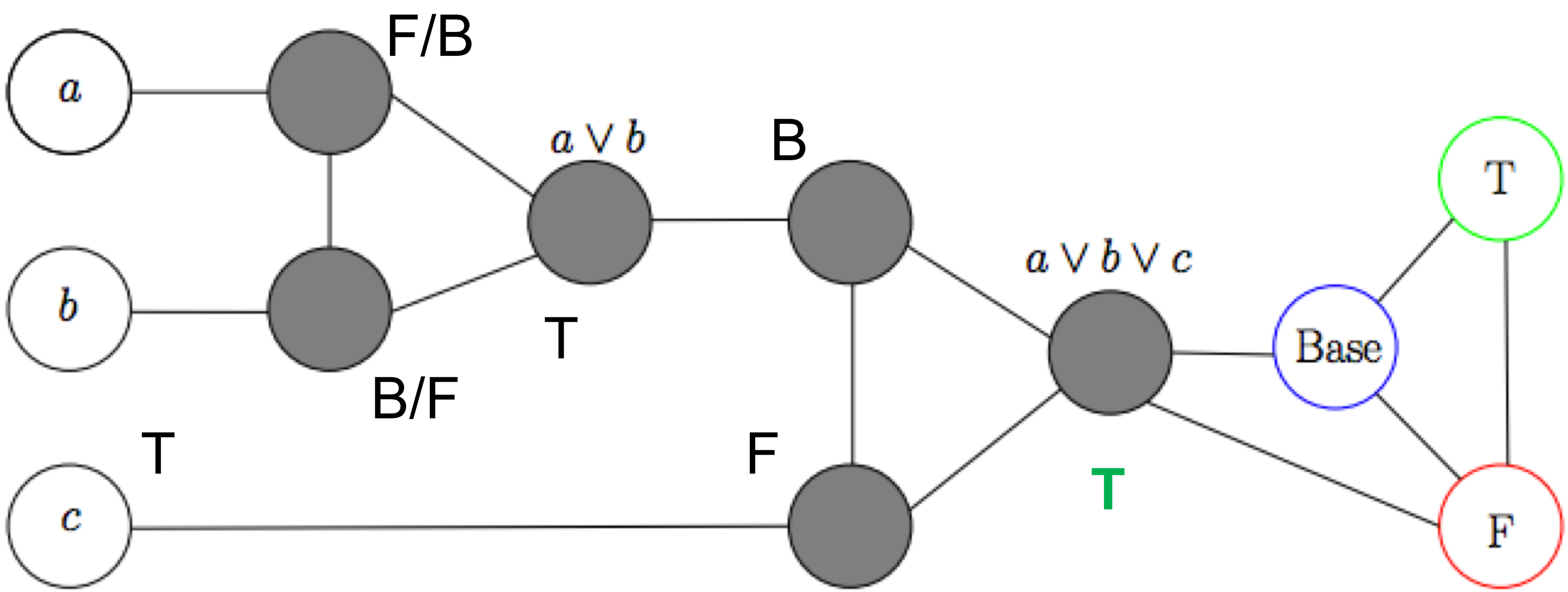
Consider a || b, ~c



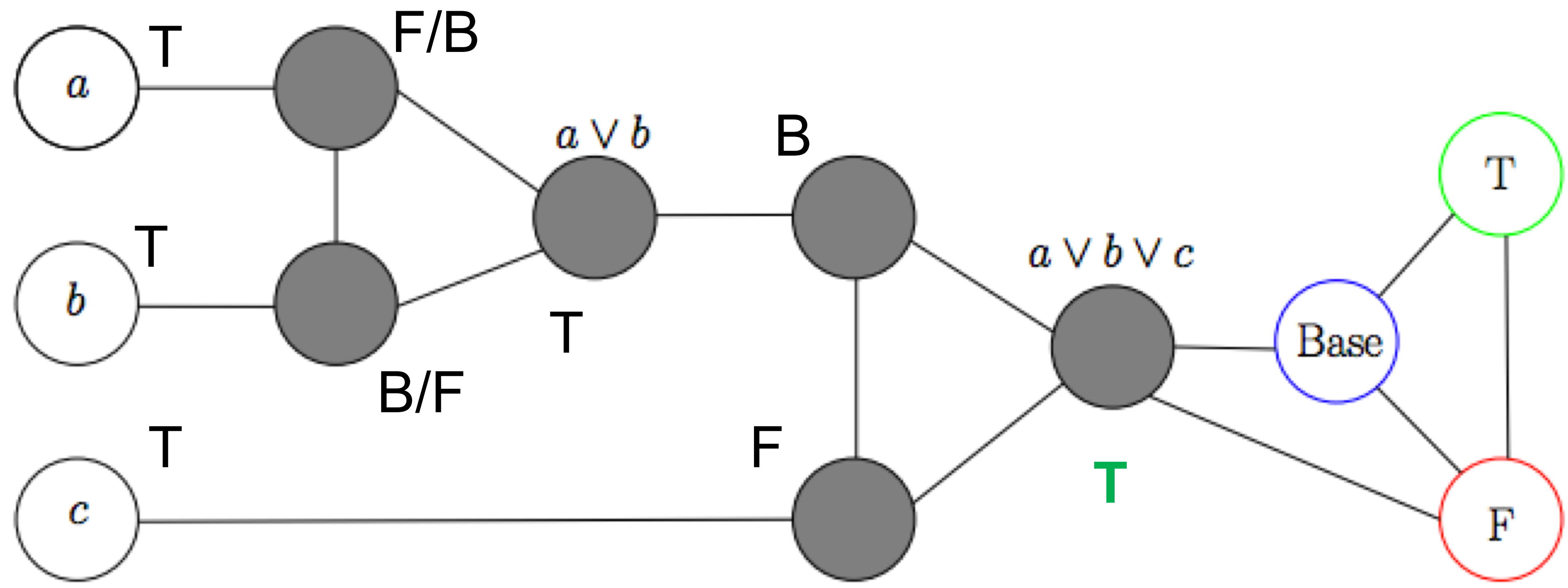
Consider ~a, ~b, c



Consider one of a || b, c

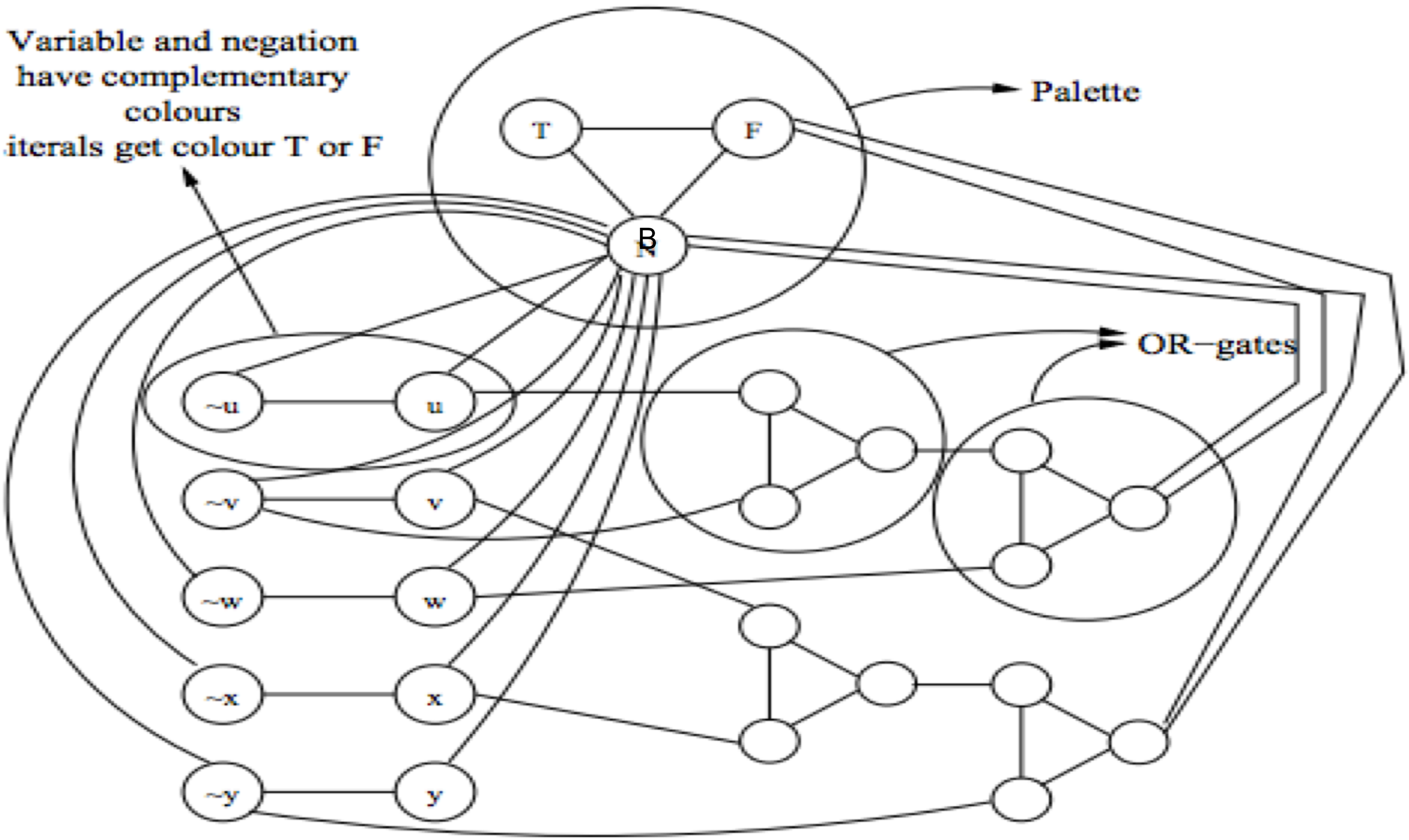


Consider a, b, c



KC Gadgets Combined

(u + ~v + w) (v + x + ~y)



K = 3

Register Allocation

- Liveness: A variable is live if its current assignment may be used at some future point in a program's flow
- Optimizers often try to keep live variables in registers
- If two variables are simultaneously live, they need to be kept in separate registers
- Consider the K-coloring problem (can the nodes of a graph be colored with at most K colors under the constraint that adjacent nodes must have different colors?)
- Register Allocation reduces to K-coloring by mapping each variable to a node and inserting an edge between variables that are simultaneously live
- K-coloring reduces to Register Allocation by interpreting nodes as variables and edges as indicating concurrent liveness
- This is a simple mapping because it's an isomorphism

PROCESSOR SCHEDULING IS NP-HARD

Processor Scheduling

- A Process Scheduling Problem can be described by
  - **m processors $P_1, P_2, \ldots, P_m$,**
  - **processor timing functions $S_1, S_2, \ldots, S_m$, each describing how the corresponding processor responds to an execution profile,**
  - **additional resources $R_1, R_2, \ldots, R_k$, e.g., memory**
  - **transmission cost matrix C_{ij} ($1 \le i , j \le m$), based on proc. data sharing,**
  - **tasks to be executed $T_1, T_2, \ldots, T_n$,**
  - **task execution profiles $A_1, A_2, \ldots, A_n$,**
  - **a partial order defined on the tasks such that $T_i < T_j$ means that T_i must complete before T_j can start execution,**
  - **communication matrix D_{ij} ($1 \le i , j \le n$); D_{ij} can be non-zero only if $T_i < T_j$,**
  - **weights $W_1, W_2, \ldots, W_n$ -- cost of deferring execution of task.**

Complexity Overview

- The intent of a scheduling algorithm is to minimize the sum of the weighted completion times of all tasks, while obeying the constraints of the task system. Weights can be made large to impose deadlines.
- The general scheduling problem is quite complex, but even simpler instances, where the processors are uniform, there are no additional resources, there is no data transmission, the execution profile is just processor time and the weights are uniform, are very hard.
- In fact, if we just specify the time to complete each task and we have no partial ordering, then finding an optimal schedule on two processors is an NP-complete problem. It is essentially the subset-sum problem.

2 Processor Scheduling

The problem of optimally scheduling n tasks $T_1, T_2, \ldots, T_n$ onto 2 processors with an empty partial order < is the same as that of dividing a set of positive whole numbers into two subsets, such that the numbers are as close to evenly divided. So, for example, given the numbers

3, 2, 4, 1

we could try a "greedy" approach as follows:

put 3 in set 1

put 2 in set 2

put 4 in set 2 (total is now 6)

put 1 in set 1 (total is now 4)

This is not the best solution. A better option is to put 3 and 2 in one set and 4 and 1 in the other. Such a solution would have been attained if we did a greedy solution on a sorted version of the original numbers. In general, however, sorting doesn't work.

2 Processor Nastiness

Try the unsorted list (2-1/m)

7, 7, 6, 6, 5, 4, 4, 5, 4

Greedy (Always in one that is least used)

7, 6, 5, 5 = 23

7, 6, 4, 4, 4 = 25

Optimal

7, 6, 6, 5 = 24

7, 4, 4, 4, 5 = 24

Sort it (non-increasing) (4/3-1/3m)

7, 7, 6, 6, 5, 5, 4, 4, 4

7, 6, 5, 4, 4 = 26

7, 6, 5, 4 = 22

Sort it (non-decreasing) (2-1/m)

4, 4, 4, 5, 5, 6, 6, 7, 7

4, 4, 5, 6, 7 = 26

4, 5, 6, 7 = 22

Both sorts are even worse than greedy unsorted !! (not a general result)

Challenge Problem

Consider the simple scheduling problem where we have a set of independent tasks running on a fixed number of processors, and we wish to minimize finishing time.

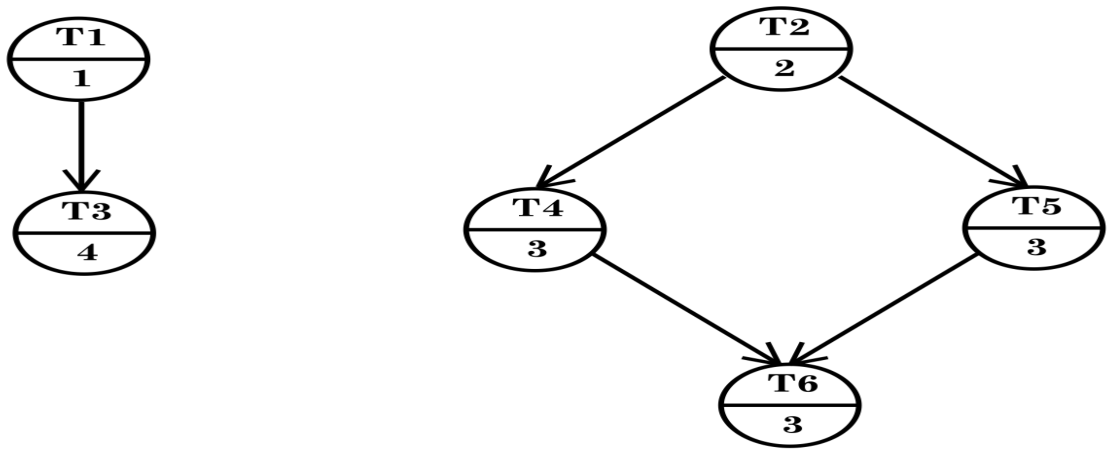
How would a list (first fit, no preemption) strategy schedule tasks with the following IDs and execution times onto four processors? Answer using Gantt chart.

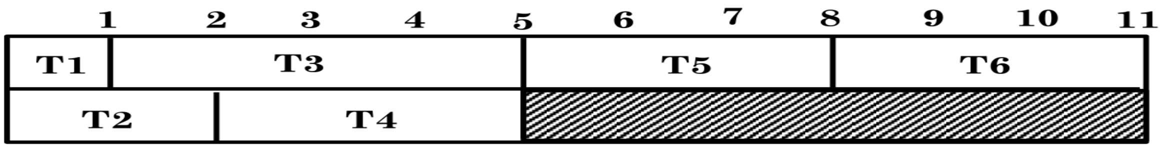
(T1,4) (T2,1) (T3,3) (T4,6) (T5,2) (T6,1) (T7,4) (T8,5) (T9,7) (T10,3) (T11,4) (2-1/m)

Now show what would happen if the times were sorted non-decreasing. **(2-1/m)**

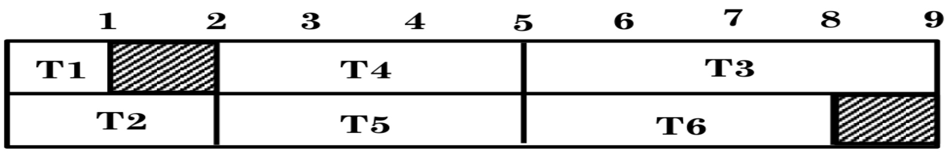
Now show what would happen if the times were sorted non-increasing. **(4/3-1/3m)**

2 Processor with partial order

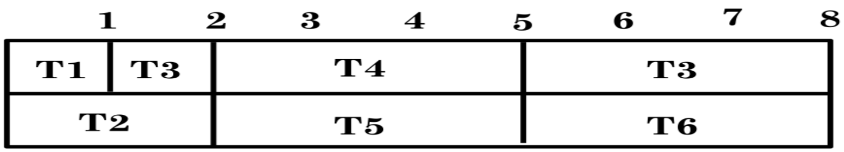




List Schedule with L = {T1,T2,T3,T4,T5,T6}

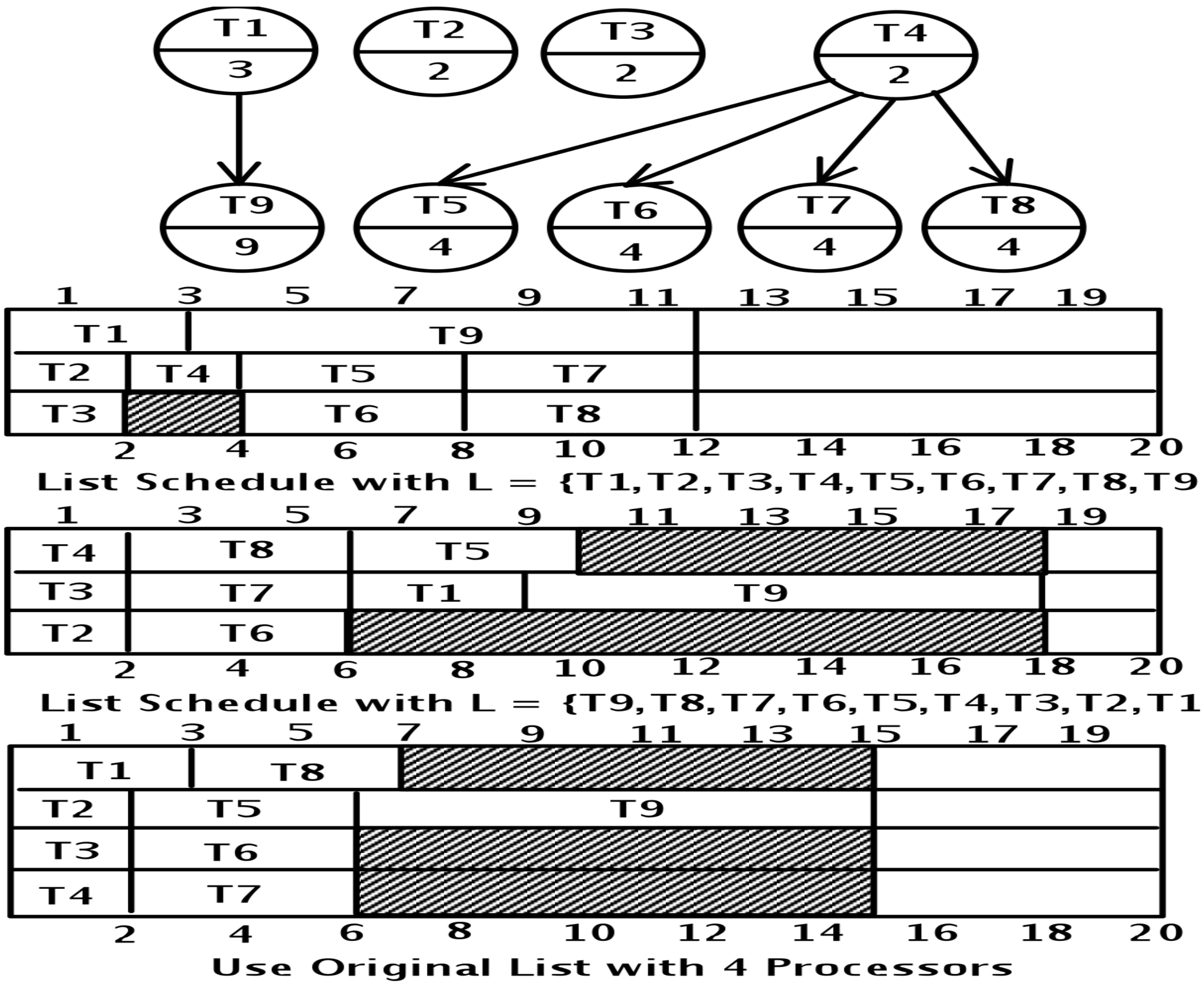


Non-Preemptive, Delays Allowed

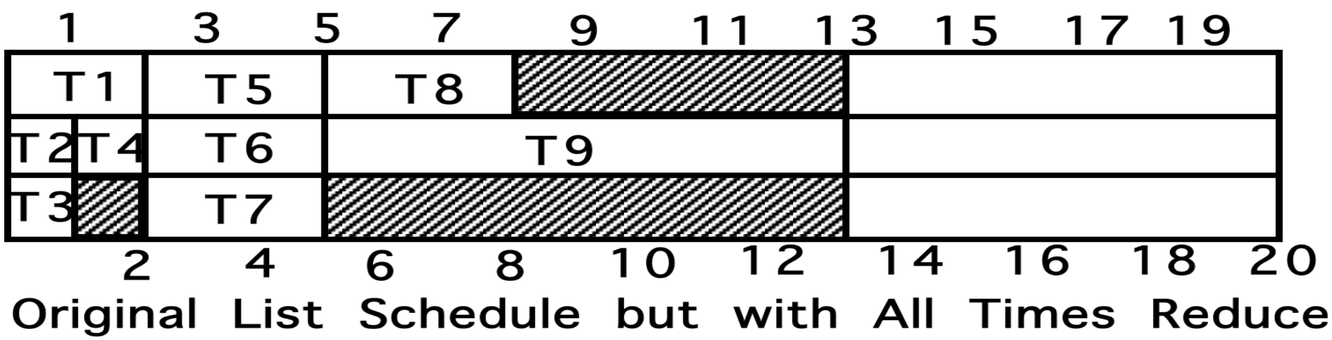


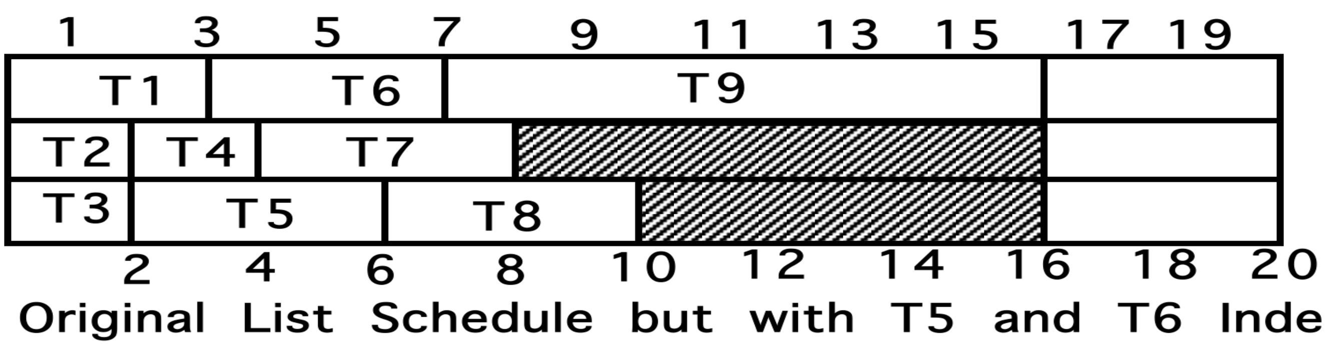
Preemptive

Anomalies everywhere



More anomalies





Critical path or level strategy

A UET is a Unit Execution Tree. Our Tree is funny. It has a single leaf by standard graph definitions.

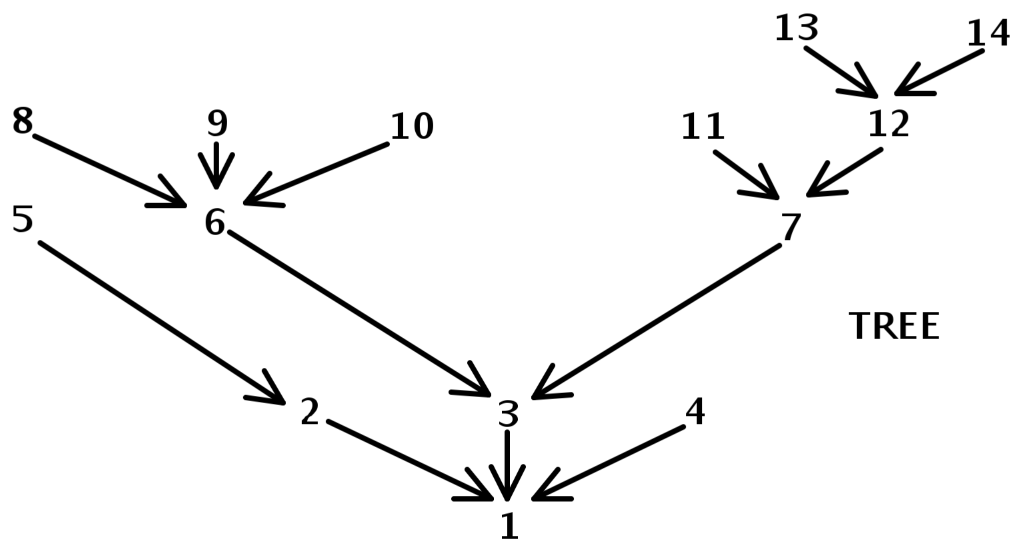
1. Assign L(T) = 1, for the leaf task T
2. Let labels 1, …, k-1 be assigned. If T is a task with lowest numbered immediate successor then define L(T) = k (non-deterministic)

   This is an order n labeling algorithm that can easily be implemented using a breadth first search.

Note: This can be used for a forest as well as a tree. Just add a new leaf. Connect all the old leafs to be immediate successors of the new one. Use the above to get priorities, starting at 0, rather than 1. Then delete the new node completely.

Note: This whole thing can also be used for anti-trees. Make a schedule, read it backwards. You cannot just reverse priorities.

Level strategy and UET



14	12	8	6	3	1
13	10	7	4		
11	9	5	2		

M=3

Theorem: Level Strategy is optimal for unit execution, m arbitrary, forest precedence

Level – DAG with unit time

1. Assign L(T) = 1, for an arbitrary leaf task T
2. Let labels 1, …, k-1 be assigned. For each task T such that

   { L(T’) is defined for all T’ in Successor(T) }

   Let N(T) be decreasing sequence of set members in
   {S(T’) | T’ is in S(T)}

   Choose T* with least N(T*).
   Define L(T*) = K.

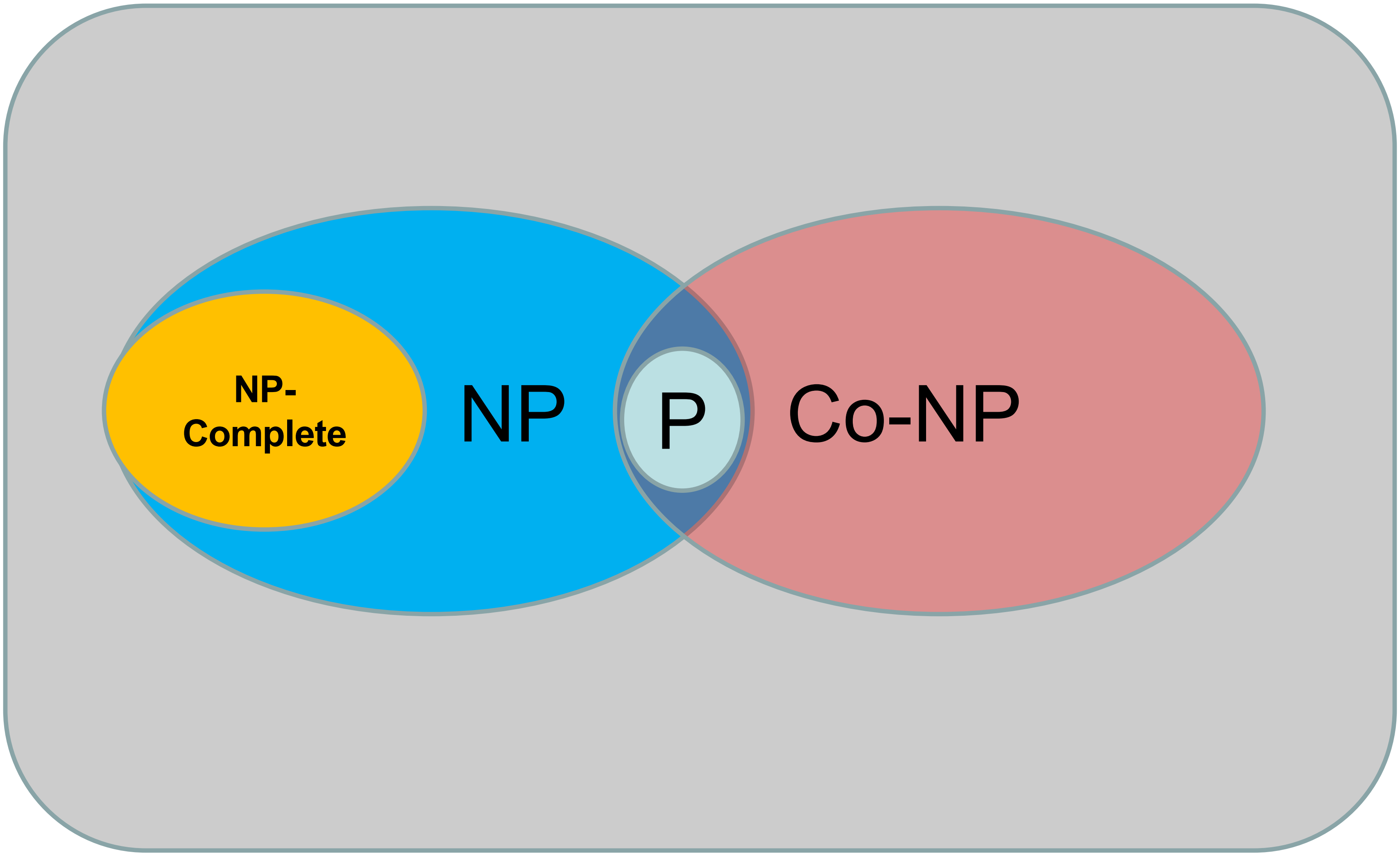
   This is an order n^2 labeling algorithm. Scheduling with it involves n union / find style operations. Such operations have been shown to be implementable in nearly constant time using an “amortization” algorithm.

Theorem: Level Strategy is optimal for unit execution, m=2, dag precedence.

Assignment#5

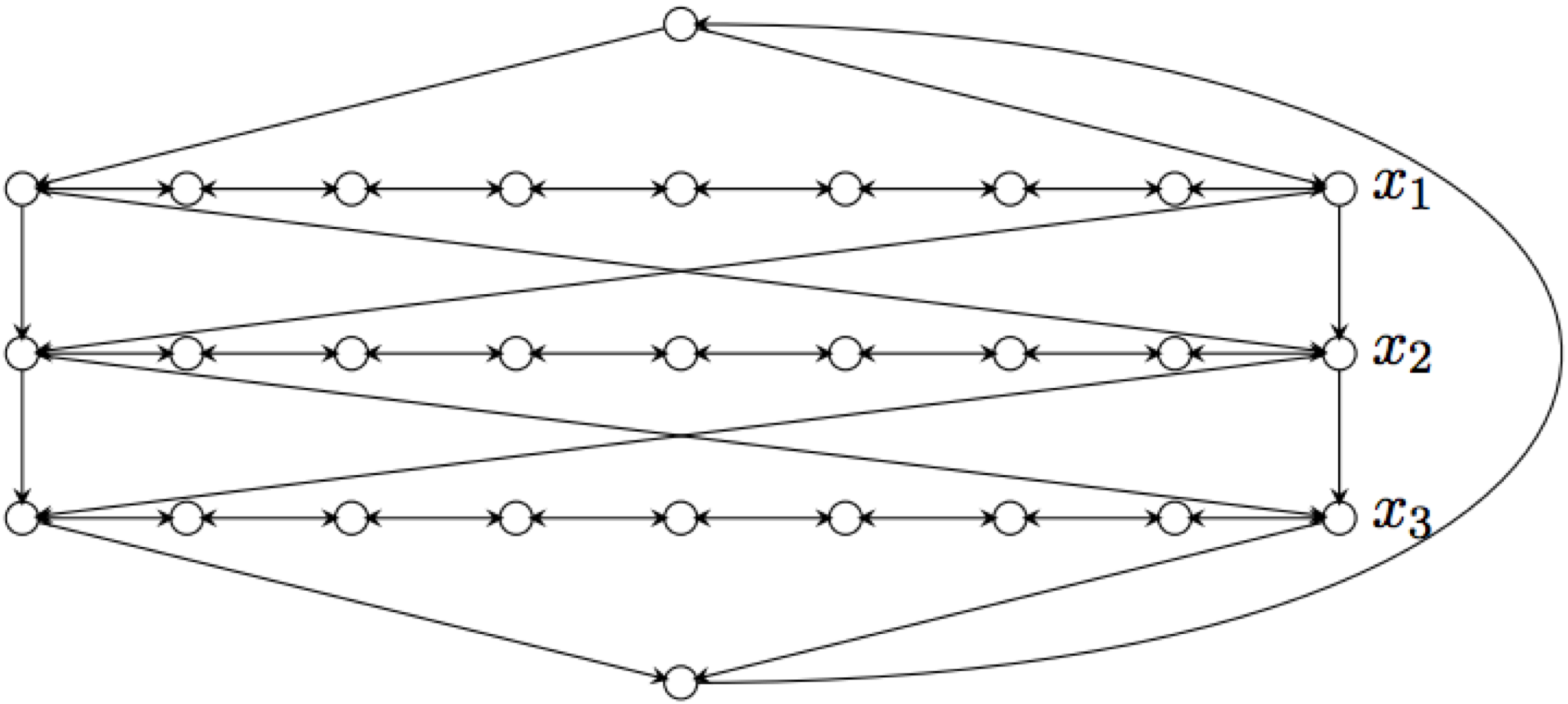
Looking back at page 678, consider adding two additional tasks numbered 15 and 16 that are siblings of 13 and 14. These four tasks must be completed before 12 is started.
a) Show the Gantt chart that reflects the new schedule associated with this enhanced tree
b) Show the Gantt chart that is associated with the corresponding anti-tree, in which all arcs are turned in the opposite direction. Use the technique of reversing the schedule from (a)
c) Show the Gantt chart associated with the anti-tree of b), where we now use the priorities obtained by treating lower numbered tasks as higher priority ones
d) Comment on the results seen in (b) versus (c), providing insight as to why they are different and why one is better than the other.

UNIVERSE OF SETS



HAMILTONIAN CIRCUIT (HC) DECISION PROBLEM IS NP-HARD

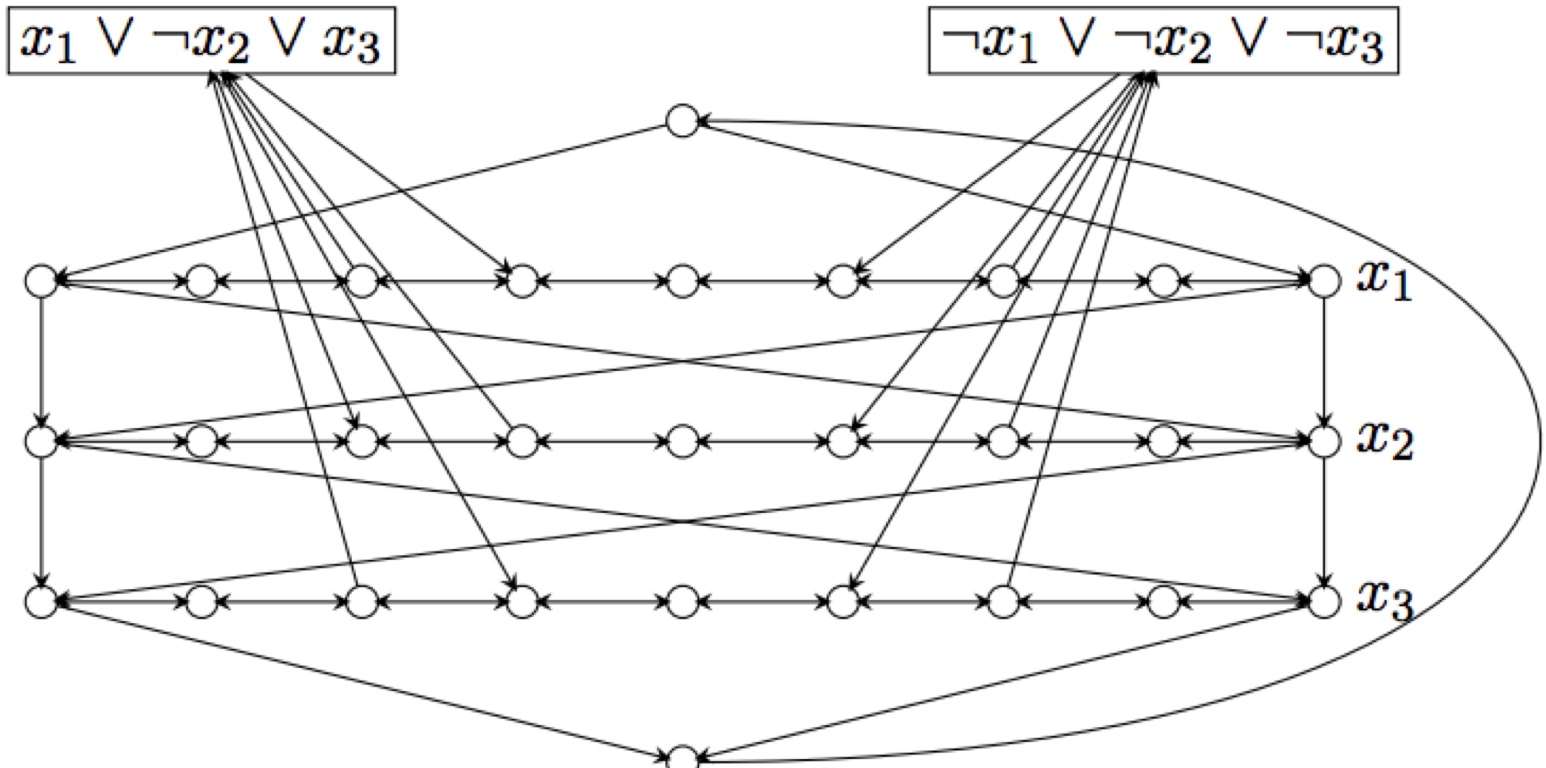
HC Variable Gadget



This has many Hamiltonian Circuits

HC Gadgets Combined

We will set convention on x_i true to be left to right and x_i false to be right to left (can fix for opposite)



This has a Hamiltonian Circuit iff all clauses are satisfied with consistent assignments to each variable

Hamiltonian Path

- Note we can split an arbitrary node, v, into two (v’,v’’) – one, v’, has in-edges of v, other, v’’, has out-edges. Path (not cycle) must start at v’’ and end at v’ and goal is still K (the number of vertices).

Travelling Salesman

- Start with HC = (V,E), K=|V|
- Set edges from HC instance to 1
- Add edges between pairs that lack such edges and make those weights 2 (often people make these K+1); this means that the reverse of unidirectional links also get weight 2
- Goal weight is K for cycle

Knapsack 0-1 Problem

- The goal is to **maximize the value of a knapsack** that can hold at most W units (i.e. lbs or kg) worth of goods from a list of items $I_0, I_1, \ldots I_{n-1}$.
  - Each item has 2 attributes:
    1) Value – let this be v_i for item I_i
    2) Weight – let this be w_i for item I_i

Thanks to Arup Guha

Knapsack 0-1 Problem

- The difference between this problem and the fractional knapsack one is that you CANNOT take a fraction of an item.
  - You can either take it or not.
  - Hence the name Knapsack 0-1 problem.

Knapsack Optimize vs Decide

- As stated, the Knapsack problem is an optimization problem.
- We can restate as decision problem to determine if there exists a set of items , each with weight < W, that reaches some fixed goal value, W.

Knapsack and SubsetSum

- Let $v_i = w_i$ for each item I_i.
- By doing so, the value is maximized when the Knapsack is filled as close to capacity.
- The related decision problem is to determine if we can attain capacity (W).
- Clearly then, given an instance of the SubsetSum problem, we can create an instance of the Knapsack decision problem problem, such that we reach the goal sum, G, iff we can attain a Knapsack value of G.

Knapsack Decision Problem

- The reduction from SubsetSum shows that the Knapsack decision problem is at least as hard as SubsetSum, so it is NP-Complete if it is in NP.
- Think about whether or not it is in NP.
- Now, think about optimization problem.

Related Bin Packing

- Have a bin capacity of B.
- Have item set $\mathbf{S = \{s_1, s_2, \ldots, s_n\}}$
- Use all items in **S**, minimizing the number of bins, while adhering to the constraint that any such subset must sum to **B** or less.
- This is similar to the processor scheduling problem without constraints, except we optimize on number of processors, not finishing time for all tasks. It is NP-Hard (WHY?)

Knapsack 0-1 Problem

- Brute Force
  - The naïve way to solve the 0-1 Knapsack problem is to cycle through all 2^n subsets of the n items and pick the subset with a legal weight that maximizes the value of the knapsack.
  - We can come up with a dynamic programming algorithm that is USUALLY faster than this brute force technique.

Knapsack 0-1 Problem

- We are going to solve the problem in terms of sub-problems.
- Our first attempt might be to characterize a sub-problem as follows:
  - Let S_k be the optimal subset of elements from $\{I_0, I_1, \ldots, I_k\}$.
    - What we find is that the optimal subset from the elements $\{I_0, I_1, \ldots, I_{k+1}\}$ may not correspond to the optimal subset of elements from $\{I_0, I_1, \ldots, I_k\}$ in any regular pattern.
  - Basically, the solution to the optimization problem for S_{k+1} might NOT contain the optimal solution from problem S_k.

Knapsack 0-1 Problem

- Let's illustrate that point with an example:

Item	Weight	Value
I_0	3	10
I_1	8	4
I_2	9	9
I_3	8	11

- **<u>The maximum weight the knapsack can hold is 20.</u>**

- The best set of items from $\{I_0, I_1, I_2\}$ is $\{I_0, I_1, I_2\}$
- BUT the best set of items from $\{I_0, I_1, I_2, I_3\}$ is $\{I_0, I_2, I_3\}$.
  - In this example, note that this optimal solution, $\{I_0, I_2, I_3\}$, does NOT build upon the previous optimal solution, $\{I_0, I_1, I_2\}$.
    - (Instead it builds upon the solution, $\{I_0, I_2\}$, which is really the optimal subset of $\{I_0, I_1, I_2\}$ with weight 12 or less.)

Knapsack 0-1 problem

- So now we must re-work the way we build upon previous sub-problems…
  - Let **B[k, w]** represent the maximum total value of a subset S_k with weight w.
  - Our goal is to find **B[n, W],** where n is the total number of items and W is the maximal weight the knapsack can carry.

- So our recursive formula for subproblems:

$$B[k, w] = B[k - 1, w], \text{ if } w_k > w$$
$$= \max \{ B[k - 1, w], B[k - 1, w - w_k] + v_k \}, \text{ otherwise}$$

- In English, this means that the best subset of S_k that has total weight w is:
  1) The best subset of S_{k-1} that has total weight w, or
  2) The best subset of S_{k-1} that has total weight w-w_k plus the item k

Knapsack 0-1 Problem – Recursive Formula

$$B[k, w] = \begin{cases} B[k-1, w], & \underline{\text{if } w_k > w} \\ \max\{B[k-1, w], B[k-1, w-w_k] + v_k\}, & \underline{\text{otherwise}} \end{cases}$$

- The best subset of S_k that has the total weight w, either contains item k or not.

- <u>**First case:**</u> $w_k > w$
  - Item *k* can't be part of the solution! If it was the total weight would be > w, which is unacceptable.

- <u>**Second case:**</u> $w_k \leq w$
  - Then the item *k* <u>can</u> be in the solution, and we choose the case with greater value.

Knapsack 0-1 Algorithm

```
for w = 0 to W {  // Initialize 1st row to 0's
   B[0,w] = 0
}
for i = 1 to n {  // Initialize 1st column to 0's
   B[i,0] = 0
}
for i = 1 to n {
   for w = 0 to W {
        if wi <= w {  //item i can be in the solution
             if vi + B[i-1,w-wi] > B[i-1,w]
                  B[i,w] = vi + B[i-1,w- wi]
             else
                  B[i,w] = B[i-1,w]
        }
        else B[i,w] = B[i-1,w] // wi > w
   }
}
```

Knapsack 0-1 Problem

- Let's run our algorithm on the following data:
  - n = 4 (# of elements)
  - W = 5 (max weight)
  - Elements (weight, value):
    (2,3), (3,4), (4,5), (5,6)

Knapsack 0-1 Example

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0					
2	0					
3	0					
4	0					

// Initialize the base cases

for w = 0 to W

  B[0,w] = 0

for i = 1 to n

  B[i,0] = 0

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0				
2	0					
3	0					
4	0					

$i = 1$
$v_i = 3$
$w_i = 2$
$w = 1$
$w - w_i = -1$

if $w_i <= w$ //item i can be in the solution
  if $v_i + B[i-1,w-w_i] > B[i-1,w]$
    $B[i,w] = v_i + B[i-1,w- w_i]$
  else
    $B[i,w] = B[i-1,w]$
else **$B[i,w] = B[i-1,w]$** // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3			
2	0					
3	0					
4	0					

$i = 1$
$v_i = 3$
$w_i = 2$
$\mathbf{w = 2}$
$w\text{-}w_i = 0$

if $w_i \le w$ //item i can be in the solution
    if $v_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = v_i + B[i-1,w-w_i]$
    else
        $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3		
2	0					
3	0					
4	0					

$i = 1$
$v_i = 3$
$w_i = 2$
$\mathbf{w = 3}$
$w\text{-}w_i = 1$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	
2	0					
3	0					
4	0					

$i = 1$
$v_i = 3$
$w_i = 2$
$\mathbf{w = 4}$
$w\text{-}w_i = 2$

if $w_i <= w$ //item i can be in the solution
    if $v_i + B[i-1,w-w_i] > B[i-1,w]$
        $\mathbf{B[i,w] = v_i + B[i-1,w- w_i]}$
    else
        $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0					
3	0					
4	0					

$i = 1$
$v_i = 3$
$w_i = 2$
$\mathbf{w = 5}$
$w\text{-}w_i = 3$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:

1: (2,3)

2: (3,4)

3: (4,5)

4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0				
3	0					
4	0					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 1}$

$w\text{-}w_i = -2$

if $w_i <= w$ //item i can be in the solution

    if $v_i + B[i-1,w-w_i] > B[i-1,w]$

        $B[i,w] = v_i + B[i-1,w- w_i]$

    else

        $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3			
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 2}$
$w\text{-}w_i = -1$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else **B[i,w] = B[i-1,w]** // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4		
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 3}$
$w - w_i = 0$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i-1,w-w_i] > B[i-1,w]$

$B[i,w] = v_i + B[i-1,w- w_i]$

else

$B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 4}$
$w - w_i = 1$

if $w_i \le w$ //item i can be in the solution

    if $v_i + B[i-1, w-w_i] > B[i-1, w]$

        $\mathbf{B[i,w] = v_i + B[i-1, w- w_i]}$

    else

        $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0					
4	0					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 5}$
$w\text{-}w_i = 2$

if $w_i <= w$ //item i can be in the solution
 if $v_i + B[i-1,w-w_i] > B[i-1,w]$
  $\mathbf{B[i,w] = v_i + B[i-1,w- w_i]}$
 else
  $B[i,w] = B[i-1,w]$
else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4		
4	0					

$i = 3$
$v_i = 5$
$w_i = 4$
$\mathbf{w = 1..3}$
$w\text{-}w_i = -3..-1$

if $w_i <= w$ //item i can be in the solution
    if $v_i + B[i-1,w-w_i] > B[i-1,w]$
        $B[i,w] = v_i + B[i-1,w- w_i]$
    else
        $B[i,w] = B[i-1,w]$
else **$B[i,w] = B[i-1,w]$** // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	
4	0					

$i = 3$
$v_i = 5$
$w_i = 4$
$\mathbf{w = 4}$
$w\text{-}w_i = 0$

if $w_i <= w$ //item i can be in the solution

  if $v_i + B[i-1,w-w_i] > B[i-1,w]$

    $\mathbf{B[i,w] = v_i + B[i-1,w- w_i]}$

  else

    $B[i,w] = B[i-1,w]$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0					

$i = 3$
$v_i = 5$
$w_i = 4$
$\mathbf{w = 5}$
$w\text{-}w_i = 1$

if $w_i <= w$ //item i can be in the solution

  if $v_i + B[i-1,w-w_i] > B[i-1,w]$

    $B[i,w] = v_i + B[i-1,w- w_i]$

  else

    $\mathbf{B[i,w] = B[i-1,w]}$

else $B[i,w] = B[i-1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
$\boxed{\text{4: (5,6)}}$

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	

$i = 4$
$v_i = 6$
$w_i = 5$
$w = 1..4$
$w\text{-}w_i = -4..-1$

if $w_i <= w$ //item i can be in the solution
  if $v_i + B[i-1,w-w_i] > B[i-1,w]$
    $B[i,w] = v_i + B[i-1,w- w_i]$
  else
    $B[i,w] = B[i-1,w]$
else **$B[i,w] = B[i-1,w]$** // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
$\boxed{\text{4: (5,6)}}$

i / w	**0**	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

$i = 4$
$v_i = 6$
$w_i = 5$
$\mathbf{w = 5}$
$w\text{-}w_i = 0$

if $w_i <= w$ //item i can be in the solution

if $v_i + B[i\text{-}1,w\text{-}w_i] > B[i\text{-}1,w]$

$B[i,w] = v_i + B[i\text{-}1,w\text{-} w_i]$

else

$B[i,w] = B[i\text{-}1,w]$

else $B[i,w] = B[i\text{-}1,w]$ // $w_i > w$

Knapsack 0-1 Example

Items:
1: (2,3)
2: (3,4)
3: (4,5)
4: (5,6)

i / w	0	1	2	3	4	5
0	0	0	0	0	0	0
1	0	0	3	3	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	7

We're DONE!!

The max possible value that can be carried in this knapsack is ***$7***

Knapsack 0-1 Problem – Run Time

```
for w = 0 to W
    B[0,w] = 0
```

O(*W*)

```
for i = 1 to n
    B[i,0] = 0
```

O(*n*)

```
for i = 1 to n
    for w = 0 to W
        < the rest of the code >
```

Repeat *n* times

O(*W*)

What is the running time of this algorithm?

O(*n*W*) – *of course, W can be mighty big*

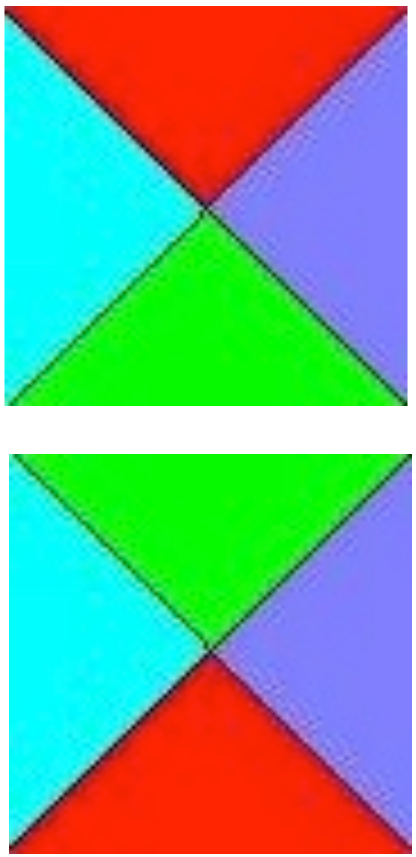
What is an analogy in world of sorting?

Remember that the brute-force algorithm takes: **O(2^n)**

Tiling

Undecidable and NP-Complete Variants

Basic Idea of Tiling

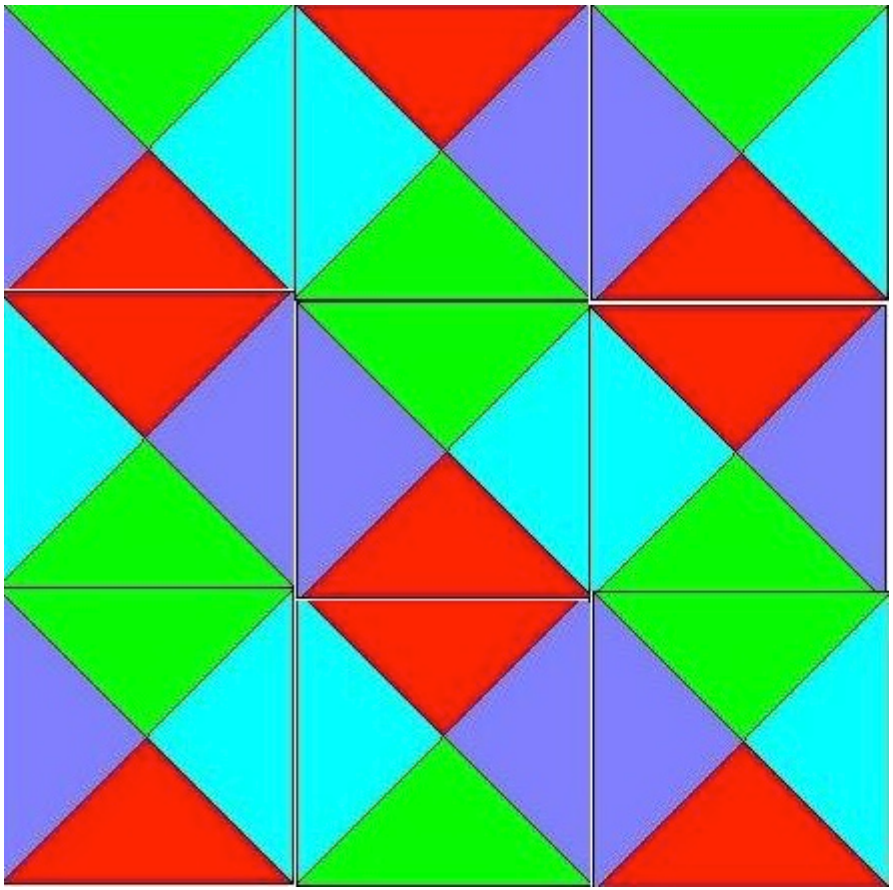


A single tile has colors on all four sides. Tiles are often called dominoes as assembling them follows the rules of placing dominoes. That is, the color (or number) of a side must match that of its adjacent tile, e.g., tile, t2, to right of a tile, t1, must have same color on Its left as is on the right side of t1. This constraint applies to top and as well as sides. Boundary tiles do not have constraints on their sides that touch the boundaries.

Instance of Tiling Problem

- A finite set of tile types (a type is determined by the colors of its edges)
- Some 2d area (finite or infinite) on which the tiles are to be laid out
- An optional starting set of tiles in fixed positions
- The goal of tiling the plane following the adjacency constraints and whatever constraints are indicated by the starting configuration.

A Valid 3 by 3 Tiling of Tile Types from Previous Slide



Some Variations

- Infinite 2d plane (impossible, co-re-non-rec) in general)
  - Our two tile types can easily tile the 2d plane
- Finite 2d plane (hard in general)
  - Our two tile types can easily tile any finite 2d plane
  - This is called the Bounded Tiling Problem.

- One dimensional space (hmm?)
- Infinite 3d space (really impossible – multiple quantifiers, in general)

Tiling the Plane

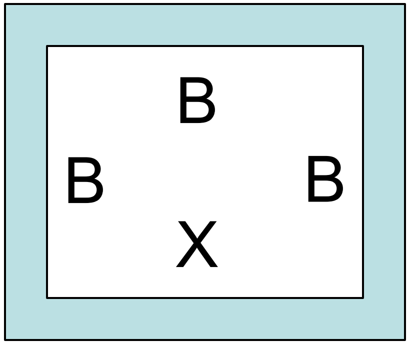
- We will start with a Post Machine, $M = (Q, \Sigma, \delta, q_0)$, with tape alphabet $\Sigma = \{B,1\}$ where B is blank and δ maps pairs from $Q \times \Sigma$ to $Q \times (\Sigma \cup \{R,L\})$. M starts in state q_0
  - (Turing Machine with each action being L, R or Print)
- We will consider the case of M starting with a blank tape
- We will constrain our machine to never go to the left of its starting position (semi unbounded tape)
- We will mimic the computation steps of M
- Termination occurs if in state q reading b and $\delta(q,b)$ is not defined
- We will use the fact that halting when starting at the left end of a semi unbounded tape in its initial state with a blank tape is undecidable

The Tiling Decision Problem

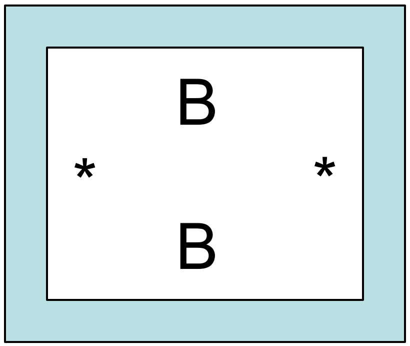
- Given a finite set of tile types and a starting tile in lower left corner of 2d plane, can we tile all places in the plane?
- A place is defined by its coordinates (x,y), $x \geq 0$, $y \geq 0$
- The fixed starting tile is at (0,0)

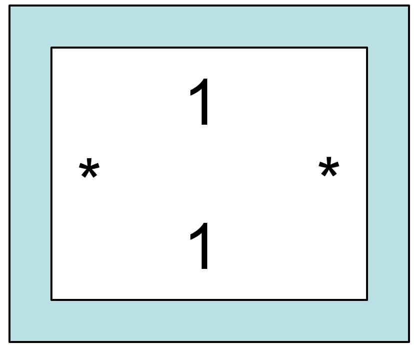
Colors

- Given M, define our tile colors as
- $\{X, Y, *, B, 1, YB, Y1\} \cup Q \times \{B,1\} \cup Q \times \{YB,Y1\} \cup Q \times \{R,L\}$
- Simplest tile (represents Blank on X axis)

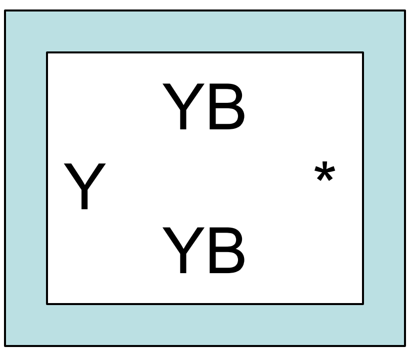


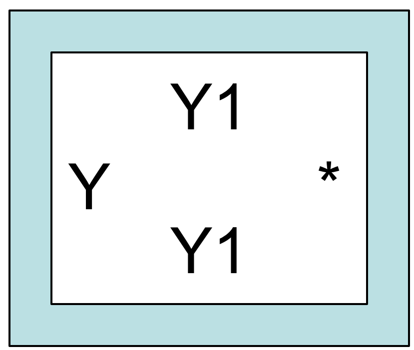
Tiles for Copying Tape Cell





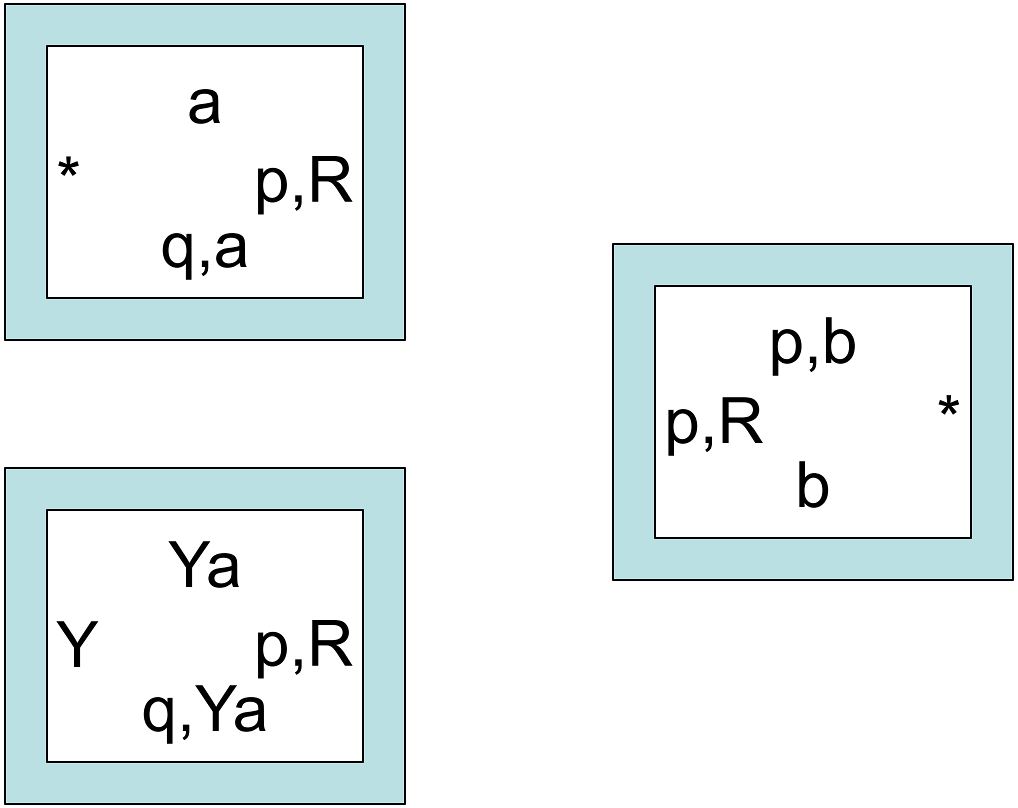
Copy cells not on left boundary and not scanned





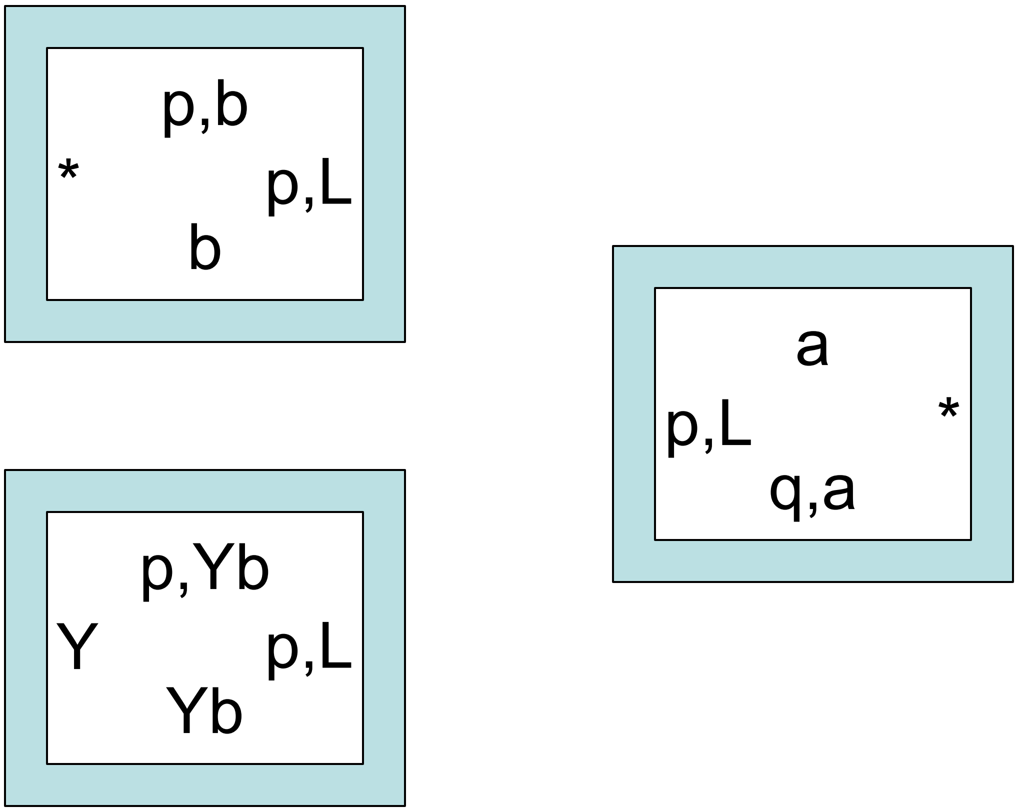
Copy cells on left boundary but not scanned

Right Move δ(q,a) = (p,R)



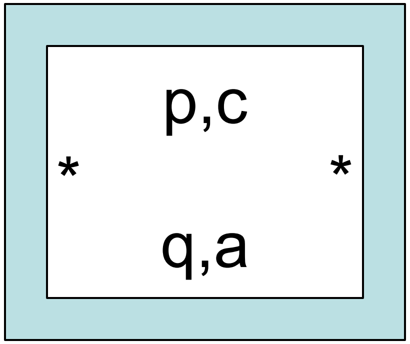
where $b \in \Sigma = \{B,1\}$

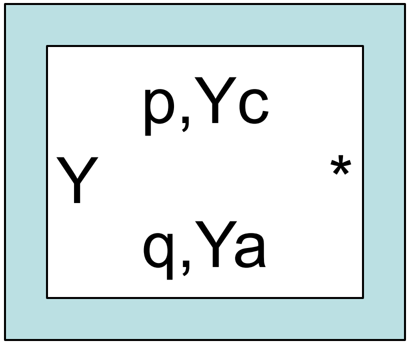
Left Move $\delta(q,a) = (p,L)$



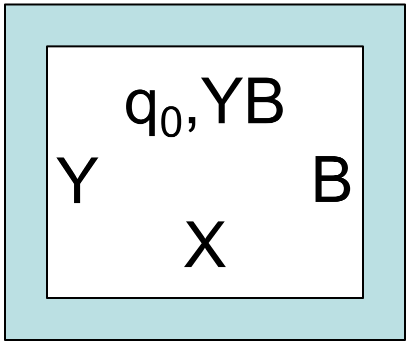
where $b \in \Sigma = \{B,1\}$

Print $\delta(q,a) = (p,c)$

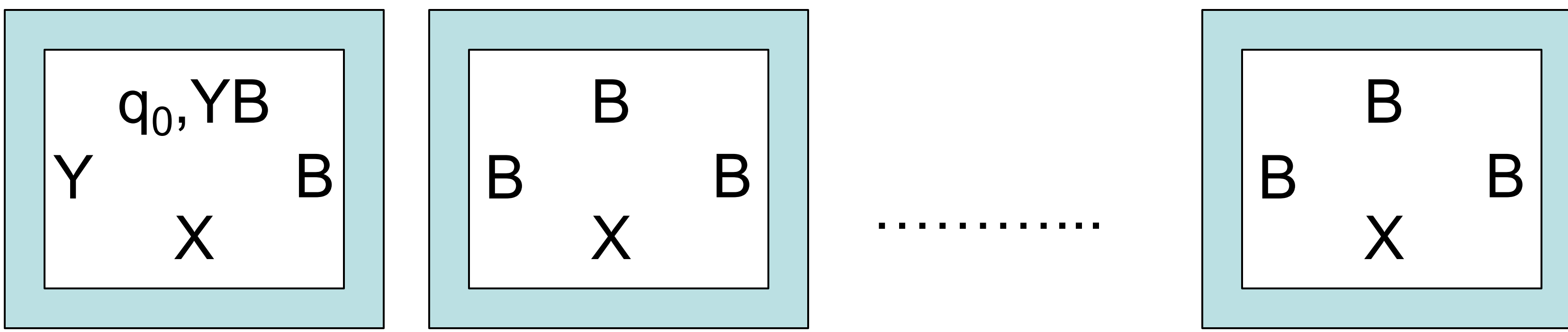




Corner Tile and Bottom Row

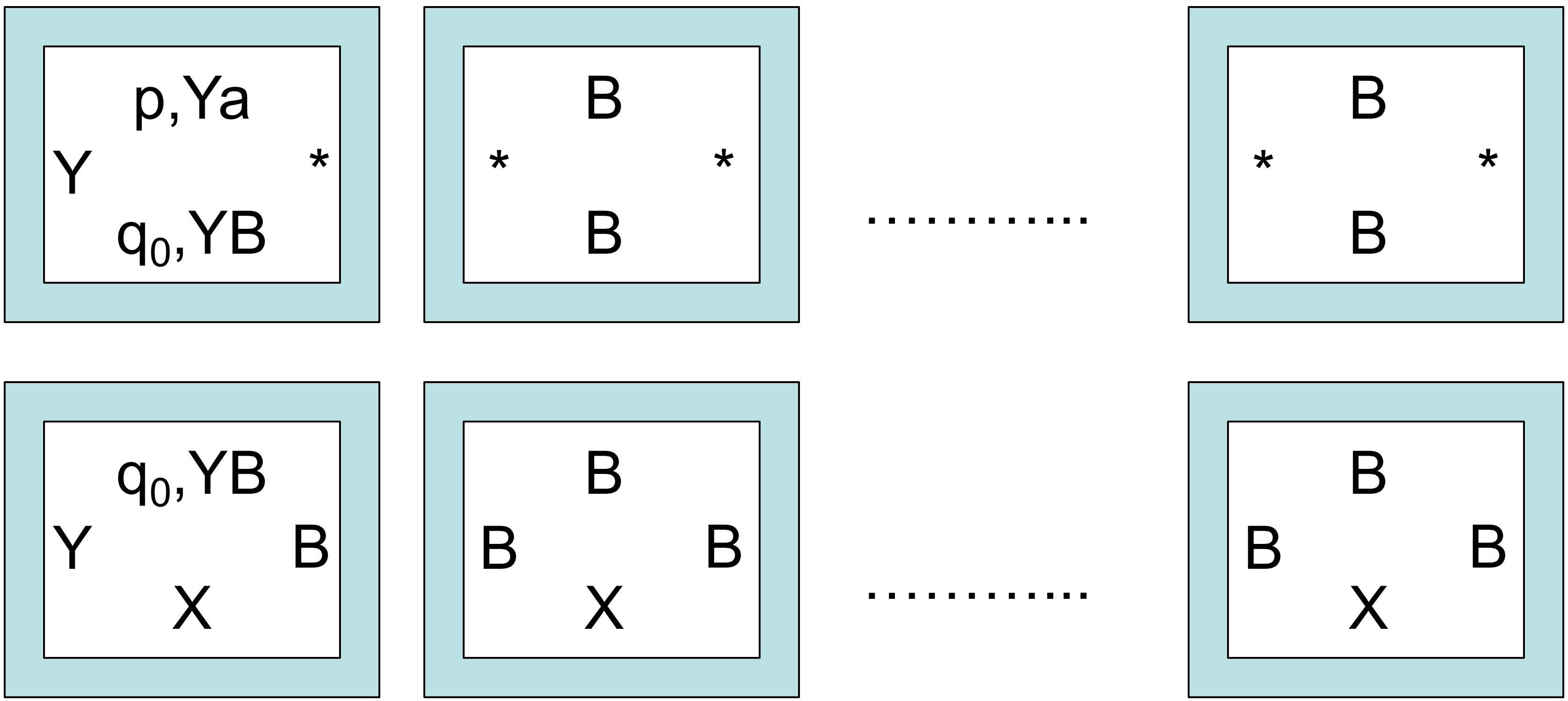


Zero-ed Row is forced to be



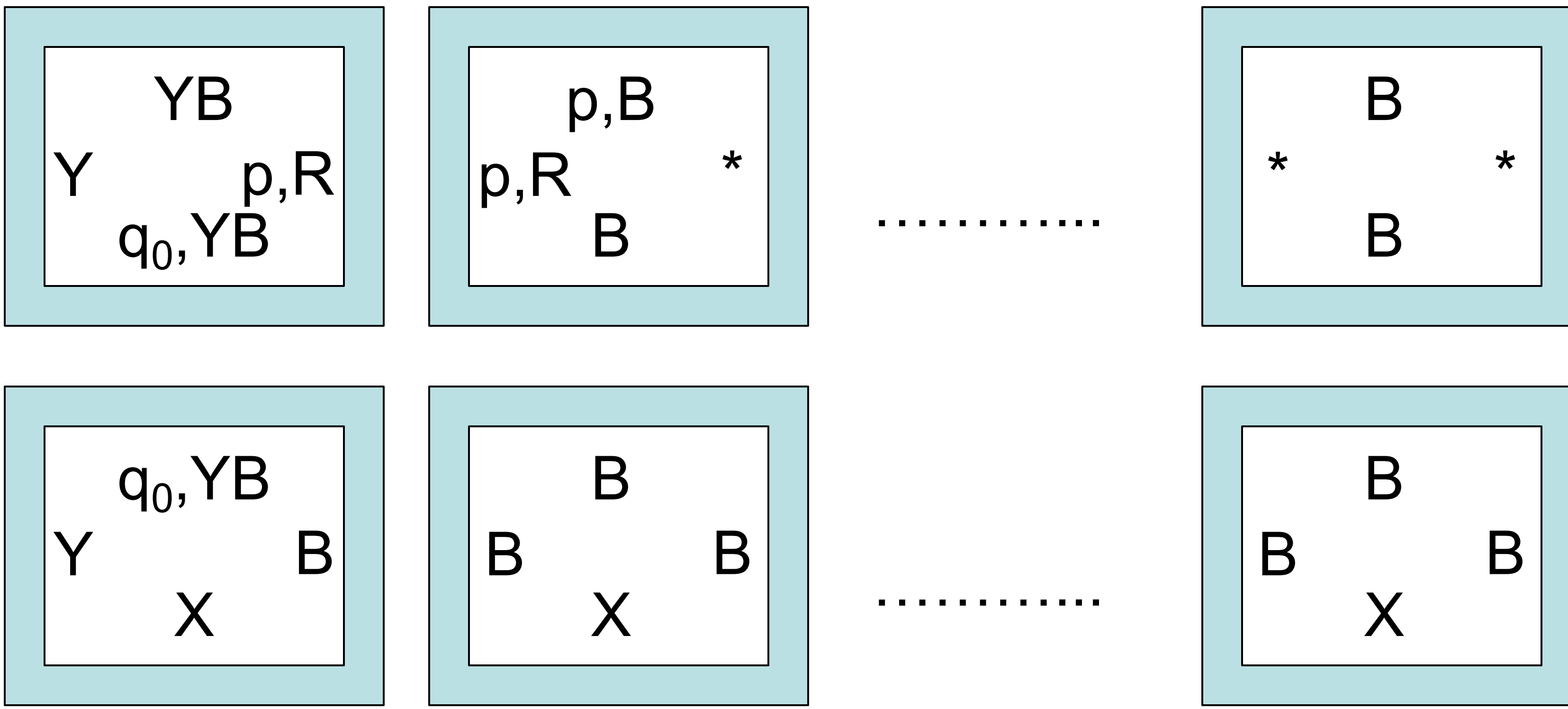
First Action Print

As we cannot move left of leftmost character first action is either right or print. Assume for now that $\delta(q_0,B) = (p,a)$



First Action Right Move

As we cannot move left of leftmost character first action is either right or print. Assume for now that $\delta(q_0,B) = (p,R)$



The Rest of the Story Part 1

- Inductively we can show that, if the i-th row represents an infinite transcription of the Turing configuration after step i then the (i+1)-st represents such a transcription after step i+1. Since we have shown the base case, we have a successful simulation.

The Rest of the Story Part 2

- Consider the case where M eventually halts when started on a blank tape in state q_0. In this case we will reach a point where no actions fill the slots above the one representing the current state. That means that we cannot tile the plane.
- If M never halts, then we can tile the plane (in the limit).

The Rest of the Story Part 3

- The consequences of Parts 1 and 2 are that Tiling the plane is as hard as the complement of the Halting problem which is co-RE Complete.
- This is not surprising as this problem involve a universal quantification over all coordinates (x,y) in the plane.

Constraints on M

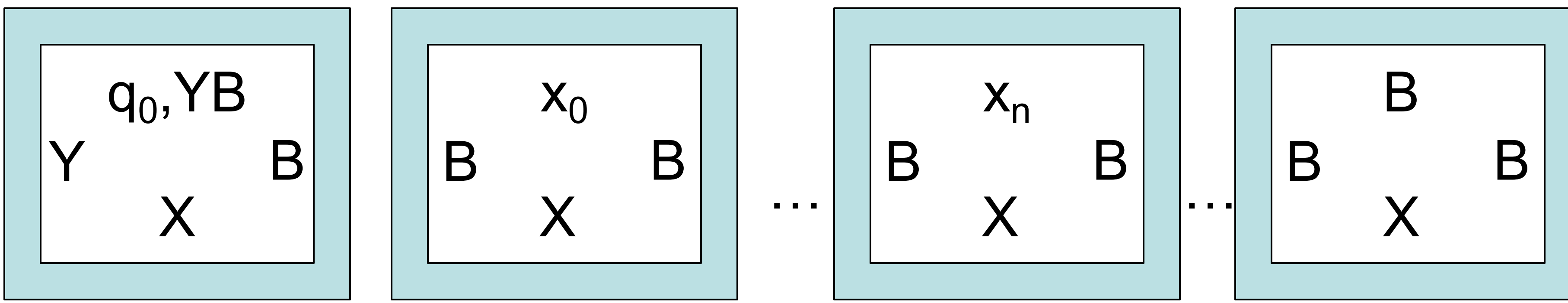
- The starting blank tape is not a real constraint as we can create M so its first actions are to write arguments on its tape.
- The semi unbounded tape is not new. If you look back at Standard Turing Computing (STC), we assumed there that we never moved left of the blank preceding our first argument.
- If you prefer to consider all computation based on the STC model then we can add to M the simple prologue $(R1)^{x1}R(R1)^{x2}R\ldots(R1)^{xk}R$ so the actual computation starts with a vector of x1 … xk on the tape and with the scanned square as the blank to right of this vector. The rest of the tape is blank.
- Think about how, in the preceding pages, you could actually start the tiling in this configuration.

Bounded Tiling Problem #1

- Consider a slight change to our machine M. First, it is non-deterministic, so our transition function maps to sets.
- Second, we add two auxiliary states $\{q_a, q_r\}$, where q_a is our only accept state and q_r is our only reject state.
- We make it so the reject state has no successor states, but the accept state always transitions back to itself rewriting the scanned square unchanged.
- We also assume our machine accepts or rejects in at most n^k steps, where n is the length of its starting input which is written immediately to the right of the initial scanned square.

Bounded Tiling Problem #2

- We limit our rows and column to be of size n^k+1. We change our initial condition of the tape to start with the input to M. Thus, it looks like



- Note that there are $n^k - n$ of these blank representations at the end. We really only need the first as the tiling constraint will force all the others to be of the same form.

Bounded Tiling Problem #3

- The finitely bounded Tiling Problem we just described mimics the operation of any given polynomially-bound non-deterministic Turing machine.
- This machine can tile the finite plane of size $(n^k+1) * (n^k+1)$ just in case the initial string is accepted in n^k or fewer steps on some path (really a trace of at most n^k).
- If the string is not accepted then we will hit a reject state on all paths and never complete tiling.
- This shows that the bounded tiling problem is NP-Hard
- Is it in NP? Yes. How? Well, we can be shown a tiling (posed solution takes space polynomial in n) and check it for completeness and consistency (this takes linear time in terms of proposed solution). Thus, we can verify the solution in time polynomial in n.

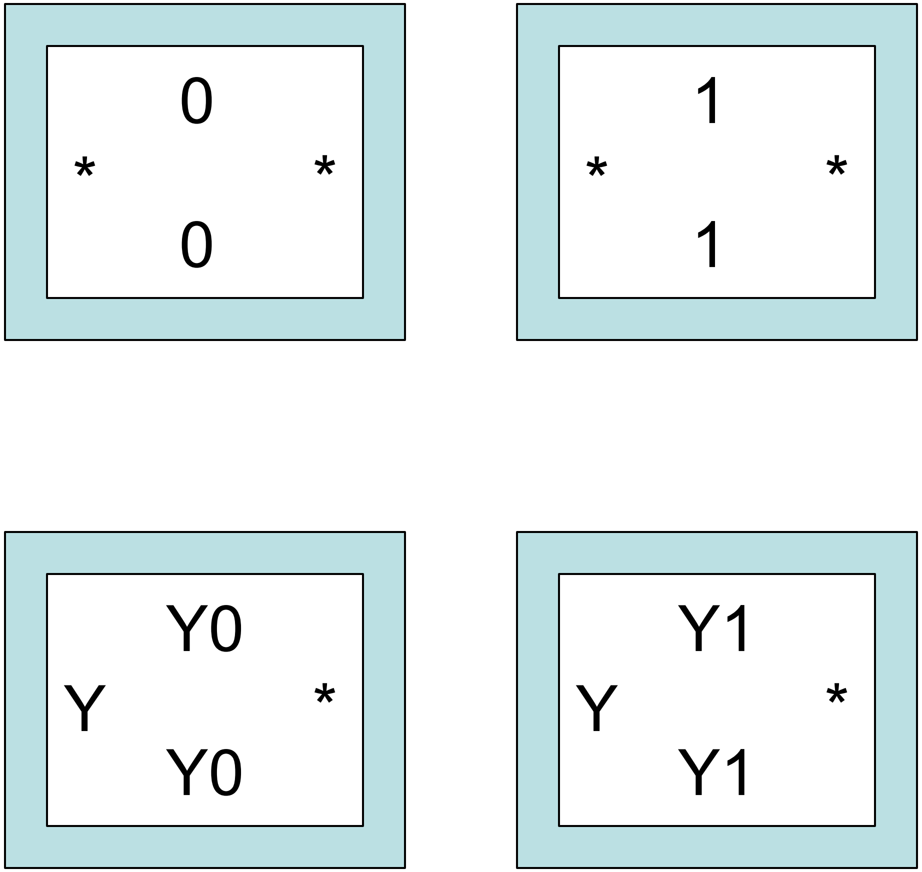
A Final Comment on Tiling

- If you look back at the unbounded version, you can see that we could have simulated a non-deterministic Turing machine there, but it would have had the problem that the plane would be tiled if any of the non-deterministic choices diverged and that is not what we desired.
- However, we need to use a non-deterministic machine for the finite case as we made this so it tiled iff some path led to acceptance. If all lead to rejection, we get stalled out on all paths as the reject state can go nowhere.

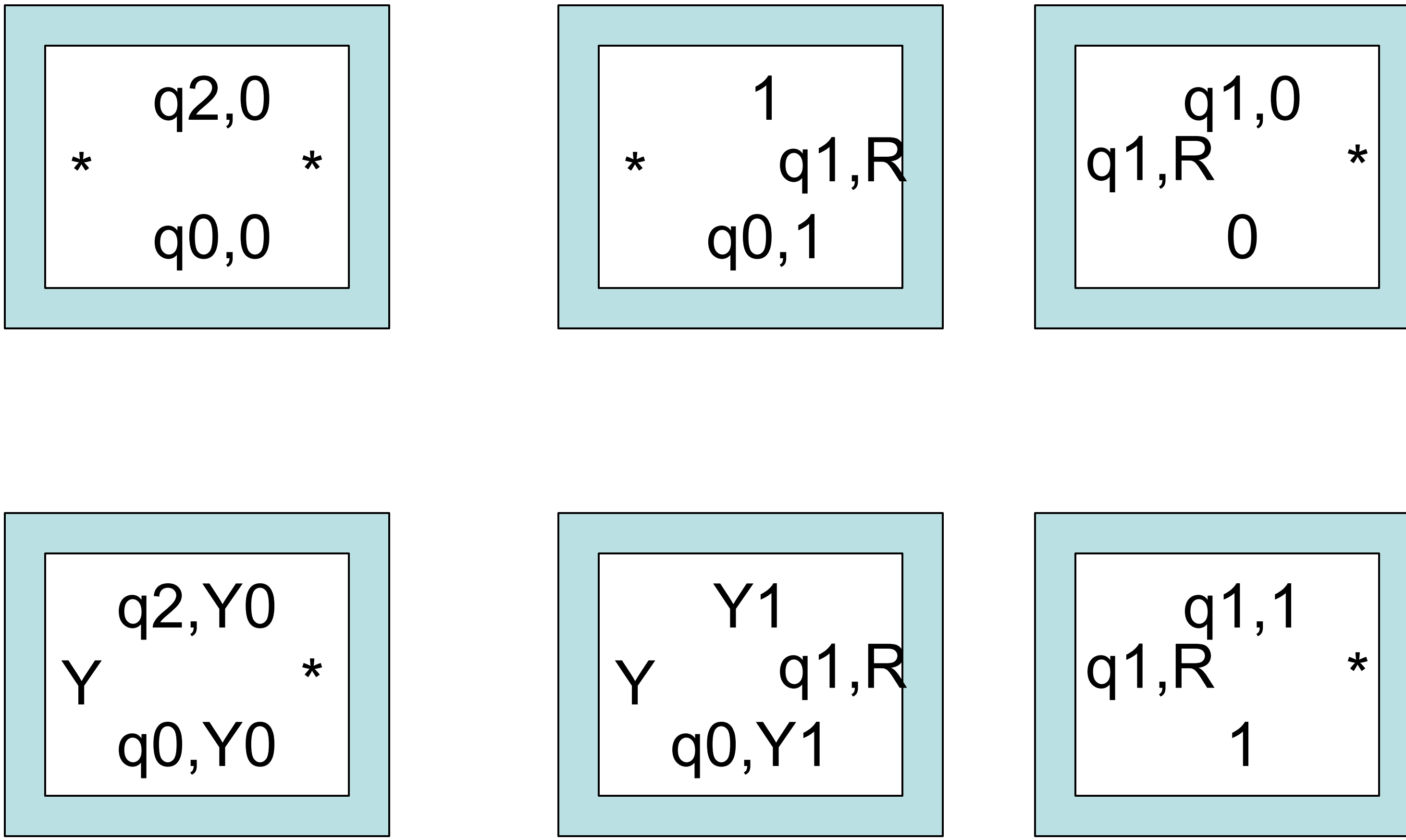
Tiling Example

- Turing Machine Recognizes strings of at least two 1's in succession.
- q0 0 0 q2
- q0 1 R q1
- q1 0 L q2
- q1 1 1 q3
- q2 0 0 q2
- q2 1 1 q2
- No q3 rules so entering here stops tiling

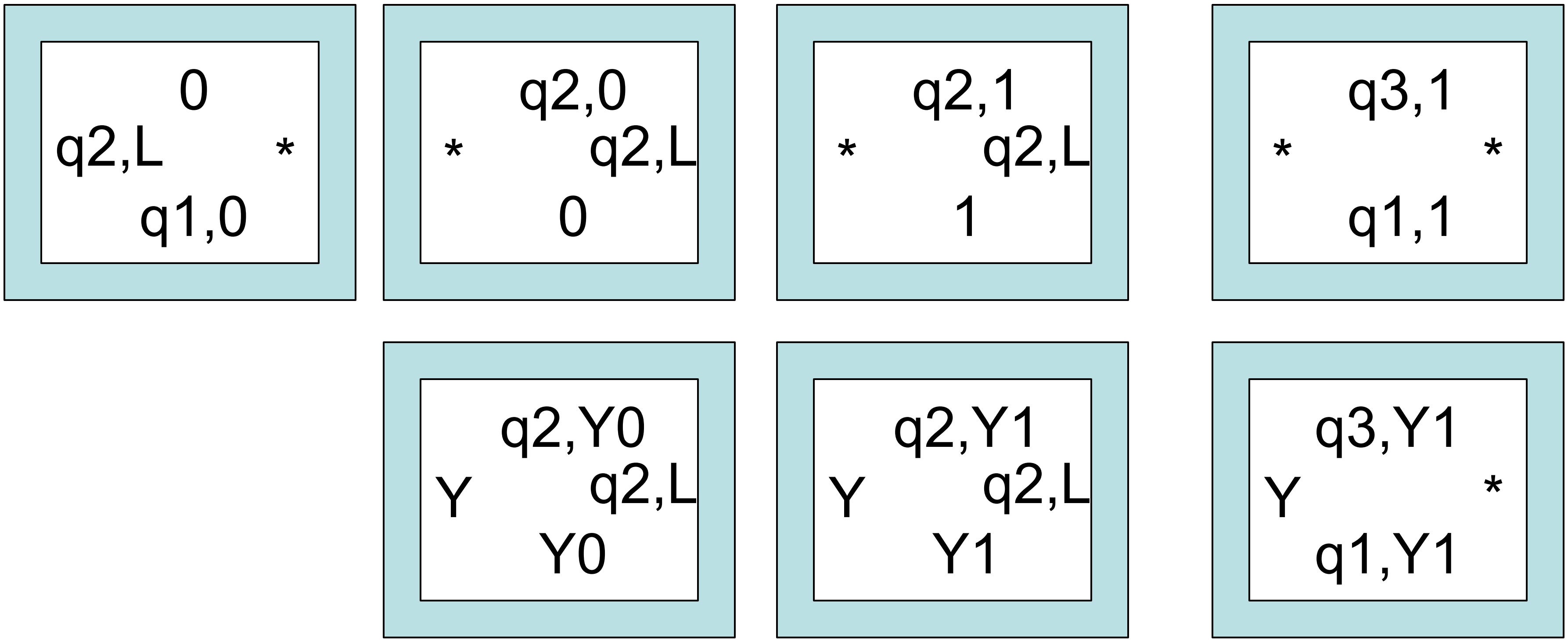
Tile Replication



q0 0 0 q2 q0 1 R q1

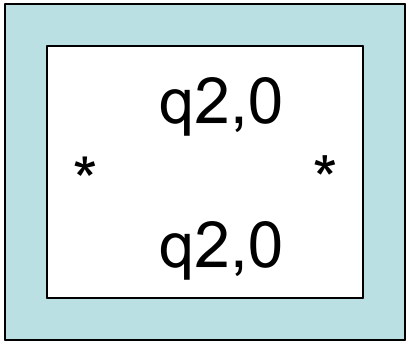


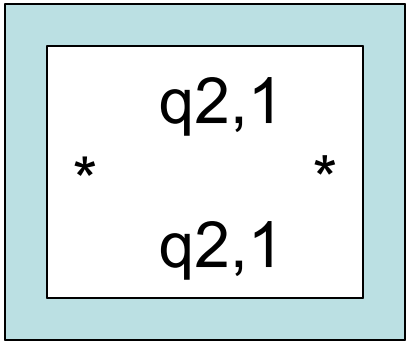
q1 0 L q2 q1 1 1 q3

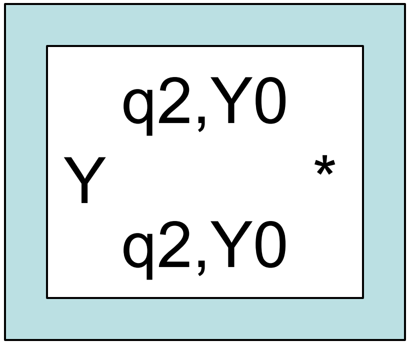


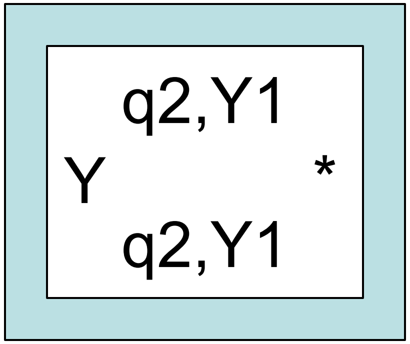
q2 0 0 q2

q2 1 1 q2

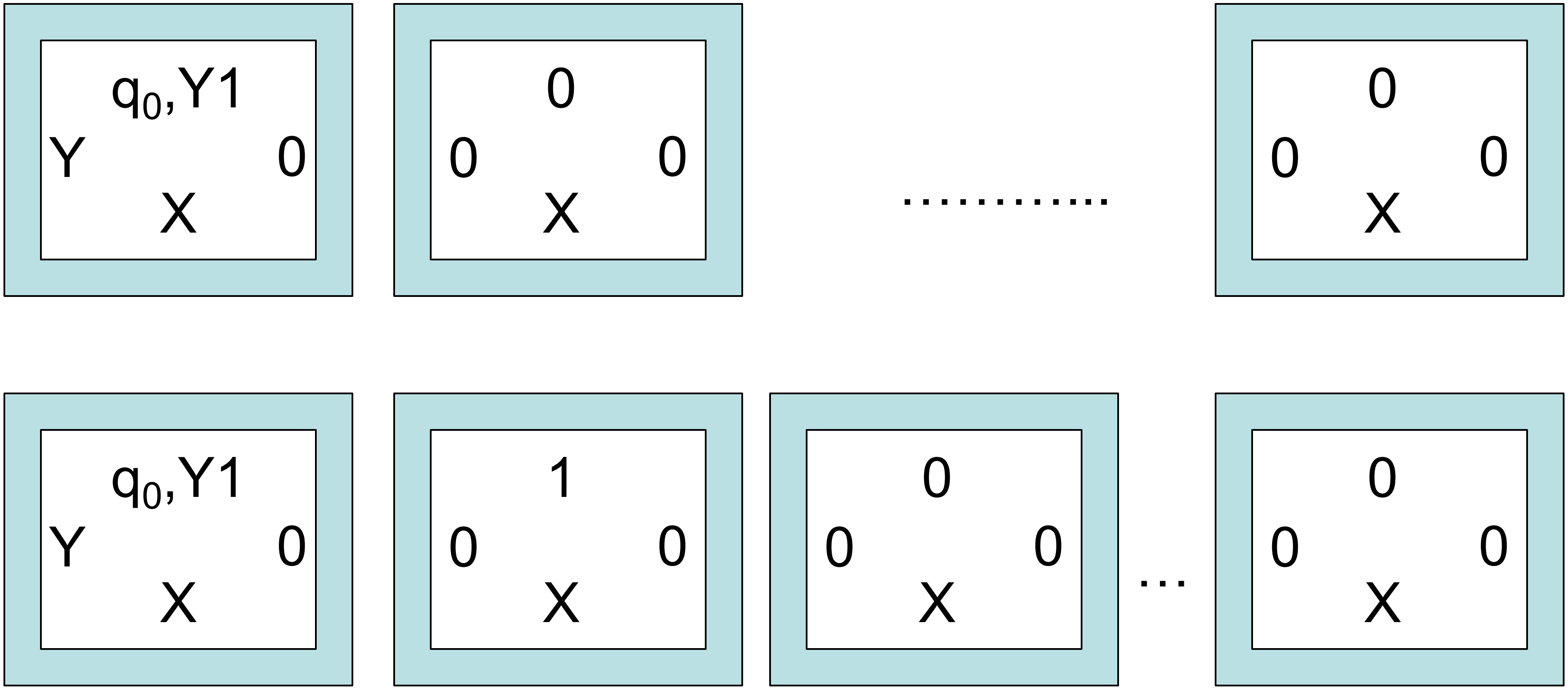




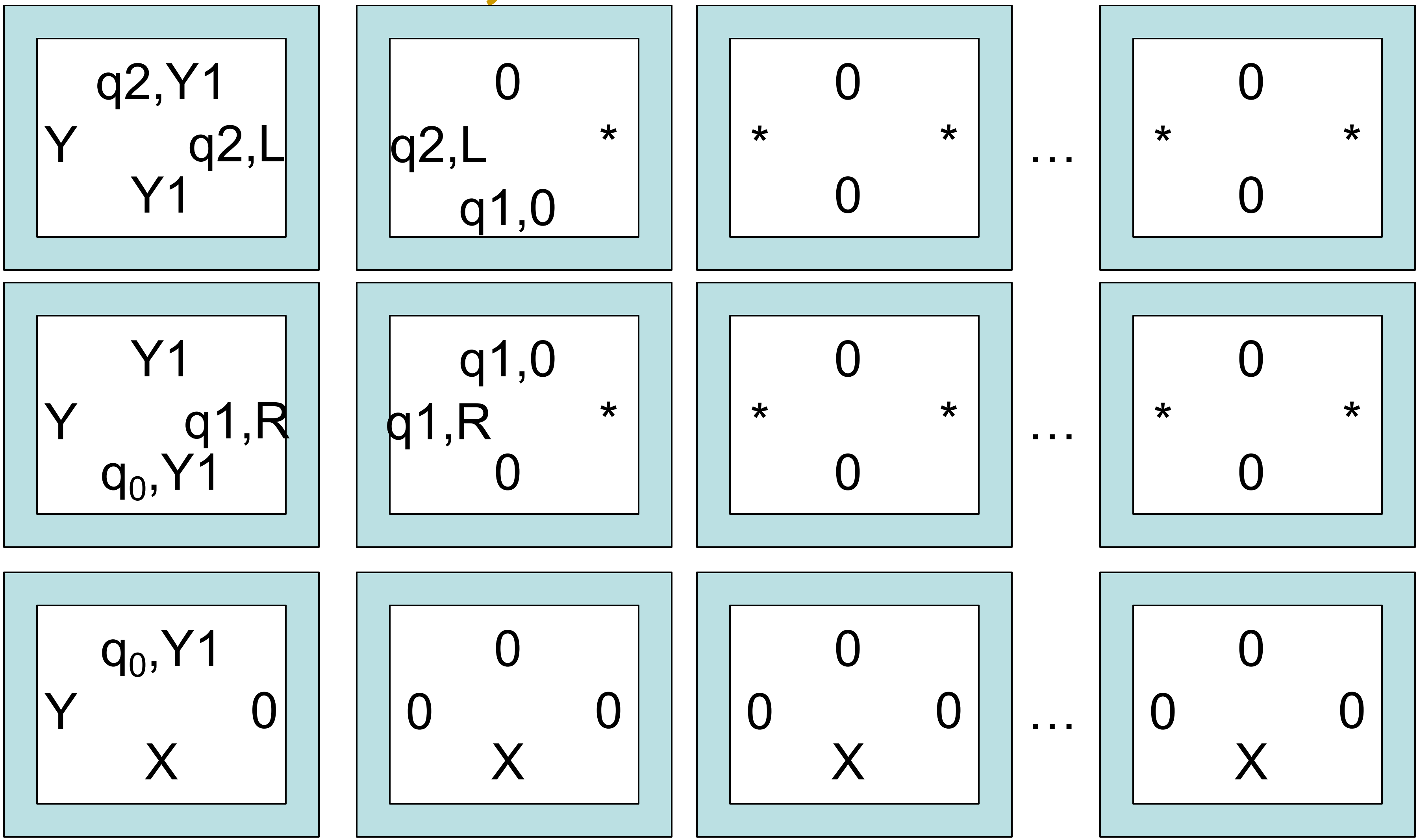




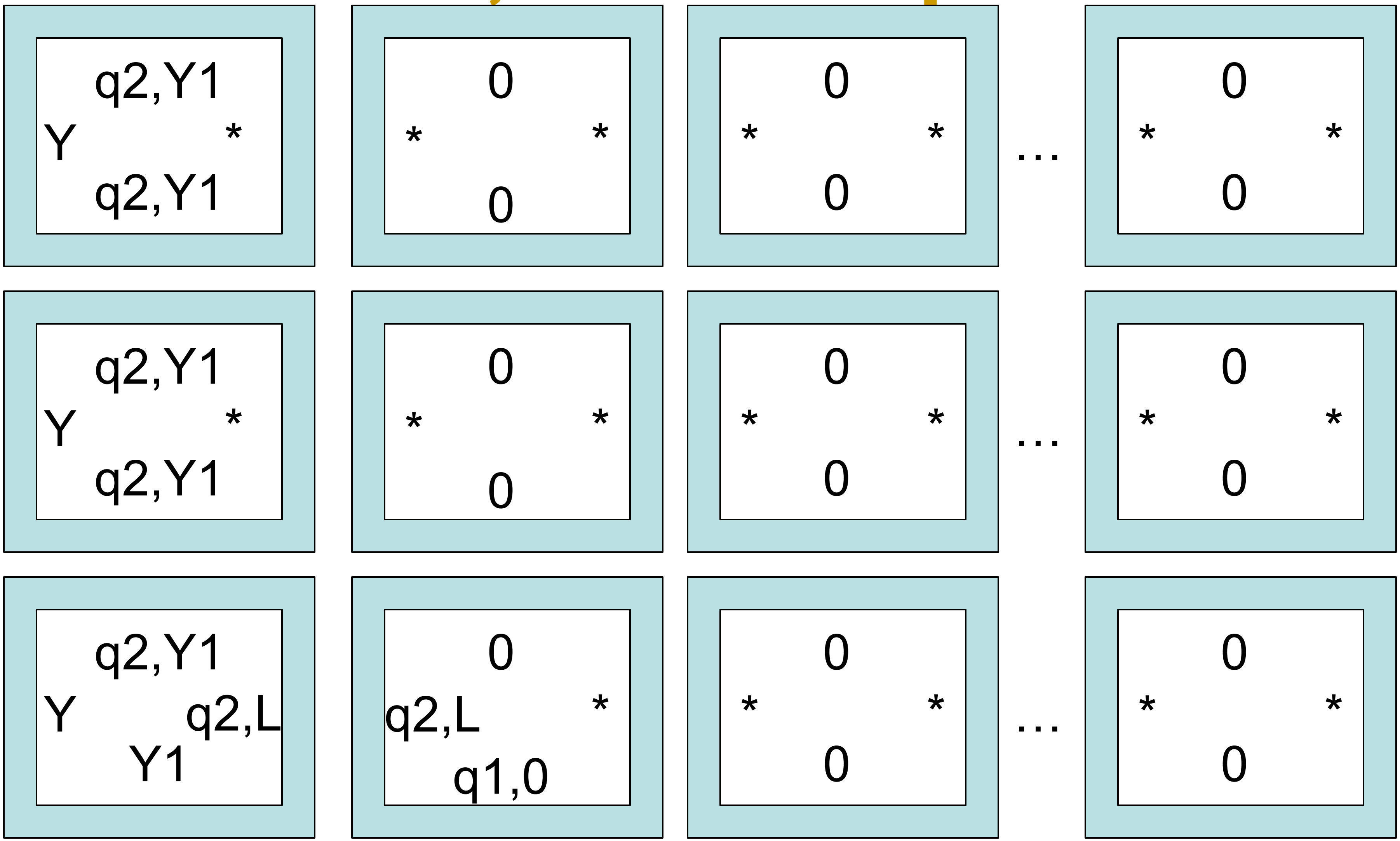
Sample Starting Rows



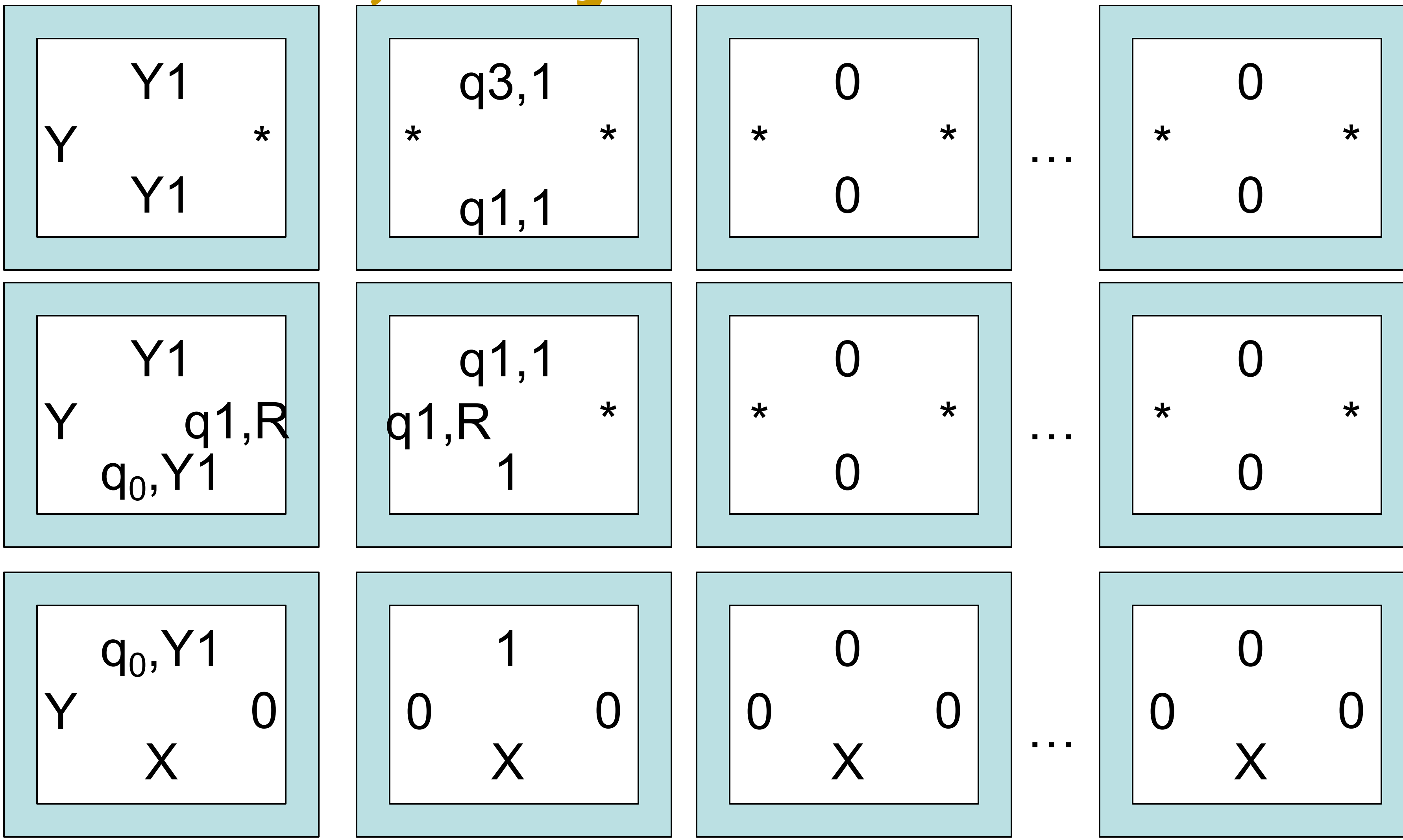
Case 1; Two More Rows



Case 1; Row 3 repeated



Case 2; Only Two More Rows



Comments on Variations

- One dimensional space (think about it)
- Infinite 3d space (really impossible in general)
  - This become a for all, there exists problem
  - In fact, one can mimic acceptance on all inputs here, meaning M is an algorithm iff we can tile the 3d space

PCP Revisited

Bounded Post Correspondence

Bounded Variation

- Limit correspondence to a length that is polynomial in n, where n is length of initial input string.
- Outline of proof we can get for almost free
  - Convert halting problem for a Non-deterministic Turing machine to word problem for a Semi-Thue System

    Note: we originally did for deterministic machines but the construction works for non-determinism and maps nicely to Semi-Thue systems which are non-deterministic by definition.
  - Recast as an instance of PCP
  - Limit the length of word to $(n+2)^k$, where original TM accepts or rejects in n^k steps.

Another Approach

- There is a tighter bound on Bounded PCP.

- Given sequences (x1, x2, …, xn) and (y1, y2, …, yn), and a positive integer
  $K \leq p(\max(|x1|+\ldots+|xn|, |y1|+\ldots+|yn|)$,
  where p is some polynomial, is there a solution to this instance involving indices i1, …,ik, k≤K (not necessarily distinct), of integers between 1 and n, such that the corresponding x and y strings are identical.
- Follows from Constable, Hunt and Sahni (1974). “On the Computational Complexity of Program Scheme Equivalence,” *Siam Journal of Computing* 9(2), 396-416.

Co-NP

Fourth Significant Class of Problems

Co–NP

For any decision problem A in NP, there is a ‘complement’ problem Co–A defined on the same instances as A, but with a question whose answer is the negation of the answer in A. That is, an instance is a "yes" instance for A if and only if it is a "no" instance in Co–A.

Notice that the complement of the complement of a problem is the original problem.

Co–NP

Co–NP is the set of all decision problems whose complements are members of NP.

For example: consider Graph Color GC

Given: A graph G and an integer k.

Question: Can G be properly colored with k colors?

Co–NP

The complement problem of GC

Co–GC

Given: A graph G and an integer k.

Question: Do all proper colorings of G require <u>more than</u> k colors?

Co–NP

Notice that Co–GC is a problem that does not appear to be in the set NP. That is, we know of no way to check in polynomial time the answer to a "Yes" instance of Co–GC.

What is the "answer" to a Yes instance that can be verified in polynomial time?

Co–NP

Not all problems in NP behave this way. For example, if X is a problem in class P, then both "yes" and "no" instances can be solved in polynomial time.

That is, both "yes" and "no" instances can be verified in polynomial time and hence, X and Co–X are both in NP, in fact, both are in P.

This implies P = Co–P and, further,

$$P = \text{Co–}P \subseteq NP \cap \text{Co–}NP.$$

Co–NP

This gives rise to a second fundamental question:

NP = Co–NP?

If P = NP, then NP = Co–NP.

This is not "if and only if."

It is possible that NP = Co–NP and, yet, $P \neq NP$.

Co–NP

If $A \leq_P B$ and both are in NP, then the same polynomial transformation will reduce Co-A to Co–B. That is,
Co–A $\leq_P$ Co–B. Therefore, Co–SAT is 'complete' in Co–NP.

In fact, corresponding to NP–Complete is the complement set Co–NP–Complete, the set of hardest problems in
Co–NP.

Turing Reductions

Now, return to Turing Reductions.

Recall that Turing reductions include polynomial transformations as a special case. So, we should expect they will be more powerful.

Turing Reductions

(1) Problems A and B can, but need not, be decision problems.

(2) No restriction placed upon the number of instances of B that are constructed.

(3) Nor, how the result, Answer_A, is computed.

In effect, we use an Oracle for B.

Turing Reductions

Technically, Turing Reductions include Polynomial Transformations, but it is useful to distinguish them.

Polynomial transformations are often the easiest to apply.

NP–Hard

Fifth Significant Class of Problems

NP–Hard

To date, we have concerned ourselves with decision problems. We are now in a position to include additional problems. In particular, optimization problems.

We require one additional tool – the second type of transformation discussed above – Turing reductions.

NP–Hard

Definition: Problem B is NP–Hard if there is a polynomial time Turing reduction A $\leq_{PT}$ B for some problem A in NP–Complete.

This implies NP–Hard problems are at least as hard as NP–Complete problems. Therefore, they cannot be solved in polynomial time unless P = NP (and maybe not then).

This use of an oracle, allows us to reduce co-NP-Complete problems to NP-Complete ones and vice versa.

QSAT

- **QSAT is the problem to determine if an arbitrary fully quantified Boolean expression is true. Note: SAT only uses existential.**
- **QSAT is NP-Hard but may not be in NP.**
- **QSAT can be solved in polynomial space (PSPACE).**

NP–Hard

Polynomial transformations are Turing reductions.

Thus, NP–Complete is a subset of NP–Hard.

Co–NP–Complete also is a subset of NP–Hard.

NP–Hard contains many other interesting problems.

NP-Easy

- **NP-Easy** is the set of function problems that are solvable in polynomial time by a deterministic Turing machine with an oracle for some decision problem in NP.
- That is, given an Oracle for some NP problem **Y**, if **X** is Turing reducible to **Y** in polynomial time then **X** is **NP-Easy**.

NP–Easy

Problem X need not be, but often is, NP–Complete.

In fact, X can be any problem in NP or Co–NP.

More to the point, an NP-Easy problem does not even need to be a decision problem – it can be an optimization problem or some other problem seeking a numerical rather than binary (yes/no answer).

NP–Equivalent

Problem B in NP–Hard is *NP–Equivalent* when B reduces to some problem X in NP, That is, B $\leq_{PT}$ X. This is, when B is also NP-Easy.

Since B is in NP–Hard, we already know there is a problem A in NP–Complete that reduces to B. That is, A $\leq_{PT}$ B.

Since X is in NP, X $\leq_{PT}$ A. Therefore, X $\leq_{PT}$ A $\leq_{PT}$ B $\leq_{PT}$ X.

Thus, X, A, and B are all polynomially equivalent, and we can say

Theorem. Problems in NP–Equivalent are polynomial if and only if P = NP.

Example: Optimization version of Subset-Sum is NP-Equivalent.

NP-Easy and Equivalent

- **NP-Easy -- these are problems that are polynomial when using an NP oracle ($\le_{pt}$)**
- **NP-Equivalent is the class of NP-Easy and NP-Hard problems (assuming Turing rather than many-one reductions)**
  - **In essence this is the functional equivalent of NP-Complete but also of Co-NP-Complete since can negate answers**

SubsetSum Optimization

$S = \{s_1, s_2, \ldots, s_n\}$

set of positive integers

and an integer B.

Optimization: Find a subset of S whose values sum to the largest attainable value ≤B?

Strategy: Use Oracle for SubsetSum Decision Problem but only use it a polynomial number of times

Using SubsetSum Oracle

```
SUBSET-SUM-OPTIMIZATION(A, b)
    int best = b;
    for i = floor(log2b) downto 0 do
        A = A + { 2^i }          // add to set
    for i = floor(log2b) downto 0 do
        A = A - { 2^i}           // remove from set
        if not SUBSET-SUM(A, best) then
            best = best - 2^i    // reduce best
    return best
```

Example of SubsetSum Opt

- Initial Values:
- A = {1, 4, 5, 7}, best = b = 15
- A = {1, 4, 5, 7, 8, 4, 2, 1}, best = 15
- A = {1, 4, 5, 7, 4, 2, 1}, best = 15
- A = {1, 4, 5, 7, 2, 1}, best = 15
- A = {1, 4, 5, 7, 1}, best = 15-2 = 13
- A = {1, 4, 5, 7}, best = 13

Another Example

- Initial Values:
- A = {1, 4, 5, 7}, best = b = 20
- A = {1, 4, 5, 7, 16, 8, 4, 2, 1}, best = 20
- A = {1, 4, 5, 7, 8, 4, 2, 1}, best = 20
- A = {1, 4, 5, 7, 4, 2, 1}, best = 20
- A = {1, 4, 5, 7, 2, 1}, best = 20-2 = 18
- A = {1, 4, 5, 7, 1}, best = 20-2 = 18
- A = {1, 4, 5, 7}, best = 18-1 = 17

Analysis

- Each loop has $O(\log_2 b)$ iterations, which is linear with respect to the size of b.
- Note that if we tried all values less that b, we would have O(b) tries and that is exponential in $\log_2 b$, the size of b.
- The correct solution takes advantage of the NP-complete power of the oracle.

Minimum Colors for a Graph

- We know K-Color is NP Complete
- We can reduce KC to Min Color problem just by seeing if Min is ≤ K.
- How do we reduce Min Color to KC asking only a log number of questions of the oracle for KC?
- Consider, if N nodes, then can easily N-Color
- Can we N/2-Color?
  - If not, then try N/4
  - If so, then try 3N/4
- This is a simple binary search for optimal value

PSPACE

- PSPACE is set of problems solvable in polynomial space with unlimited time

  PSPACE = U SPACE(n^k)
- PSPACE = co-PSPACE = NPSPACE
- PSPACE is a strict superset of CSLs
- A PSPACE-Complete Problem is, given a regular expression e over Σ, does e denote all strings in Σ*?
- Another PSPACE-Complete problem is QSAT

EXPTIME and EXPSPACE

- EXPTIME is the set of problems solvable in $2^{p(n)}$ where is p is some polynomial.
- NEXPTIME is the set of problems solvable in $2^{p(n)}$ on a non-deterministic TM.
- EXPSPACE is set of problems solvable in $2^{p(n)}$ space and unlimited time

Complexity Hierarchy

- $P \subseteq NP \subseteq PSPACE = NPSPACE \subseteq EXPTIME \subseteq NEXPTIME \subseteq EXPSPACE \not\subseteq$ 2-EXPTIME $\not\subseteq$ 3-EXPTIME $\not\subseteq \ldots \not\subseteq$ ELEMENTARY $\not\subseteq$ PRF $\not\subseteq$ REC
- P $\neq$ EXPTIME; At least one of these is true
  - P $\not\subseteq$ NP
  - NP $\not\subseteq$ PSPACE
  - PSPACE $\not\subseteq$ EXPTIME
- NP $\neq$ NEXPTIME
  - Note that EXPTIME = NEXPTIME iff P=NP
  - Note that k-EXPTIME $\not\subseteq$ (k+1)-EXPTIME, k>0
- PSPACE $\neq$ EXPSPACE; At least one of these is true
  - PSPACE $\not\subseteq$ EXPTIME
  - EXPTIME $\not\subseteq$ EXPSPACE

Alternating TM (ATM)

- ATM adds to NDTM notation the notion where, for each state q, q has one of the following properties: (accept, reject, $\vee$, $\wedge$)
  - $\vee$ means mean accept the string if <u>any</u> final state reached after q is accepting
  - $\wedge$ means mean accept the string if <u>all</u> final states reached after q are accepting
- AP = PSPACE where AP is class of problems solvable in polynomial time on an ATM

QSAT, Petri Net, Presburger

- QSAT is solvable by an alternating TM in polynomial time and polynomial space
- As noted before, QSAT is PSPACE-Complete
- Petri net reachability is EXPSPACE-hard and requires 2-EXPTIME
- Presburger arithmetic is at least in 2-EXPTIME, at most in 3-EXPTIME, and can be solved by an ATM with n alternating quantifiers in doubly exponential time

FP and FNP

- **FP is functional equivalent to P R(x,y) in FP if can provide value y for input x via a deterministic polynomial time algorithm**
- **FNP is functional equivalent to NP; R(x,y) in FNP if can verify any pair (x,y) via a deterministic polynomial time algorithm**

TFNP

- TFNP is the subset of FNP where a solution always exists, i.e., there is a y for each x such that R(x,y).
  - Task of a TFNP algorithm is to find a y, given x, such that R(x,y)
  - Unlike FNP, the search for a y is always successful
- FNP properly contains TFNP contains FP (we don't know if proper)

Prime Factoring

- **Prime factoring is defined as, given n and k, does n have a prime factor <k?**
- **Factoring is in NP and co-NP**
  - **Given candidate factor can check its primality in poly time and then see if it divides n**
  - **Given candidate set of factors can check their primalities, and see if product equals n; if so, and no candidate < k, then answer is no**

Prime Factoring and TFNP

- Prime Factoring as a functional problem is in TFNP, but is it in FP?
- If TFNP in FP then TFNP = FP since FP contained in TFNP
- If that is so, then carrying out Prime Factoring is in FP and its decision problem is in P
  - If this is so, we must fear for encryption, most of which depends on difficulty of finding factors of a large number

More TFNP

- **There is no known recursive enumeration of TFNP but there is of FNP**
  - **This is similar to total versus partially recursive functions (analogies are everywhere)**
- **It appears that TFNP does not have any complete problems!!!**
  - **But there are subclasses of TFNP that do have complete problems!!**

Another Possible Analogy

- Is P = (NP intersect Co-NP)?
- Recall that REC = (RE intersect co-RE)
- The analogous result may not hold here

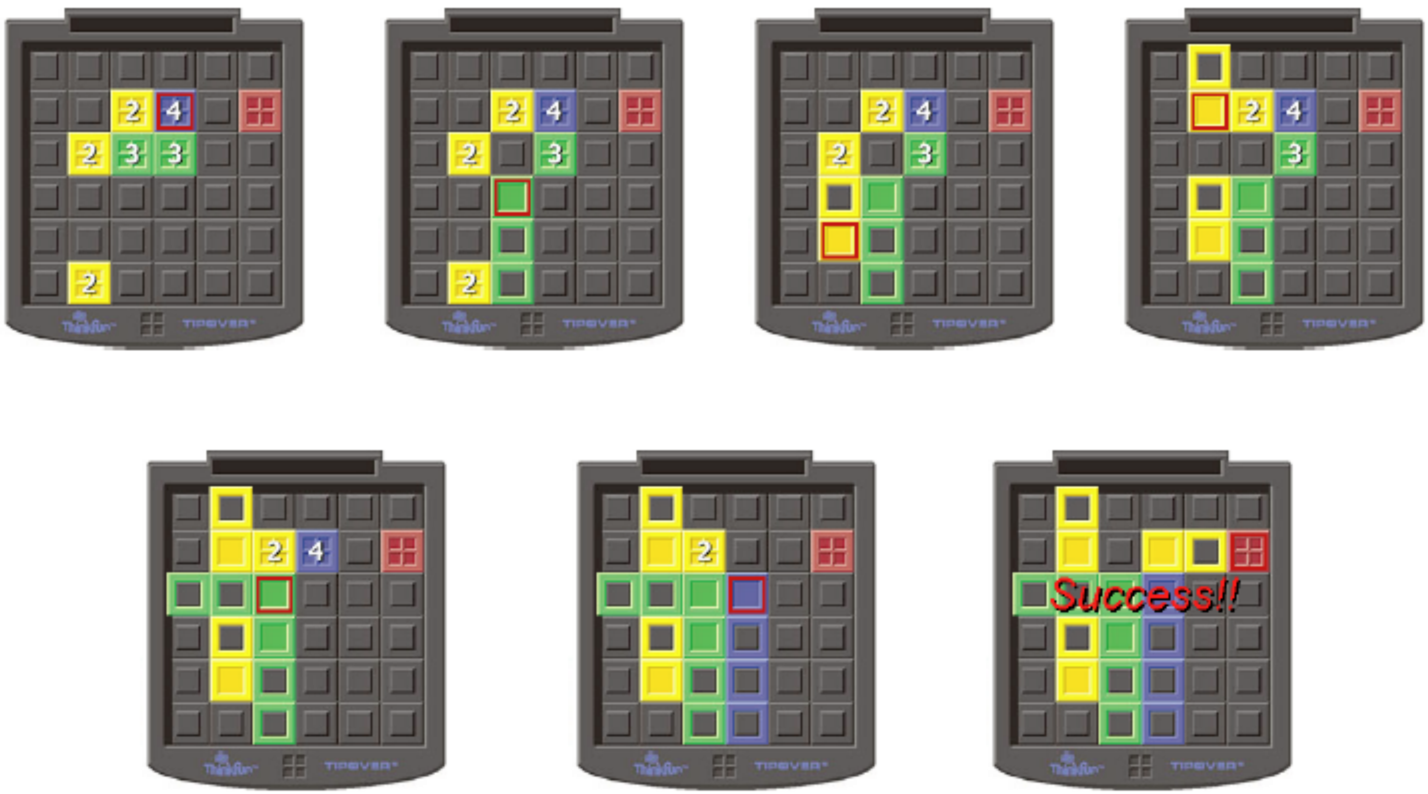
Turing vs m-1 Reductions

- In effect, our normal polynomial reduction ($\leq$p) is a many-one polynomial time reduction as it just asks and then accepts its oracle's answer
- In contrast, NP-Easy and NP-Equivalent employ a Turing machine polynomial time reduction ($\leq$pt) that uses rather than mimics answers from its oracle

More Examples of NP Complete Problems

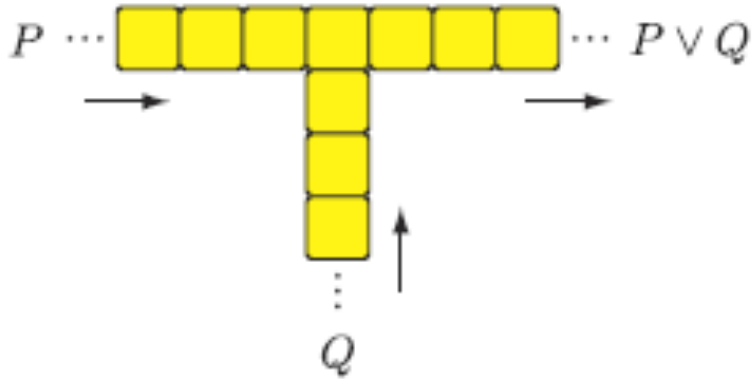
TipOver

Rules of Game



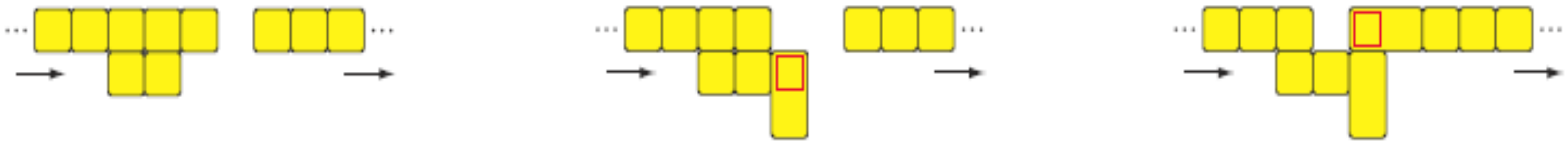
Numbers are height of crate stack;
If could get 4 high out of way we can attain goal

Problematic OR Gadget



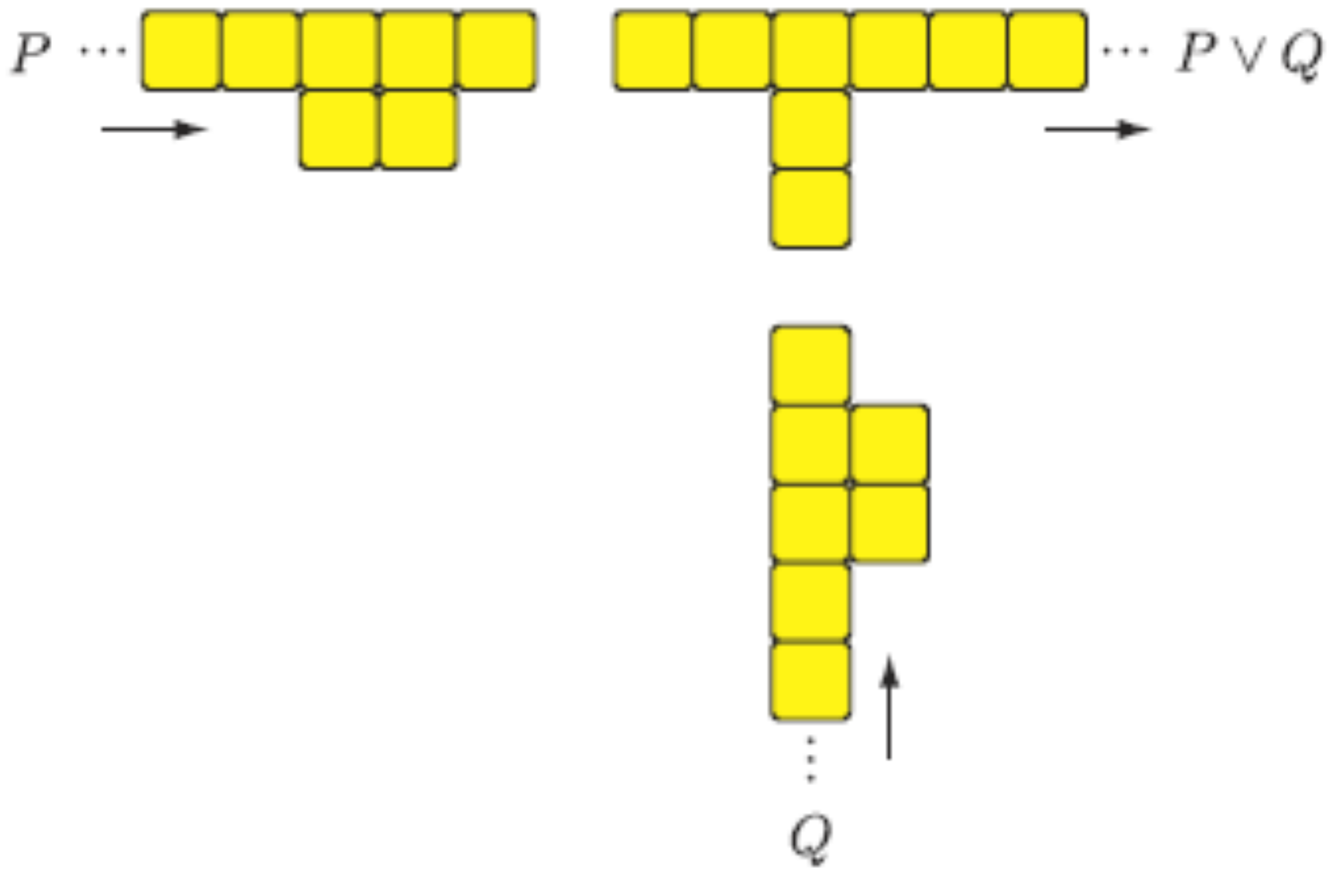
Can go out where did not enter

Directional gadget

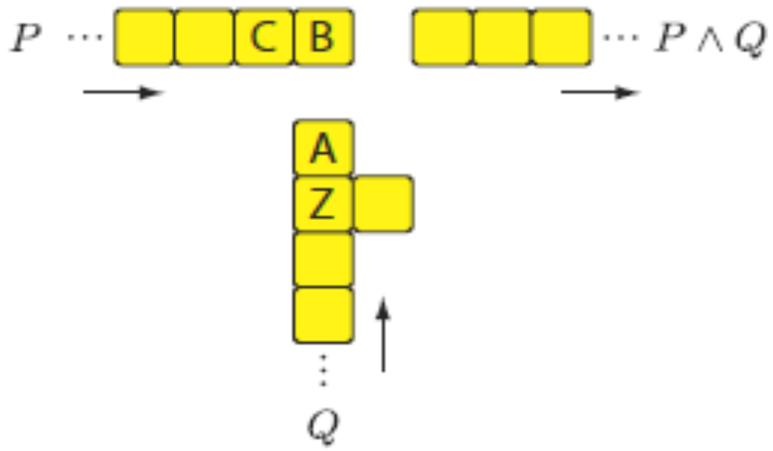


Single stack is two high;
tipped over stack is one high, two long;
red square is location of person travelling the towers

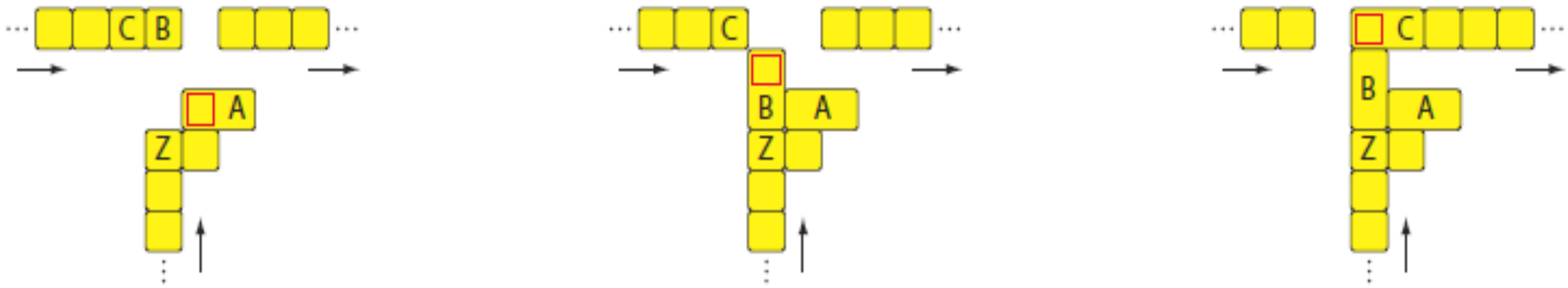
One directional Or gadget



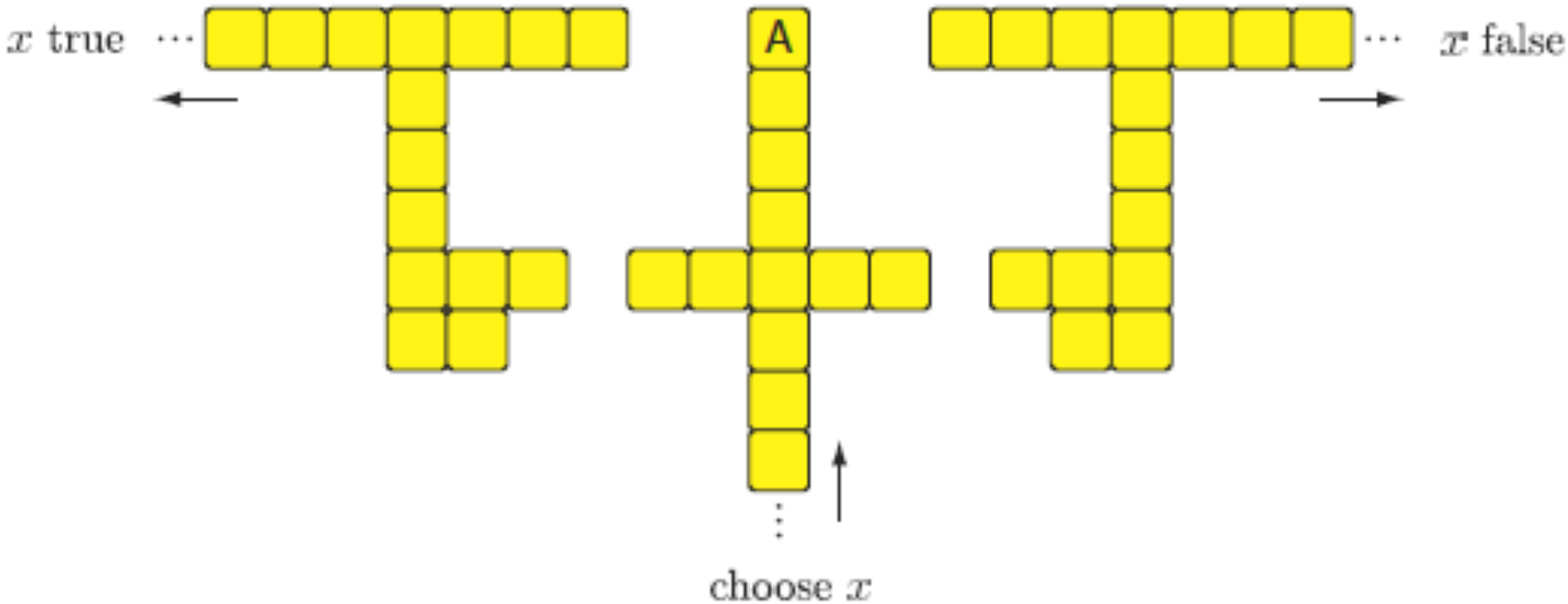
AND Gadget



How AND Works

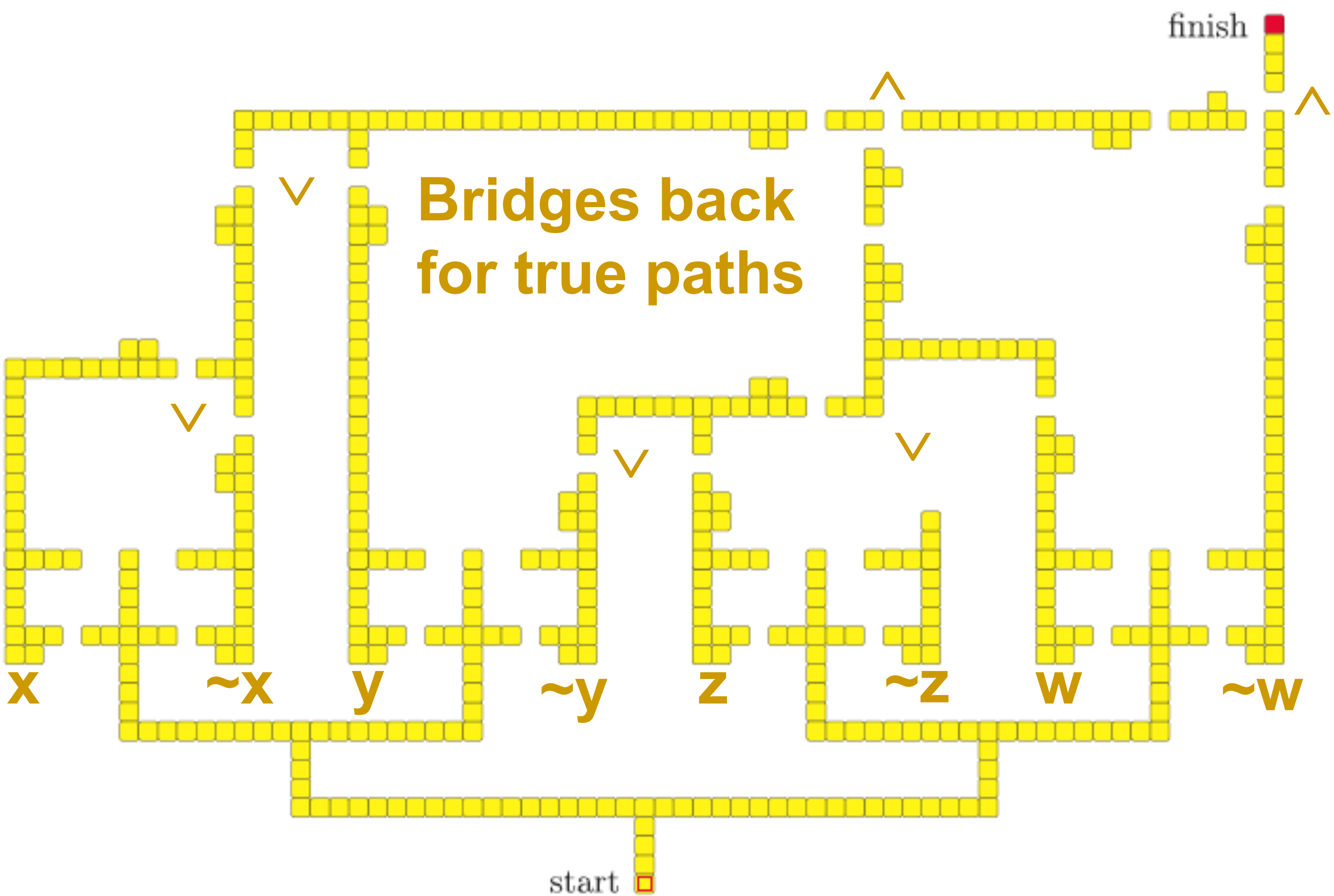


Variable Select Gadget



Tip A left to set x true; right to set x false
Can build bridge to go back but never to change choice

$((x \vee \sim x \vee y) \wedge (\sim y \vee z \vee w) \wedge \sim w)$



Win Strategy is NP-Complete

- TipOver win strategy is NP-Complete
- Minesweeper consistency is NP-Complete
- Phutball single move win is NP-Complete
  - Do not know complexity of winning strategy
- Checkers is really interesting
  - Single move to King is in P
  - Winning strategy is PSpace-Complete

Finding Triangle Strips

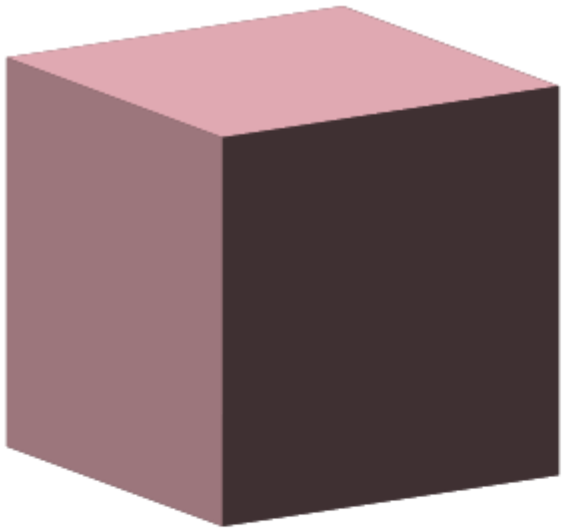
Adapted from presentation by
Ajit Hakke Patil
Spring 2010

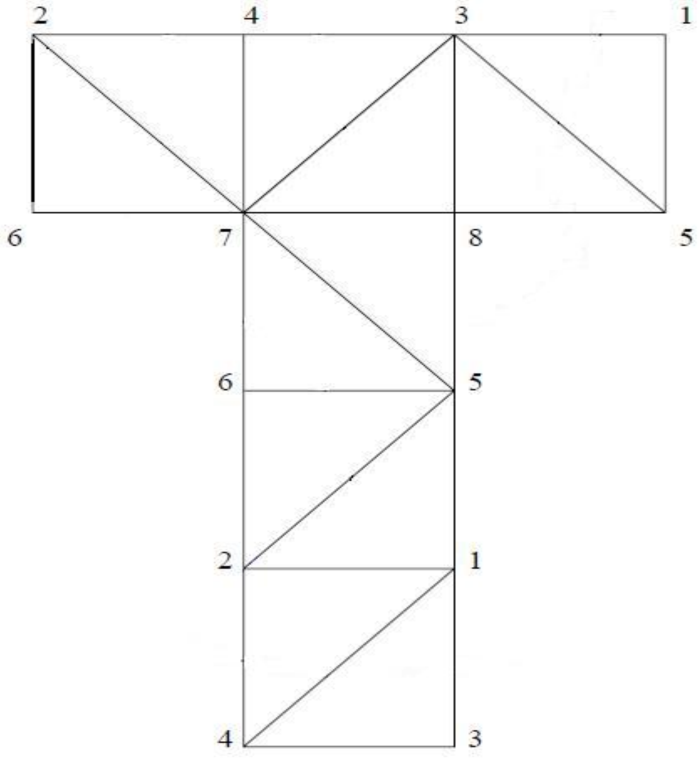
Graphics Subsystem

- **The graphics subsystem (GS) receives graphics commands from the application running on CPU/GPU over a bus, builds the image specified by the commands, and outputs the resulting image to display hardware**
- **Graphics Libraries:**
  - **OpenGL, DirectX.**

Surface Visualization

- As Triangle Mesh
- Generated by triangulating the geometry



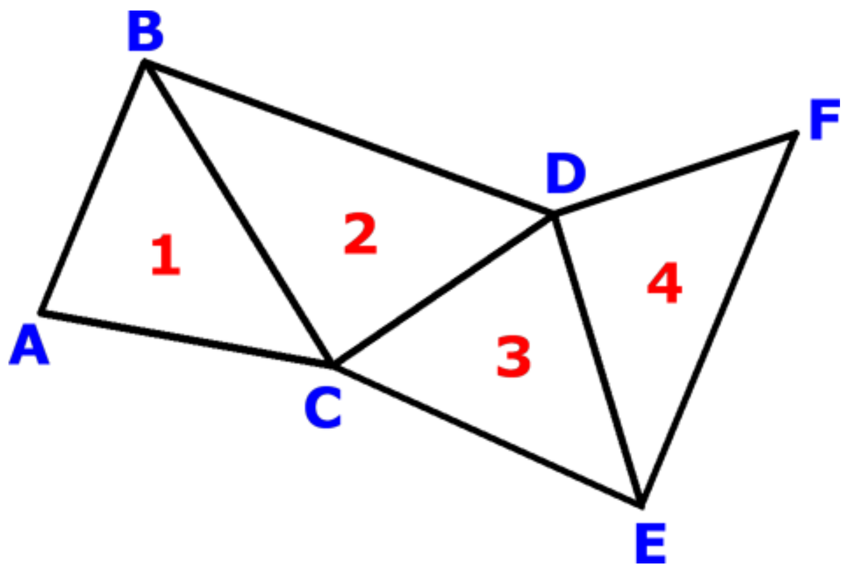


Triangle List vs Triangle Strip

- Triangle List: *Arbitrary ordering of triangles.*
- Triangle Strip: *A triangle strip is a sequential ordering of triangles*. i.e consecutive triangles share an edge
- In case of triangle lists we draw each triangle separately.
- So for drawing N triangles you need to call/send 3N vertex drawing commands/data.
- However, using a Triangle Strip reduces this requirement from 3N to N + 2, provided a single strip is sufficient.

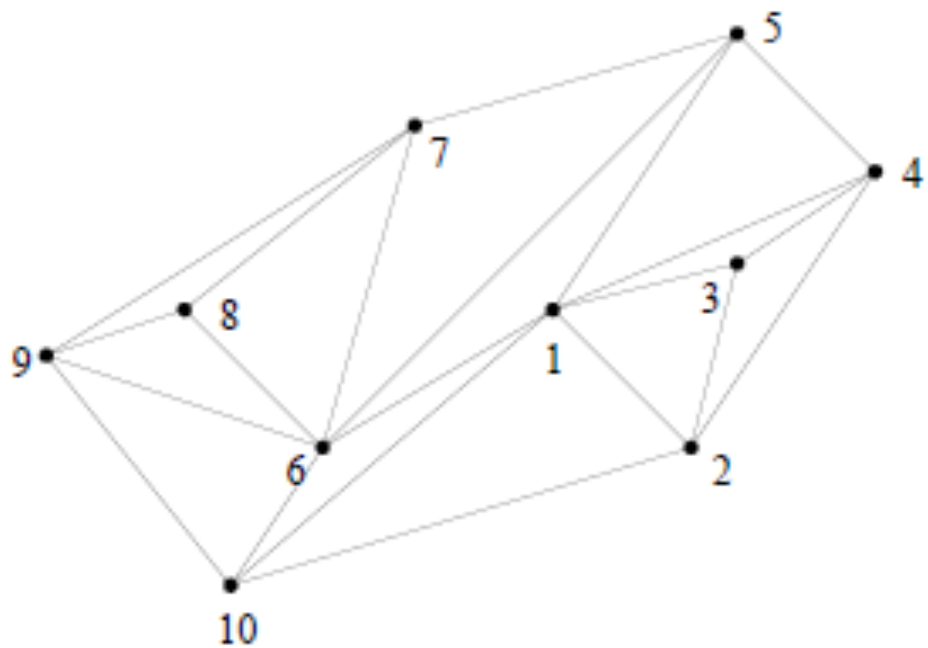
Triangle List vs Triangle Strip

- four separate triangles: ABC, CBD, CDE, and EDF
- But if we know that it is a triangle strip or if we rearrange the triangles such that it becomes a triangle strip, then we can store it as a sequence of vertices ABCDEF
- This sequence would be decoded as a set of triangles ABC, BCD, CDE and DEF
- Storage requirement:
  - 3N => N + 2



Tri-strips example

- Single tri-strip that describes triangles is: 1,2,3,4,1,5,6,7,8,9,6,10,1,2



K-Stripability

- Given some positive integer K (less than the number of triangles).
- Can we create K tri-strips for some given triangulation – no repeated triangles.

Triangle List vs Triangle Strip

```
// Draw Triangle Strip
glBegin(GL_TRIANGLE_STRIP);
 For each Vertex
{
   glVertex3f(x,y,z); //vertex
}
glEnd();
```

```
// Draw Triangle List
glBegin(GL_TRIANGLES);
For each Triangle
{
      glVertex3f(x1,y1,z1);// vertex 1
      glVertex3f(x2,y2,z2);// vertex 2
      glVertex3f(x3,y3,z3);// vertex 3
}
glEnd();
```

Problem Definition

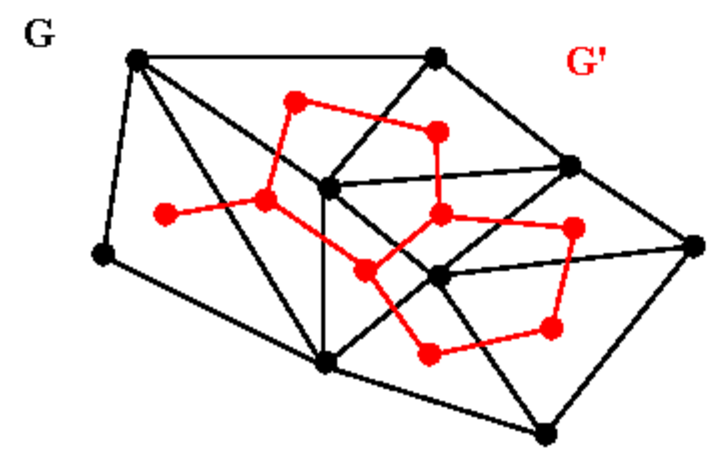
- Given a triangulation $T = \{t_1, t_2, t_3 ,\ldots t_n\}$. Find the triangle strip (sequential ordering) for it?
- Converting this to a decision problem.
- Formal Definition:
  Given a triangulation $T = \{t_1, t_2, t_3 ,\ldots t_N\}$. Does there exists a triangle strip?

NP Proof

- Provided a witness of a ‘Yes’ instance of the problem. we can verify it in polynomial time by checking if the sequential triangles are connected.
- Cost of checking if the consecutive triangles are connected
  – For i to N -1
    - Check of i_{th} and $i+1_{th}$ triangle are adjacent (have a common edge)
    - Three edge comparisions or six vertex comparisions
  – ~ 6N
- Hence it is in NP.

Dual Graph

- The *dual graph* of a triangulation is obtained by defining a vertex for each triangle and drawing an edge between two vertices if their corresponding triangles share an edge
- This gives the triangulations *edge-adjacency* in terms of a graph
- Cost of building a Dual Graph
  - $O(N^2)$
- e.g G' is a dual graph of G.



NP-Completeness

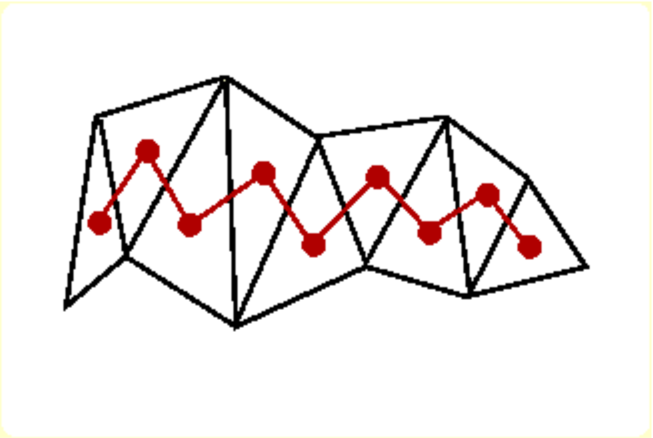
- To prove it's NP-Complete we reduce a known NP-Complete problem to this one; the Hamiltonian Path Problem.
- Hamiltonian Path Problem:
  – Given: A Graph G = (V, E). Does G contains a path that visits every vertex exactly once?

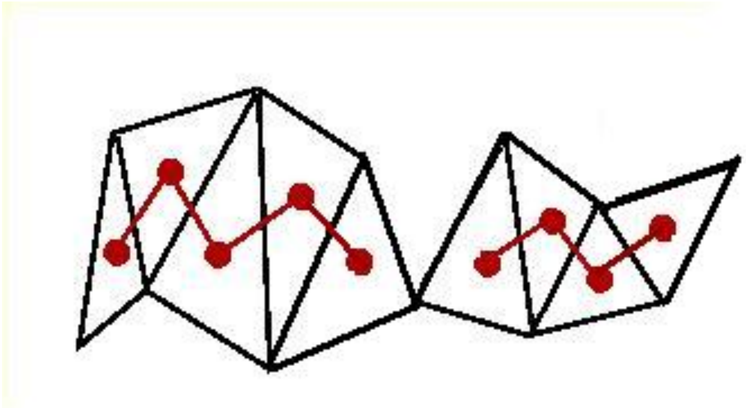
NP-Completeness proof by restriction

- Accept an Instance of Hamiltonian Path, G = (V, E), we restrict this graph to have max. degree = 3.The problem is still NP-Complete.
- Construct an Instance of HasTriangleStrip
  - G' = G
    - V' = V
    - E' = E
  - Let this be the dual graph G' = (V', E') of the triangulation T = {t1, t2, t3 ,.. tN}.
    - V' ~ Vertex v_i represents triangle $t_{i,}$ i = 1 to N
    - E' ~ An edge represents that two triangles are *edge-adjacent* (share an edge)
- Return HasTriangleStrip(T)

NP-Completeness

- G will have a Hamiltonian Path iff G’ has one (they are the same).
- G’ has a Hamiltonian Path iff T has a triangle strip of length N – 1.
- T will have a triangle strip of length N – 1 iff G (G’) has a Hamiltonian Path.
- ‘Yes’ instance maps to ‘Yes’ instance. ‘No’ maps to ‘No.’





HP $<_P$ HasTriangleStrip

- The ‘Yes/No’ instance maps to ‘Yes/No’ instance respectively and the transformation runs in polynomial time.
- Polynomial Transformation
- Hence finding Triangle Strip in a given triangulation is a NP-Complete Problem

Undecidability of Finite Convergence for Operators on Formal Languages

Relation to Real-Time (Constant Time) Execution

Simple Operators

- Concatenation
  - $A \bullet B = \{ xy \mid x \in A \ \& \ y \in B \}$
- Insertion
  - $A \triangleright B = \{ xyz \mid y \in A, xz \in B, x, y, z \in \Sigma^* \}$
  - Clearly, since x can be λ, $A \bullet B \subseteq A \triangleright B$

K-insertion

- $A \triangleright^{[k]} B = \{ x_1y_1x_2y_2 \ldots x_ky_kx_{k+1} \mid$
  $y_1y_2 \ldots y_k \in A,$
  $x_1x_2 \ldots x_kx_{k+1} \in B,$
  $x_i, y_j \in \Sigma^*\}$

- Clearly, $A \bullet B \subseteq A \triangleright^{[k]} B$, for all $k>0$

Iterated Insertion

- $A\ (1) \triangleright^{[n]} B = A \triangleright^{[n]} B$
- $A\ (k+1) \triangleright^{[n]} B = A \triangleright^{[n]} (A\ (k) \triangleright^{[n]} B)$

Shuffle

- Shuffle (product and bounded product)
  - $A \diamond B = \cup_{j \geq 1} A \triangleright^{[j]} B$
  - $A \diamond^{[k]} B = \cup_{1 \leq j \leq k} A \triangleright^{[j]} B = A \triangleright^{[k]} B$

- One is tempted to define shuffle product as
  $A \diamond B = A \triangleright^{[k]} B$ where
  $k = \mu\, y\, [\, A \triangleright^{[j]} B = A \triangleright^{[j+1]} B\,]$
  but such a k may not exist – in fact, we will show the undecidability of determining whether or not k exists

More Shuffles

- Iterated shuffle
  - $A \diamond^{0} B = A$
  - $A \diamond^{k+1} B = (A \diamond^{[k]} B) \diamond B$
- Shuffle closure
  - $A \diamond^{*} B = \cup_{k \geq 0} (A \diamond^{[k]} B)$

Crossover

- Unconstrained crossover is defined by
  $A \otimes_u B = \{ wz, yx \mid wx \in A \text{ and } yz \in B \}$

- Constrained crossover is defined by
  $A \otimes_c B = \{ wz, yx \mid wx \in A \text{ and } yz \in B, |w| = |y|, |x| = |z| \}$

Who Cares?

- People with no real life (me?)
- Insertion and a related deletion operation are used in biomolecular computing and dynamical systems
- Shuffle is used in analyzing concurrency as the arbitrary interleaving of parallel events
- Crossover is used in genetic algorithms

Some Known Results

- Regular languages, A and B
  - A $\bullet$ B is regular
  - A $\triangleright^{[k]}$ B is regular, for all k>0
  - A $\Diamond$ B is regular
  - A $\Diamond^*$ B is not necessarily regular
    - Deciding whether or not A $\Diamond^*$ B is regular is an open problem

More Known Stuff

- CFLs, A and B
  - A • B is a CFL
  - A $\triangleright$ B is a CFL
  - A $\triangleright^{[k]}$ B is not necessarily a CFL, for k>1
    - Consider $A=a^nb^n$; $B = c^md^m$ and k=2
    - Trick is to consider (A $\triangleright^{[2]}$ B) $\cap$ a*c*b*d*
  - A $\diamond$ B is not necessarily a CFL
  - A $\diamond^*$ B is not necessarily a CFL
    - Deciding whether or not A $\diamond^*$ B is a CFL is an open problem

Immediate Convergence

- $L = L^2$?
- $L = L \triangleright L$?
- $L = L \diamond L$?
- $L = L \diamond^* L$?
- $L = L \otimes_c L$?
- $L = L \otimes_u L$?

Finite Convergence

- $\exists k>0\ L^k = L^{k+1}$
- $\exists k\geq 0\ L\ (k) \triangleright L = L\ (k+1) \triangleright L$
- $\exists k\geq 0\ L \triangleright^{[\,k\,]} L = L \triangleright^{[\,k+1\,]} L$
- $\exists k\geq 0\ L \diamond^{k} L = L \diamond^{k+1} L$
- $\exists k\geq 0\ L\ (k) \otimes_c L = L\ (k+1) \otimes_c L$
- $\exists k\geq 0\ L\ (k) \otimes_u L = L\ (k+1) \otimes_u L$

- $\exists k\geq 0\ A\ (k) \triangleright B = A\ (k+1) \triangleright B$
- $\exists k\geq 0\ A \triangleright^{[\,k\,]} B = A \triangleright^{[\,k+1\,]} B$
- $\exists k\geq 0\ A \diamond^{k} B = A \diamond^{k+1} B$
- $\exists k\geq 0\ A\ (k) \otimes_c B = A\ (k+1) \otimes_c B$
- $\exists k\geq 0\ A\ (k) \otimes_u B = A\ (k+1) \otimes_u L$

Finite Power of CFG

- Let G be a context free grammar.
- Consider $L(G)^n$
- Question1: Is $L(G) = L(G)^2$?
- Question2: Is $L(G)^n = L(G)^{n+1}$, for some finite $n>0$?
- These questions are both undecidable.
- Think about why question1 is as hard as whether or not $L(G)$ is Σ^*.
- Question2 requires much more thought.

1981 Results

- Theorem 1:
  The problem to determine if $L = \Sigma^*$ is Turing reducible to the problem to decide if $L \bullet L \subseteq L$, so long as L is selected from a class of languages C over the alphabet Σ for which we can decide if $\Sigma \cup \{\lambda\} \subseteq L$.
- Corollary 1:
  The problem "is $L \bullet L = L$, for L context free or context sensitive?" is undecidable

Proof #1

- Question: Does $L \bullet L$ get us anything new?
  - i.e., Is $L \bullet L = L$?
- Membership in a CSL is decidable.
- Claim is that $L = \Sigma^*$ iff
  (1) $\Sigma \cup \{\lambda\} \subseteq L$; and
  (2) $L \bullet L = L$
- Clearly, if $L = \Sigma^*$ then (1) and (2) trivially hold.
- Conversely, we have $\Sigma^* \subseteq L^* = \cup_{n\geq 0} L^n \subseteq L$
  - first inclusion follows from (1); second from (2)

Subsuming •

- Let $\oplus$ be any operation that subsumes concatenation, that is $A \bullet B \subseteq A \oplus B$.
- Simple insertion is such an operation, since $A \bullet B \subseteq A \triangleright B$.
- Unconstrained crossover also subsumes •,
  $A \otimes_c B = \{ wz, yx \mid wx \in A \text{ and } yz \in B\}$

$L = L \oplus L$?

- Theorem 2:
  The problem to determine if $L = \Sigma^*$ is Turing reducible to the problem to decide if $L \oplus L \subseteq L$, so long as $L \bullet L \subseteq L \oplus L$ and L is selected from a class of languages C over Σ for which we can decide if $\Sigma \cup \{\lambda\} \subseteq L$.

Proof #2

- Question: Does $L \oplus L$ get us anything new?
  - i.e., Is $L \oplus L = L$?
- Membership in a CSL is decidable.
- Claim is that $L = \Sigma^*$ iff

  (1) $\Sigma \cup \{\lambda\} \subseteq L$; and

  (2) $L \oplus L = L$
- Clearly, if $L = \Sigma^*$ then (1) and (2) trivially hold.
- Conversely, we have $\Sigma^* \subseteq L^* = \cup_{n \geq 0} L^n \subseteq L$
  - first inclusion follows from (1); second from (1), (2) and the fact that $L \bullet L \subseteq L \oplus L$