University of
Central
Florida

Computability & Complexity Theory

Charles E. Hughes

COT 6410 – Spring 2014

Notes

Who, What, Where and When

- Instructor: **Charles Hughes;**
  Harris Engineering 247C; 823-2762
  (phone is not a good way to get me);
  charles.e.hughes@knights.ucf.edu
  (e-mail is a good way to get me)
  Subject: COT6410
- Web Page: **http://www.cs.ucf.edu/courses/cot6410/spr20124**
- Meetings: **TR 1:30PM-2:45PM, CB1-120;**
  28 periods, each 75 minutes long.
  Final Exam (Tuesday, April 29 from 1:00PM to 3:50PM) is separate from class meetings
- Office Hours: **TR 3:00PM-4:15PM**

Text Material

- This and other material linked from web site.
- References:
  - Arora&Barak, *Computational Complexity: A Modern Approach*, Cambridge University Press, 2009. http://www.cs.princeton.edu/theory/complexity/
  - Sipser, *Introduction to the Theory of Computation 3rd Ed*., Cengage Learning, 2013.
  - Garey&Johnson, *Computers and Intractability: A guide to the Theory of NP-Completeness*, W. H. Freeman & Co., 1979.
  - Papadimitriou & Lewis, *Elements of the Theory of Computation*, Prentice-Hall, 1997.
  - Hopcroft, Motwani&Ullman, *Intro to Automata Theory, Languages and Computation 3rd Ed.*, Prentice-Hall, 2006.
  - Davis, Sigal&Weyuker, *Computability, Complexity and Languages 2nd Ed*, Acad. Press (Morgan Kaufmann), 1994.
  - Oded Goldreich, *Computational Complexity: A Conceptual Approach*, Cambridge University Press, 2008. http://www.wisdom.weizmann.ac.il/~odedg/cc-drafts.html
  - Oded Goldreich, *P, NP, and NP-Completeness: The Basics of Complexity Theory*, Cambridge University Press, 2010. http://www.wisdom.weizmann.ac.il/~odedg/bc-drafts.html

Goals of Course

- Introduce Computability and Complexity Theory, including
  - Simple 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.

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
  - I will grade harder than my actual expectations run. In general, I will award everyone 110% of the grade they are assigned on the returned papers when it comes to final grade computation.
- 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., February 25 (tentative)
- Spring Break – March 3-8
- Withdraw Deadline – Tuesday, March 18
- Final – Tues., April 29, 1:00PM–3:50PM

Evaluation (tentative)

- Mid Term – 125 points ; Final – 175 points
- Assignments – 100 points;
  Paper and Presentation – 75 points
- Extra – 25 points used to increase weight of exams or assignments, 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)

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 well via purely computational means.

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 problem’s inherent complexity

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.

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 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 changed by some 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.

History

The Quest for Mechanizing Mathematics

Hilbert, Russell and Whitehead

- Late 1800’s to 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.
- 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

Goals on Computability

- Provide characterizations (computational models) of the class of effective procedures / algorithms.
- Study the boundaries between complete (or so it seems) 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.
- Apply results to various other areas of CS.

Basic Definitions

The Preliminaries

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, Problems & Predicates

- 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”.
- Predicate -- A mapping from some universe **U** into the Boolean set {**true**, **false**}. A predicate need not be defined for all values in **U**.

How They relate

- Let S be an arbitrary subset of some universe **U**. The predicate χ_S over **U** may be defined by:

  $\chi_S(\mathbf{x}) = \mathbf{true}$ if and only if $\mathbf{x} \in \mathbf{S}$

  χ_S is called the characteristic function of **S**.
- Let **K** be some arbitrary predicate defined over some universe **U**. The problem $\mathbf{P_K}$ associated with **K** is the problem to decide of an arbitrary member **x** of **U**, whether or not **K(x)** is true.
- Let **P** be an arbitrary decision problem and let **U** denote the set of questions in **P** (usually just the set over which a single variable part of the questions ranges). The set $\mathbf{S_P}$ associated with **P** is

  { x | x ∈ **U** and **x** has answer “yes” in **P** }

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

Existence of Undecidables

- A counting argument
  - The number of mappings from $\aleph$ to $\aleph$ is at least as great as the number of subsets of $\aleph$. But the number of subsets of $\aleph$ 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

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 ℵ, 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”.

Immediate Implications

- **P** solved implies **P** solvable implies **P** semi-decidable (re).
- **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.

Hilbert’s Tenth

Diophantine Equations are Unsolvable

One Variable Diophantine Equations are Solvable

Diophantine Equations

- Polynomial equation: $\mathbf{P(x_1, x_2, \ldots, x_n) = 0}$
- Coefficients and exponents in **P** must be positive integer constants
- Implied existential quantifier for each variable, indicating a selection of integral values for each of $\mathbf{x_1, x_2, \ldots, x_n}$
- Answer is "yes" if $\mathbf{P(a_1, a_2, \ldots, a_n)}$ evaluates to **0**, for some integer assignment $\mathbf{x_1=a_1, x_2=a_2, \ldots, x_n=a_n}$

Hilbert's 10th is Semi-Decidable

- Consider over one variable: $\mathbf{P(x) = 0}$
- 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 $\mathbf{P(x_1,x_2,..,x_k) = 0}$
  Think about how to do this correctly and what the pitfalls are

P(x) = 0 is Decidable

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

P(x) = 0 is Decidable

- Can bound the search to values of **x** in range [± **n** * ($\mathbf{c_{max}}$ / $\mathbf{c_n}$)], where
  n = highest order exponent in **P(x)**
  $\mathbf{c_{max}}$ = largest absolute value coefficient
  $\mathbf{c_n}$ = 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 $\mathbf{P(x_1,x_2,..,x_k) = 0}$.

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 $f(x)\downarrow$.

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 $f(x)\uparrow$.

Of course, if $f(x)\downarrow$ then f defines a value for x. In fact we also say that $f(x)$ is defined if $f(x)\downarrow$ and undefined if $f(x)\uparrow$.

Finally, we define the domain of f as $\{x \mid f(x)\downarrow\}$.
The range of f is $\{y \mid f(x)\downarrow$ and $f(x) = y\}$.

Halting Problem

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

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

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

$$\mathbf{pair(x,y) = <x,y> = 2^x \ (2y + 1) - 1}$$

with inverses

$$\mathbf{<z>_1 = \log_2(z+1)}$$

$$\mathbf{<z>_2 = (((\ z + 1\ ) // 2^{<z>_1}\ ) - 1\ ) // 2}$$

The Contradiction

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

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

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**
⇔ $\varphi_d(d)\uparrow$
⇔ **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 $\varphi_{\mathbf{x}}$ is the **x**-th procedure and $\varphi_{\mathbf{x}}\mathbf{(y)}$ 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,

$$\mathbf{Univ(x,y) = \varphi_x(y)}$$

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 $\mathbf{G(x) = Univ (F(x) , x) + 1 = \varphi_{F(x)}(x) = F_x(x) + 1}$

- But then **G** is itself an algorithm. Assume it is the **g**-th one

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

  Then, $\mathbf{G(g) = F_g(g) + 1 = G(g) + 1}$

- 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.

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 primitive recursion/iteration?
- 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

Basic Description

- We will use a simplified form that is a variant of Post's and Turing's models.
- Here, each machine is represented by a finite set of states 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 print **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.

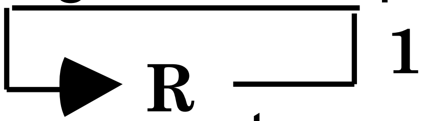
Base Machines

- **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

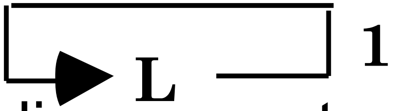
ℛ -- move right to next **0** (not including current square)

…$\underline{?}$11…10… ⇒ …?11…1$\underline{0}$…



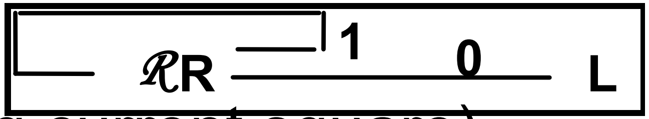
ℒ -- move left to next **0** (not including current square)

…011…1$\underline{?}$… ⇒ …$\underline{0}$11…1?…



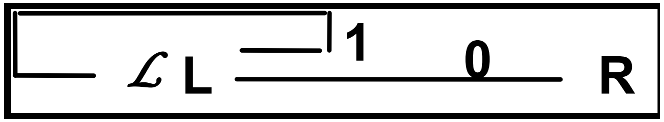
R -- move right to next **00** (not including current square)

…$\underline{?}$11…1011…10…11…100… ⇒ …?11…1011…10… 11…1$\underline{0}$0…



L -- move left to next **00** (not including current square)

…0011…1011…10…11…1$\underline{?}$… ⇒ …0$\underline{0}$11…1011…10… 11…1?…



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

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

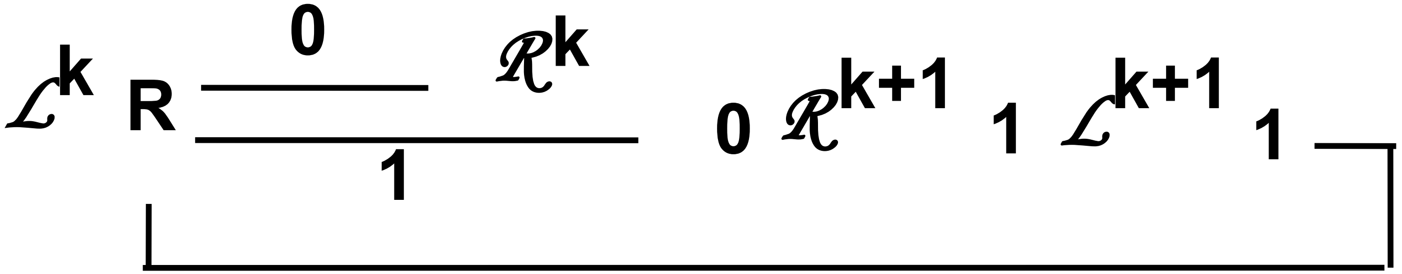
and ends with

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

where **y = F(x1, … , xn)**.

Addition by TM

Need the copy family of useful submachines, where $\mathbf{C_k}$ copies **k**-th preceding value.



The add machine is then

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

Turing Machine Variations

- Two tracks
- N tracks
- Non-deterministic (We will return to this)
- 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 the first n registers and ends with the result in the **n+1**-st 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+n+1**-th register, for any **k**, such that there are at least k**+n+1** registers.
- Sometimes, we use the notation of finishing with the results in the first register, and the arguments appearing in **2** to **n+1**.

Register Instructions

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

  $\mathbf{INC_r[i]}$ –

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

  $\mathbf{DEC_r[p, z]}$ –

  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 (r3 ← r1 + r2)

```
1.  DEC3[1,2]     : Zero result (r3) and work (r4) registers
2.  DEC4[2,3]
3.  DEC1[4,6]     : Add r1 to r3, saving original r1 in r4
4.  INC3[5]
5.  INC4[3]
6.  DEC4[7,8]     : Restore r1
7.  INC1[6]
8.  DEC2[9,11]    : Add r2 to r3, saving original r2 in r4
9.  INC3[10]
10. INC4[8]
11. DEC4[12,13]   : Restore r2
12. INC2[11]
13.               : Halt by branching here
```

Limited Subtraction by RM

Subtraction (r3 ← r1 - r2, if r1≥r2; 0, otherwise)

```
1.  DEC3[1,2]      : Zero result (r3) and work (r4) registers
2.  DEC4[2,3]
3.  DEC1[4,6]      : Add r1 to r3, saving original r1 in r4
4.  INC3[5]
5.  INC4[3]
6.  DEC4[7,8]      : Restore r1
7.  INC1[6]
8.  DEC2[9,11]     : Subtract r2 from r3, saving original r2 in r4
9.  DEC3[10,10]    : Note that decrementing 0 does nothing
10. INC4[8]
11. DEC4[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 $3^{x1}5^{x2}$ becomes 2^{x1+x2}

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

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 $\mathbf{2^{max(0,x1-x2)}}$

$$\begin{aligned} &3 \cdot 5x \rightarrow x \\ &3x \rightarrow 2x \\ &5x \rightarrow x \end{aligned}$$

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

$$
\begin{aligned}
&\mathbf{3\cdot 5x} \rightarrow \mathbf{x}\\
&\mathbf{3x} \rightarrow \mathbf{2x}\\
&\mathbf{5x} \rightarrow \mathbf{x}
\end{aligned}
$$

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

$$\mathbf{135 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2}$$

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

$$
\begin{aligned}
&\mathbf{135 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2}\\
&\mathbf{135 \Rightarrow 90 \Rightarrow 60 \Rightarrow 40 \Rightarrow 8 = 2^3}\\
&\mathbf{135 \Rightarrow 45 \Rightarrow 3 \Rightarrow 2 = 2^1}\\
&\mathbf{135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 1 = 2^0}\\
&\mathbf{135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 5 \Rightarrow 1 = 2^0}\\
&\mathbf{135 \Rightarrow 45 \Rightarrow 15 \Rightarrow 3 \Rightarrow 2 = 2^1}\\
&\mathbf{135 \Rightarrow 45 \Rightarrow 9 \Rightarrow 6 \Rightarrow 4 = 2^2}\\
&\mathbf{135 \Rightarrow 90 \Rightarrow 60 \Rightarrow 40 \Rightarrow 8 = 2^3}
\end{aligned}
$$

…

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

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

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

starting with the number **2**.

Sample RM and FRS

Present a Register Machine that computes IsOdd. Assume R2=x; at termination, set R2=1 if x is odd; 0 otherwise.

1. DEC2[2, 4]
2. DEC2[1, 3]
3. INC2[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 → x

3 x → 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:
  - $a x + c \quad \rightarrow \quad b x + d$
- 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.

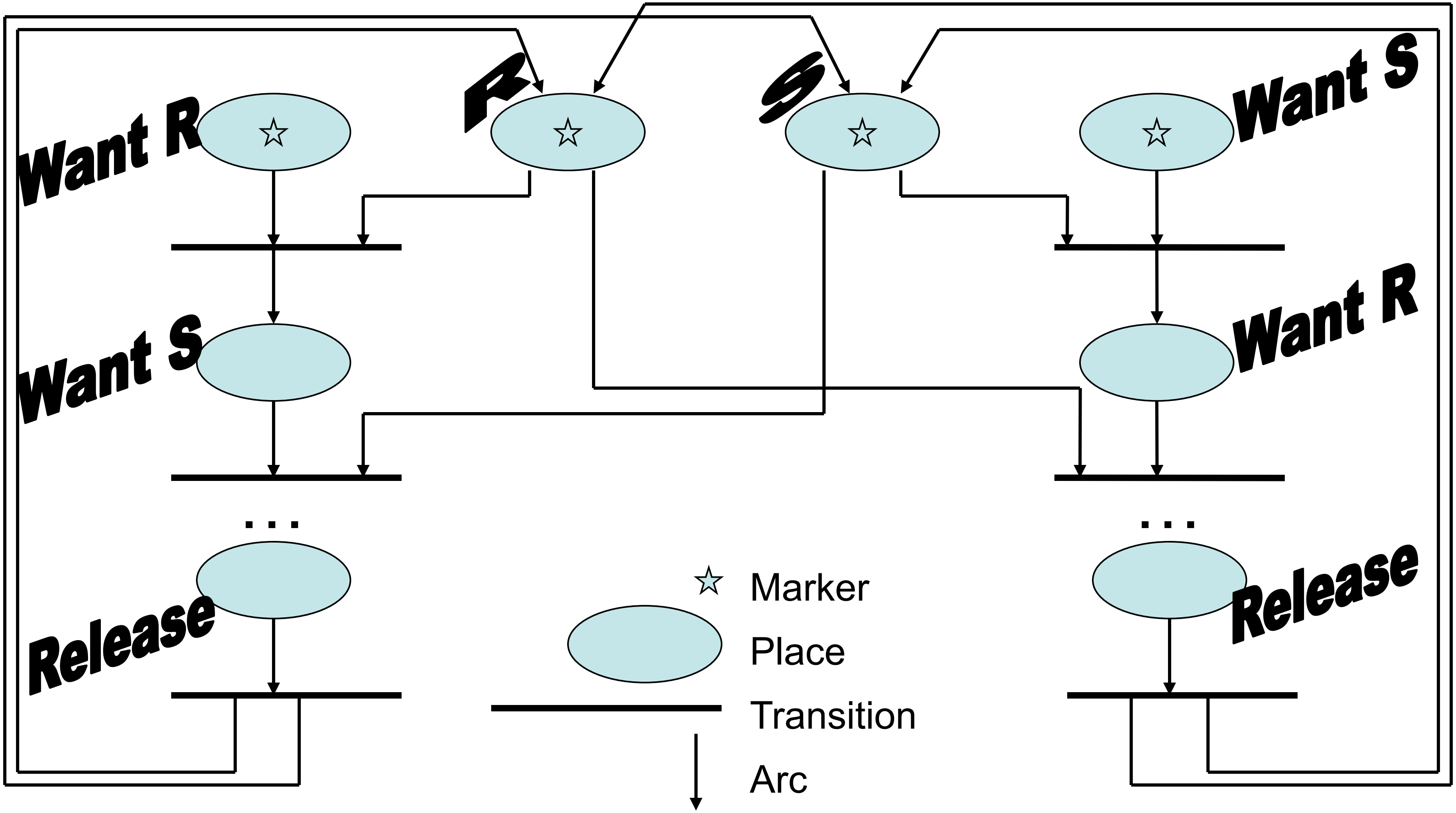
Petri Net Computation

- 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



Want R

R

S

Want S

Want S

Want R

. . .

. . .

Release

Release

☆ Marker

Place

Transition

Arc

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 **x,y** ∈ **G** then **x • y** ∈ **G**
2. Associativity: **x • (y • z) = (x • y) • z**

S is a monoid if

3. Identity: ∃**e** ∈ **G** ∀**x** ∈ **G** **[e • x = x • e = x]**

S is a group if

4. Inverse: ∀**x** ∈ **G** ∃$\mathbf{x^{-1}}$ ∈ **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**, **H_1**, … , **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

$x == 0$:

$\quad 0 == 0 = 1$

$\quad (y+1) == 0 = 0$

$x == y$:

$\quad x{==}y = ((x - y) + (y - x)) == 0$

$x \leq y$:

$\quad x{\leq}y = (x - y) == 0$

$x \geq y$:

$\quad x{\geq}y = y{\leq}x$

$x > y$:

$\quad x{>}y = \sim(x{\leq}y)$ /* See ~ on next page */

$x < y$:

$\quad x{<}y = \sim(x{\geq}y)$

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

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

$$\mathbf{f(x) = P(x) * g(x) + (1\text{-}P(x)) * h(x)}$$

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

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

Bounded Minimization 1

$f(x) = \mu\, z\ (z \le x)\ [\ P(z)\ ]$ if ∃ such a z,

$\qquad = x+1$, otherwise

where $P(z)$ is primitive recursive.

Can show f is primitive recursive by

$$
\begin{array}{llll}
f(0) & = & 1-P(0) & \\
f(x+1) & = & f(x) & \text{if } f(x) \le x \\
 & = & x+2-P(x+1) & \text{otherwise}
\end{array}
$$

Bounded Minimization 2

$\mathbf{f(x)} = \mu\, \mathbf{z\ (z < x)\ [\ P(z)\ ]}$ if ∃ such a $\mathbf{z}$,
        $= \mathbf{x}$, otherwise
where $\mathbf{P(z)}$ is primitive recursive.

Can show **f** is primitive recursive by
$\mathbf{f(0) = 0}$
$\mathbf{f(x+1)} = \mu\, \mathbf{z\ (z \le x)\ [\ P(z)\ ]}$

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

$x \wedge y$:

$x \wedge 0 = 1$

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

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

$\exp(x,i) = \mu z \ (z<x) \ [\sim(p_i \wedge (z+1) \mid x)]$

Pairing Functions

- **pair(x,y) = <x,y> = 2^x (2y + 1) – 1**
- with inverses

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

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

  <x,y,z> = <x, <y,z> > (note: stack analogy)

Pairing Function is 1-1 Onto

Prove that the pairing function <x,y> = 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, <x,y>+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 $y \geq 0$; 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 $<x,y> = 2^x(2y+1)-1$.

Moreover, $<2^x(2y+1)-1>_1$ = Factor($2^x(2y+1)$,0) = x, since 2y+1 must be odd, and

$<2^x(2y+1)-1>_2$ = (($2^x(2y+1)$/2^Factor($2^x(2y+1)$,0))-1)/2 = 2y/2 = y.

Thus, $(<x,y>_1,<x,y>_2) = (x,y)$, as was desired.

Let z be an arbitrary natural number, then the inverse of the pairing is $(<z>_1,<z>_2)$

Moreover, $<<z>_1,<z>_2>$ = 2^$<z>_1$ *(2$<z>_2$+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.

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, $\mathbf{R}_{\boldsymbol{M}}$ **=** { **h(d) | d** is an instance of an id of ***M*** }, and
  - $\mathbf{id'_0} \Rightarrow^*_{M'} \mathbf{h(id_0)}$, and $\mathbf{h(id_0)}$ is the only member of $\mathbf{R}_{\boldsymbol{M}}$ in the configurations encountered in this derivation.
  - $\mathbf{h(id_i)} \Rightarrow^+_{M'} \mathbf{h(id_{i+1})}$, $\mathbf{i \geq 0}$, and $\mathbf{h(id_{i+1})}$ is the only member of $\mathbf{R}_{\boldsymbol{M}}$ in this derivation.
- The above, in effect, provides an inductive proof that
  - $\mathbf{id_0} \Rightarrow^*_{M} \mathbf{id}$ implies $\mathbf{id'_0} \Rightarrow^*_{M'} \mathbf{h(id)}$, and
  - If $\mathbf{id'_0} \Rightarrow^*_{M'} \mathbf{id'}$ then either $\mathbf{id_0} \Rightarrow^*_{M} \mathbf{id}$, where $\mathbf{id' = h(id)}$, or $\mathbf{id'} \notin \mathbf{R_M}$

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 $\underline{\mathbf{0}}$ 0 0 …
- The underscore indicates the square being read. We denote this by the finite id
  1 0 1 0 0 1 1 q7 $\underline{\mathbf{0}}$
- 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 $\underline{\mathbf{0}}$ 0 0 …
  is just (**0, 83, 7**).
  … 0 0 1 0 q5 $\underline{\mathbf{1}}$ 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           : If (qj 0 R qk) is rule in TM
          MUL_r2__BY_2
          JUMP[set_k]
qj+1.     MUL_r1_BY_2           : If (qj 0 L qk) is rule in TM
          IF_r2_ODD then INC1
          DIV_r2__BY_2[set_k]
…
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 **r1**.
- Can also have a prolog that starts with arguments in first n registers 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 r1, 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.      INC2[4]       : putting answer in r2
4.      DIV_r1_BY_2[2] : strip a 1 from r1
5.      DEC1[5,6]     : Set r1 to 0 (prepare for answer)
6.      DEC2[6,7]     : Copy r2 to r1
7.   INC1[6]  :
8.                 : Answer is now in r1
```

REGISTER $\leq$ FACTOR

Encoding a RM's State

- This is a really easy one based on the fact that every member of **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 **p_i**'s are distinct primes and the **k_i**'s are non-zero values, except that the number **1** would be represented by **2^0**.
- Let R be an arbitrary **n**-register machine, having m instructions.

  Encode the contents of registers **r1,…,rn** by the powers of **$p_1,\ldots p_n$** .

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

  Use **pn+m+1** as prime factor that indicates simulation is done.
- This is in essence the 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]

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

j. DECr[s, f]

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

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

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

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

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.

Example of Order

Consider the simple machine to compute **r1:=r2 – r3** (limited)

1. **DEC3[2,3]**
2. **DEC2[1,1]**
3. **DEC2[4,5]**
4. **INC1[3]**
5.

Subtraction Encoding

Start with 3^x5^y7

$7 \cdot 5\,x \rightarrow 11\,x$

$7\,x \rightarrow 13\,x$

$11 \cdot 3\,x \rightarrow 7\,x$

$11\,x \rightarrow 7\,x$

$13 \cdot 3\,x \rightarrow 17\,x$

$13\,x \rightarrow 19\,x$

$17\,x \rightarrow 13 \cdot 2\,x$

$19\,x \rightarrow x$

Analysis of Problem

- If we don't obey the ordering here, we could take an input like **$3^5 5^2 7$** and immediately apply the second rule (the one that mimics a failed decrement).
- We then have **$3^5 5^2 13$**, signifying that we will mimic instruction number **3**, never having subtracted the **2** from **5**.
- Now, we mimic copying **r2** to **r1** and get **$2^5 5^2 19$** .
- 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 → 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

  $$\mathbf{RULE(F, x) = \mu\, z\, (1 \le z \le exp(F, 0)+1)\, [\, exp(F, 2*z-1) \mid x\,]}$$

- Note: if **x** is divisible by $\mathbf{a_i}$, and **i** is the least integer for which this is true, then $\mathbf{exp(F,2*i-1) = a_i}$ where $\mathbf{a_i}$ is the number of prime factors of **F** involving $\mathbf{p_{2i-1}}$. Thus, **RULE(F,x) = i**.

  If x is not divisible by any $\mathbf{a_i}$, $\mathbf{1 \le i \le n}$, 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

  NEXT(F, x) = (x // exp(F, 2*RULE(F, x)-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 only if we stop

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.

FRS Subtraction

- $2^0 3^a 5^b \Rightarrow 2^{a-b}$
  $3*5x \rightarrow x$ or $1/15$
  $5x \rightarrow x$ or $1/5$
  $3x \rightarrow 2x$ 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
- $\mathrm{RULE}(F, x) = \mu\ z\ (1 \le z \le 4)\ [\ \exp(F, 2*z-1) \mid x\ ]$
  $\mathrm{RULE}\ (F,3^4\ 5^2) = 1$, as 15 divides $3^4\ 5^2$
- $\mathrm{NEXT}(F, x) = (x\ //\ \exp(F, 2*\mathrm{RULE}(F, x)-1)) * \exp(F, 2*\mathrm{RULE}(F, x))$
  $\mathrm{NEXT}(F,3^4\ 5^2) = (3^4\ 5^2\ //\ 15 * 1) = 3^3 5^1$
  $\mathrm{NEXT}(F,3^3\ 5^1) = (3^3\ 5^1\ //\ 15 * 1) = 3^2$
  $\mathrm{NEXT}(F,3^2) = (3^2\ //\ 3 * 2) = 2^1 3^1$
  $\mathrm{NEXT}(F, 2^1 3^1) = (2^1 3^1\ //\ 3 * 2) = 2^2$
  $\mathrm{NEXT}(F, 2^2) = (2^2\ //\ 1 * 1) = 2^2$

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 ≤ 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

  $\ldots \mathbf{01^{x1}01^{x2}0\ldots01^{xn}\underline{0}}\ldots$

  and ends with

  $\ldots \mathbf{01^{x1}01^{x2}0\ldots01^{xn}01^{y}\underline{0}}\ldots$

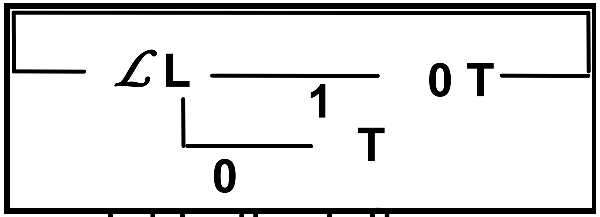
  where **y = F(x1, … , xn)**.

More Helpers

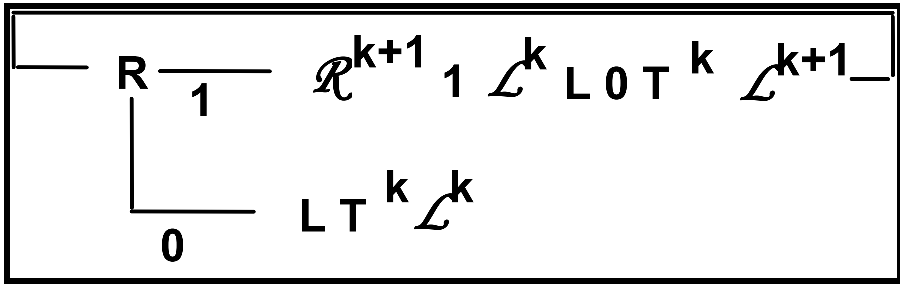
- To build our simulation we need to construct some useful submachines, in addition to the $\mathcal{R}$, $\mathcal{L}$, **R**, **L**, and $\mathbf{C}_k$ machines already defined.
- **T** -- translate moves a value left one tape square
  $\ldots\underline{?}01^x0\ldots \Rightarrow \ldots ?1^x\underline{0}0\ldots$

R1 $\mathcal{R}$L0

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


$\mathcal{L}$L
1
0 T
T
0

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


R
1
$\mathcal{R}^{k+1}$ 1 $\mathcal{L}^k$ L 0 T^k $\mathcal{L}^{k+1}$
0
L T^k $\mathcal{L}^k$

Basic Functions

All Basis Recursive Functions are Turing computable:

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

$(R1)^a R$

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

C_{n-i+1}

- $S(x) = x+1$

$C_1 1R$

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

$$\mathcal{L}^{n+m}\ (\mathrm{Rot}_{n+m})^n\ \mathcal{R}^{n+m}\ E\ \ \mathcal{L}^{n+m+1}\ (\mathrm{Rot}_{n+m})^m\ \mathcal{R}^{n+m+1}$$

Can now define **F** by

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

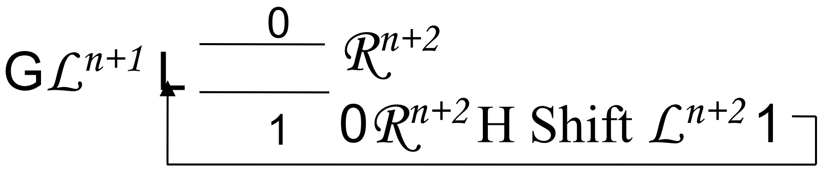
Closure Under Induction

To prove the 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:



G $\mathcal{L}^{n+1}$ L

0 $\mathcal{R}^{n+2}$

1 0 $\mathcal{R}^{n+2}$ H Shift $\mathcal{L}^{n+2}$ 1

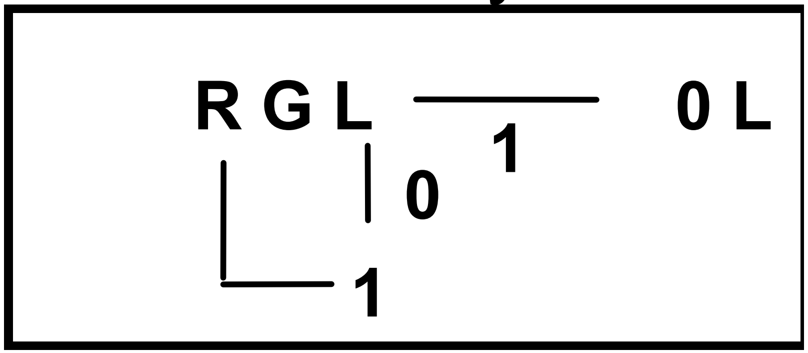
Since our Turing Machine simulator can produce the same value for any arbitrary PRF, F, we show that Turing Machines are closed under induction (primitive recursion).

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



R G L **1** **0 L**

0

1

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

- $\Phi^{(n)}(f, x_1,\ldots,x_n)$ = 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 the common **SNAP** (snapshot) with arguments permuted

  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.

Recursively Enumerable

Properties of re Sets

Definition of re

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

  S ⊆ ℵ 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 ⊆ ℵ is re iff **S =** ∅ 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 $\mathbf{g_S}$ 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 primitive recursive 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 ℵ, whose range is exactly **S**. Define

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

This achieves our result as the domain of ψ_S is the range of $\mathbf{F_S}$, or empty if **S == Ø**.

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 $\mathbf{g_S}$

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

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

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

  $$\mathbf{W_n = \{ x \in \aleph \mid \varphi(x,n)\downarrow \}}$$

- Theorem: A set **B** is re iff there exists an **n** such that $\mathbf{B = W_n}$.

  Proof: Follows from definition of $\varphi\mathbf{(x,n)}$.

- 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 **~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: Suppose **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

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 α $\mathbf{x} \rightarrow \beta$ $\mathbf{x}$ in **F**, for each rule in **F**
  $\mathbf{q\ x \rightarrow q\ x}$ ensures we loop at end
  $\mathbf{x \rightarrow q\ p_{m+1}^{u1} \ldots p_{m+n}^{un}}\ \sigma\ \mathbf{x}$
  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

- **S** is decidable iff there exists an algorithm χ_S (called **S**'s characteristic function) such that
  $\mathbf{x \in S \Leftrightarrow \chi_S(x)}$
  This is just the definition of decidable.
- **S** is re iff there exists an algorithm $\mathbf{A_S}$ where
  $\mathbf{x \in S \Leftrightarrow \exists t\ A_S(x,t)}$
  This is clear since, if $\mathbf{g_S}$ is the index of the procedure ψ_S defined earlier that semi-decides **S** then
  $\mathbf{x \in S \Leftrightarrow \exists t\ STP(g_S, x, t)}$
  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 $\mathbf{G(x) = Univ (F(x) , x) + 1 = \varphi(F(x), x)+1 = F_x(x) + 1}$
- But then **G** is itself an algorithm. Assume it is the **g**-th one

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

  Then, $\mathbf{G(g) = F_g(g) + 1 = G(g) + 1}$
- 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

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

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

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,

  $f \in \mathrm{TOT} \Leftrightarrow \forall x \exists t\ (\ \mathrm{STP}(\ f, x, t\ )\ )$

  and this is the minimum quantification we can use, given that the quantified predicate is recursive.

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, $\mathbf{K_0 \leq S}$. 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 **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 | ∀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** ∈ ℵ
- 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 **L_u** is the **Halting Problem**.
- The sets **L_{ne}** and **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

- **L_{ne}** is enumerated by

  F(<f, x, t>) = f * STP(f, x, t)

- This assumes that **0** is in **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 **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 **<x,y,z> = <x,<y,z>>**.
- Now $\mathbf{L_{ne}}$ cannot be recursive, for if it were then $\mathbf{L_u}$ 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**. Hence, this new function's index is in $\mathbf{L_{ne}}$ just in case (**x**, **y**) is in $\mathbf{L_u}$.
- Thus, a decision procedure for $\mathbf{L_{ne}}$ (equivalently for $\mathbf{L_e}$) implies one for $\mathbf{L_u}$.

L_{ne} is re by Quantification

- Can do by observing that

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

- 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

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

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**,
  A $\leq_m$ **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**,
  A $\equiv_m$ **B**, if **A** $\leq_m$ **B** and **B** $\leq_m$ **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: ∅, ℵ, 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**, $\mathbf{A} \leq_1 (\leq_m , \leq_t) \mathbf{C}$.
- 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) ↓}**
- 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** ∈ **A** iff **x** ∈ **dom(φ_a)** iff φ_a**(x)**↓ iff **<a,x>** ∈ **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

- **K = { f | φ_f(f) is defined }**
- Define **f_x(y) = φ_f(x)**. That is, **f_x(y) = φ_f(x)**. The index for **f_x** can be computed from **f** and **x** using **$S_{1,1}$**, where we add a dummy argument, **y**, to **φ_f**. Let that index be **f_x**. (Yeah, that's overloading.)
- **<f,x>** ∈ **K_0** iff **x** ∈ **dom(φ_f)** iff **$\forall$y[φ_{f_x}(y)$\downarrow$]** iff **f_x** ∈ **K**.
- The above provides a 1-1 function that reduces **K_0** to **K**.
- Since **K_0** is an re (1-1, m-1, Turing) complete set and **K** is re, then **K** is also re (1-1, m-1, Turing) complete.

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-1

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-1

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, **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 **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 **P** be some non-trivial I/O property of the indices of recursive functions. Then

$$\mathbf{S_P = \{ x \mid \varphi_x \text{ has property } P) \}}$$

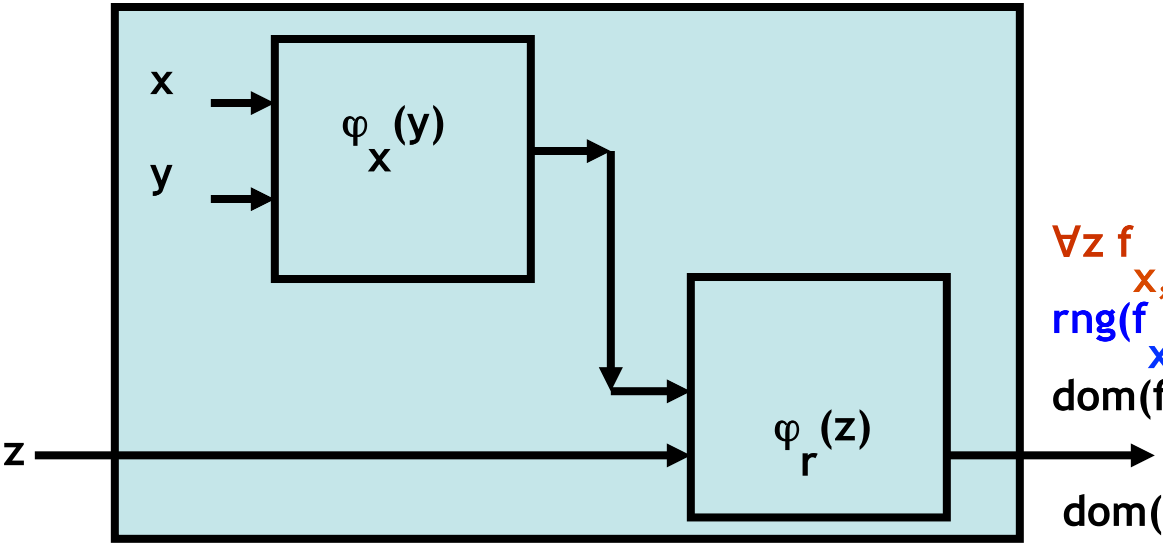
is undecidable. Note that membership in $\mathbf{S_P}$ is based purely on the input/output behavior of a function, not on any aspect of its implementation.

Strong Rice's Proof

- Given **x**, **y**, **r**, where **r** is in the set $\mathbf{S_{P}}$.= {**f** | φ_f has property **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** $\varphi_x(y)\downarrow$; $= \phi$ **if** $\varphi_x(y)\uparrow$. Thus, $\varphi_x(y)\downarrow$ iff $\mathbf{f_{x,y,r}}$ has property P, and so $\mathbf{K_0 \leq S_P}$.

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 = Ø**

b) **L** is finite

c) **L** is a regular set

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

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 | ∃K ∀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 | ∃K ∀$C_{|C|≤K}$ [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**.

Exam#1 Review

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 ∀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) **$A \cup B$**

   b) **$A \cap B$**

   c) **~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 φ_f**(x)** converges in **s** or fewer steps and, if so, **VALUE(f, x, s) =** φ_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

   E = { z | z = x + y, where x ∈ S and y ∈ 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 | ∀x,** φ_f**(x+1)>**φ_f**(x) }**.
   Let **TOT = { f | ∀x,** φ_f**(x)**↓ **}**.
   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 **$<x,y>$** and its two inverses **$<z>_1$** and **$<z>_2$**, where if **$z = <x,y>$**, then **$x = <z>_1$** and **$y = <z>_2$**.

Sample Question#11

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

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**?

Exam#1 Topics

Exclusions

- No explicit Turing Machines to write
- No explicit FRSs to write
- No explicit Register Machines to write
- No explicit S-m-n theorem, but assumed ability to apply it

Inclusions

- 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

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 = (Σ, 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 ⇒* 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 $\Sigma = \{a,b,\#\}$. Semi-Thue rules:
  aba → b
  #b# → λ
  For above, $\#a^n ba^m\# \Rightarrow^* \lambda$ iff $n=m$
- Alphabet $\Sigma = \{0,1,c,\#\}$. Normal rules:
  0c → 1
  1c → c0
  #c → #1
  0 → 0
  1 → 1
  # → #
  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 αqaβ is represented by the string hαqaβh, a being the scanned symbol.
- The tuple q a b s leads to
  qa → sb
- Moving right and left can be harder due to blanks.

Details of Halt(TM) ≤ Word(ST)

- Let M = (Q, {0,1}, T), T is Turing table.
- If qabs ∈ T, add rule qa → sb
- If qaRs ∈ T, add rules
  - q1b → 1sb   a=1, ∀b∈{0,1}
  - q1h → 1s0h   a=1
  - cq0b → c0sb   a=0, ∀b,c∈{0,1}
  - hq0b → hsb   a=0, ∀b∈{0,1}
  - cq0h → c0s0h   a=0, ∀c∈{0,1}
  - hq0h → hs0h   a=0
- If qaLs ∈ T, add rules
  - bqac → sbac   ∀a,b,c∈{0,1}
  - hqac → hs0ac   ∀a,c∈{0,1}
  - bq1h → sb1h   a=1, ∀b∈{0,1}
  - hq1h → hs01h   a=1
  - bq0h → sbh   a=0, ∀b∈{0,1}
  - hq0h → 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
  - $q_0a \rightarrow q_0$ $\forall a \in \{0,1\}$
  - $bq_0 \rightarrow q_0$ $\forall b \in \{0,1\}$

- 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:
- $h1^x q_1 0h \Rightarrow_{\sum(M)}^* hq_0h$ iff $x \in \mathcal{L}(M)$.
- $hq_0h \Rightarrow_{\prod(M)}^* h1^x q_1 0h$ iff $x \in \mathcal{L}(M)$.
- $hq_0h \Leftrightarrow_{\sum (M)}^* h1^x q_1 0h$ 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 Review

Pretty Basic Stuff

Closure Properties

- Regular (Finite State) Languages
  - Union, intersection, complement, substitution, quotient (with anything), max, min, cycle, reversal
  - Use of Pumping Lemma and Myhill-Nerode
- Context Free
  - Union, intersection with regular, substitution, quotient with regular, cycle, reversal
  - Use of Pumping and Ogden's Lemma
- Context Sensitive Languages
  - Union, intersection, complement, Epsilon-free substitution, cycle, reversal

Non-Closure

- CFLs not closed under
  - Intersection, complement, max, min
- CSLs not closed under
  - Homomorphism (or substitution with empty string), max (similar to homomorphism)

Grammars and re Sets

- Every grammar lists an re set.
- Some grammars (regular, CFL and CSG) produce recursive sets.
- Type 0 grammars are as powerful at listing re sets as Turing machines are at enumerating re sets (Proof later).

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
$(x_1, \ldots , x_n), (y_1, \ldots , y_n)$ over Σ,
does there exist a sequence $i_1, \ldots , i_k$, $k>0$, $1 \leq i_j \leq n$, such that
$x_{i_1} \ldots x_{i_k} = y_{i_1} \ldots y_{i_k}$?
- 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 → ab; a → aa; b → a
  - Instance of word problem: bbbb ⇒*? aa
- Convert to PCP

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

  $\mathbf{S \rightarrow A \mid B}$

  $\mathbf{A \rightarrow x_i\ A\ [i] \mid x_i\ [i]} \qquad \mathbf{1 \leq i \leq n}$

  $\mathbf{B \rightarrow y_i\ B\ [i] \mid y_i\ [i]} \qquad \mathbf{1 \leq i \leq n}$

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

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

- 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

$$
\begin{aligned}
S &\rightarrow x_i\, S\, y_i^R \mid x_i\, T\, y_i^R \quad 1 \le i \le n \\
a\, T\, a &\rightarrow *\, T\, * \\
*\, a &\rightarrow a\, * \\
a\, * &\rightarrow *\, a \\
T &\rightarrow *
\end{aligned}
$$

- 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 (Valid Computations)

- A trace of a machine M, is a word of the form

  $\# X_0 \# X_1 \# X_2 \# X_3 \# \ldots \# X_{k-1} \# X_k \#$

  where $X_i \Rightarrow X_{i+1}$ $0 \le i < k$, X_0 is a starting configuration and X_k is a terminating configuration.
- We allow some laxness, where the configurations might be encoded in a convenient manner. Many texts show that 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 to intersect the two languages, so the result is a trace. This then allows us to create CFLs L1 and L2, where L1 ∩ L2 ≠ Ø , 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.

Traces of FRS

- 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. $\mathrm{INC}_r[i]$ $\quad p_{n+j}\ x \rightarrow p_{n+i}\ p_r\ x$
  j. $\mathrm{DEC}_r[s, f]$ $\quad p_{n+j}\ p_r\ x \rightarrow p_{n+s}\ x$
  $\quad p_{n+j}\ pr\ x + k\ p_{n+j} \rightarrow p_{n+f}\ p_r\ x + k\ p_{n+f}$, $1 \le k < p_r$
  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 → $1^{ai}F_i1^{ci}$ | $1^{ai+bi}\#1^{ci+di}$ $1 \le i \le k$

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

  S_2 → $\# 1^{x0}S_11^{z0}\#$ 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.

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 $X_i \Rightarrow X_{i+1}$, $0 \leq i < 2k$, 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.

Quotients of CFLs (concept)

Let 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.
Now, let L2 = L(G2) = {$X_0 \$ \# X_0 \# X_1 \# X_2 \# X_3 \# X_4 \# \ldots \# X_{2k-1} \# X_{2k} \# Z_0 \#$}
where $X_{2i-1} \Rightarrow X_{2i}$, $1 \le i \le k$ 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 ways 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 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.

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

  $T_1 \rightarrow \# F_i\ T_1 \mid \# F_i\ \# \qquad 1 \le i \le k$

  $A \rightarrow 1\ A\ 1 \mid \$\ \#$

  $S_1 \rightarrow \$T_1$

  $S_2 \rightarrow A\ T_1\ \#\ 1^{z0}\ \#$ **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,

  $L1 = L(\ G1\ ) = \{\ \$\ \#Y_0\ \#\ Y_1\ \#\ Y_2\ \#\ Y_3\ \#\ \ldots\ \#\ Y_{2j}\ \#\ Y_{2j+1}\ \#\ \}$

  where $Y_{2i} \Rightarrow Y_{2i+1}$, $0 \le i \le j$.

  This checks the even/odd steps of an even length computation.

  But, $L2 = L(\ G2\ ) = \{\ X\ \$\ \ \#X_0\ \#\ X_1\ \#\ X_2\ \#\ X_3\ \#\ X_4\ \#\ \ldots\ \#\ X_{2k-1}\ \#\ X_{2k}\#\ Z_0\ \#\ \}$

  where $X_{2i-1} \Rightarrow X_{2i}$, $1 \le i \le k$ 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 $.

Finish Quotient

Now, consider the quotient of L2 / L1 . The only ways 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 actually easier to show a simulation of a Turing machine than of a Factor System.
- 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 = Yq0aX 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 ∪ {S’, D}, T ∪ {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 $\lvert\alpha\rvert \leq \lvert\beta\rvert$**
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 ∈ L(G) and m≥0 is some integer }
- For each w ∈ L(G), we cannot, in general, determine for which values of m, e w e dm ∈ L(G’). We would need to ask a potentially infinite number of questions of the form
“does e w e d^m ∈ L(G’)” to determine if w ∈ L(G). That’s a semi-decision procedure.

Some Consequences

- CSGs are not closed under Init, Final, Mid, quotient with regular sets and homomorphism (okay for λ-free homomorphism)
- 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 =$\emptyset$, for CSL, L?
- Is L=Σ^*, 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 $L_1 \cap L_2 = \emptyset$ for CFLs (CSLs), L_1, L_2?
- Is L regular, for CFL (CSL), L?
- Is $L_1 \cap L_2$ a CFL for CFLs, L_1, L_2?
- Is ~L CFL, for CFL, L?

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, L_1 and L_2 CFLs?

Word to Grammar Problem

- Recast semi-Thue system making all symbols non-terminal, adding S and V to non-terminals and terminal set $\Sigma=\{a\}$

  $G: S \rightarrow h1^xq_10h$

  $hq_0h \rightarrow V$

  $V \rightarrow aV$

  $V \rightarrow \lambda$

- $x \in \mathcal{L}(M)$ 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 Grammar

- Unsolvables
  - $\mathcal{L}(G) = \varnothing$
  - $\mathcal{L}(G) = \Sigma^*$
  - $\mathcal{L}(G)$ infinite
  - $w \in \mathcal{L}(G)$, for arbitrary w
  - $\mathcal{L}(G) \subseteq \mathcal{L}(G2)$
  - $\mathcal{L}(G) = \mathcal{L}(G2)$
- Latter two results follow when have
  - G2: $S \rightarrow aS \mid \lambda \quad a \in \Sigma$

Turing Machine Traces

- A valid trace
  - $C_1 \# C_2^R \$ C_3 \# C_4^R \ \ldots \$ C_{2k-1} \# C_{2k}^R \$$, where $k \geq 1$ and $C_i \Rightarrow_M C_{i+1}$, for $1 \leq i < 2k$. Here, $\Rightarrow_M$ means derive in M, and C^R means C with its characters reversed
- An invalid trace
  - $C_1 \# C_2^R \$ C_3 \# C_4^R \ \ldots \$ C_{2k-1} \# C_{2k}^R \$$, where $k \geq 1$ and for some i, it is false that $C_i \Rightarrow_M C_{i+1}$.

What’s Context Free?

- Given a Turing Machine M
  - The set of invalid traces of M is Context Free
  - The set of valid traces is Context Sensitive
  - The set of valid terminating traces is Context Sensitive
  - The complement of the valid traces is Context Free
  - The complement of the valid terminating traces is Context Free

Partially correct traces

L1 = L(G1) = { $\#Y_0 \# Y_1 \# Y_2 \# Y_3 \# \ldots \# Y_{2j} \# Y_{2j+1} \#$ }
where $Y_{2i} \Rightarrow Y_{2i+1}$, $0 \leq i \leq j$.
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 $X_{2i-1} \Rightarrow X_{2i}$, $1 \leq i \leq k$.
This checks the odd/steps of an even length computation.

L = L1 ∩ L2 describes correct traces (checked even/odd and odd/even). If Z_0 is chosen to be a terminal configuration, then these are terminating traces. If we pick a fixed X_0, then X_0 is a halting configuration iff L is non-empty. 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 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.

$L = \Sigma^*?$

- If L is regular, then $L = \Sigma^*?$ is decidable
  - Easy – Reduce to minimal deterministic FSA, $\mathbf{A}_L$ accepting L. $L = \Sigma^*$ iff $\mathbf{A}_L$ 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 Turing Machine's valid terminating traces

Undecidability of Finite Convergence for Operators on Formal 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 • L get us anything new?**
  - **i.e., Is L • L = L?**
- **Membership in a CSL is decidable.**
- **Claim is that $L = \Sigma^*$ iff**

  (1) $\Sigma \cup \{\lambda\} \subseteq L$; and

  (2) L • L = L
- **Clearly, if $L = \Sigma^*$ then (1) and (2) trivially hold.**
- **Conversely, we have $\Sigma^* \subseteq L^* = \cup_{n\geq0} L^n \subseteq L$**
  - **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 $\}$,
- $L_2 = \{ C_1\#C_2^R\$C_3\#C_4^R \ \ldots \ \$C_{2k-1}\#C_{2k}^R\$ \mid$ 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 $L_1^n \cap L_2 \neq \emptyset$ 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.

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 • 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 ▷ 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) ∩ a*c*b*d*
  - A ◇ B is not necessarily a CFL
  - A ◇* B is not necessarily a CFL
    - Deciding whether or not A ◇* 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

- $\exists k>0 \; L^{k} = L^{k+1}$
- $\exists k\geq 0 \; L\,(k) \rhd L = L\,(k+1) \rhd L$
- $\exists k\geq 0 \; L \rhd^{[\,k\,]} L = L \rhd^{[\,k+1\,]} L$
- $\exists k\geq 0 \; L \diamond^{k} L = L \diamond^{k+1} L$
- $\exists k\geq 0 \; L\,(k) \otimes_{c} L = L\,(k+1) \otimes_{c} L$
- $\exists k\geq 0 \; L\,(k) \otimes_{u} L = L\,(k+1) \otimes_{u} L$

- $\exists k\geq 0 \; A\,(k) \rhd B = A\,(k+1) \rhd B$
- $\exists k\geq 0 \; A \rhd^{[\,k\,]} B = A \rhd^{[\,k+1\,]} B$
- $\exists k\geq 0 \; A \diamond^{k} B = A \diamond^{k+1} B$
- $\exists k\geq 0 \; A\,(k) \otimes_{c} B = A\,(k+1) \otimes_{c} B$
- $\exists k\geq 0 \; A\,(k) \otimes_{u} B = A\,(k+1) \otimes_{u} L$

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 • L get us anything new?
  - i.e., Is L • L = L?
- Membership in a CSL is decidable.
- Claim is that $L = \Sigma^*$ iff

  (1) $\Sigma \cup \{\lambda\} \subseteq L$; and

  (2) L • L = L
- Clearly, if $L = \Sigma^*$ then (1) and (2) trivially hold.
- Conversely, we have $\Sigma^* \subseteq L^* = \cup_{n\geq0} L^n \subseteq L$
  - first inclusion follows from (1); second from (2)

Subsuming •

- Let ⊕ be any operation that subsumes concatenation, that is $A \bullet B \subseteq A \oplus B$.
- Simple insertion is such an operation, since $A \bullet B \subseteq A \rhd B$.
- 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 $L \bullet L \subseteq L \oplus L$

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?
- Is L regular, for CFL (CSL), L?
- Is $L_1 \cap L_2$ a CFL for CFLs, L_1, L_2?
- Is ~L CFL, for CFL, L?

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, L_1 and L_2 CFLs?

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.,
$(p1 \vee p2) \vee p3 \;\; \supset \;\; p1 \vee (p2 \vee p3)$

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 v Q), (R v ~Q)
  then can determine that (P v 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 ⊃ p) as form of bottom of stack
- Use (p ⊃ (p ⊃ p)) as form for letter 0
- Use (p ⊃ (p ⊃ (p ⊃ p))) as form for 1
- Etc.
- String 01 (reading top to bottom of stack) is
  - (((p ⊃ p) ⊃ ((p ⊃ p) ⊃ ((p ⊃ p) ⊃ (p ⊃ p)))) ⊃
    (((p ⊃ p) ⊃ ((p ⊃ p) ⊃ ((p ⊃ p) ⊃ (p ⊃ p)))) ⊃
    ((p ⊃ p) ⊃ ((p ⊃ p) ⊃ ((p ⊃ p) ⊃ (p ⊃ 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*** $[p \supset \xi_2 (p)]$

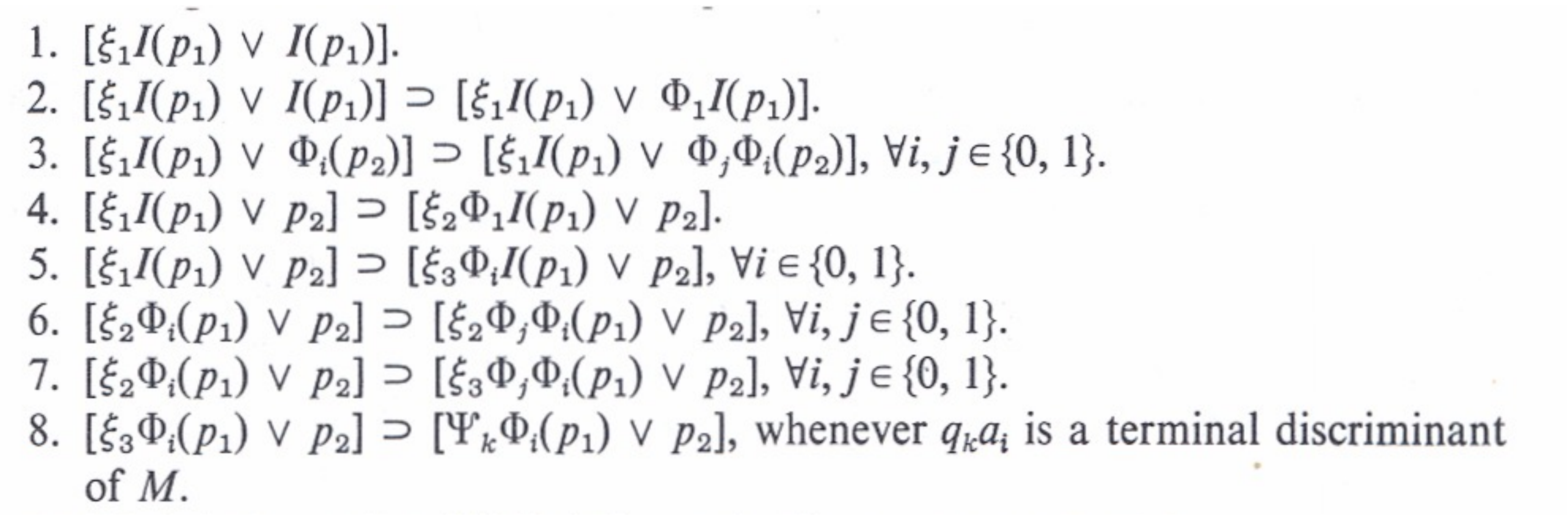
$\psi_1(p)$ ***is*** $[p \supset \xi_3 (p)]$

$\psi_2(p)$ ***is*** $[p \supset \psi_1 (p)]$

…

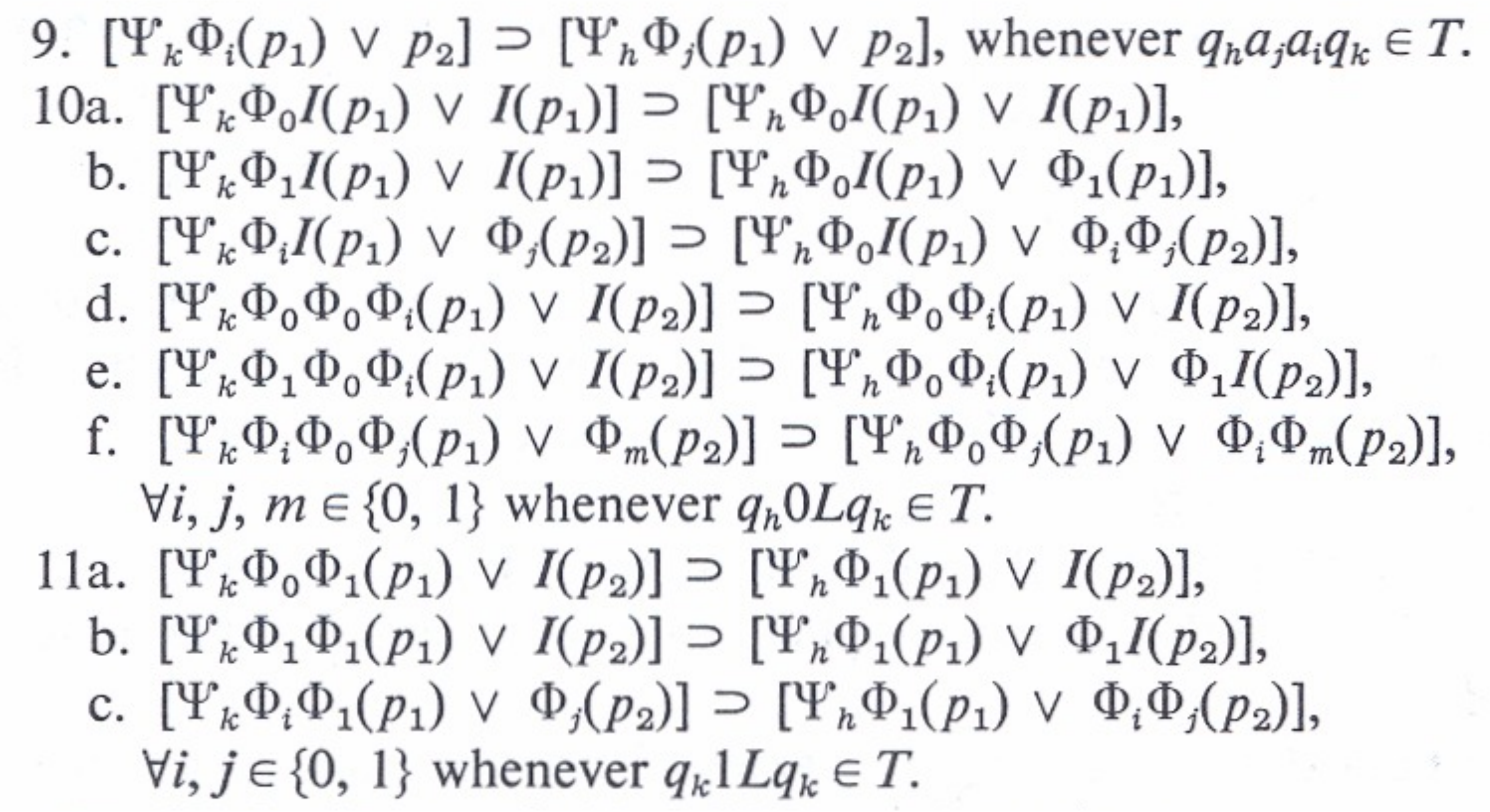
$\psi_m(p)$ ***is*** $[p \supset \psi_{m-1} (p)]$

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 $q_h 0 L q_k \in T$.

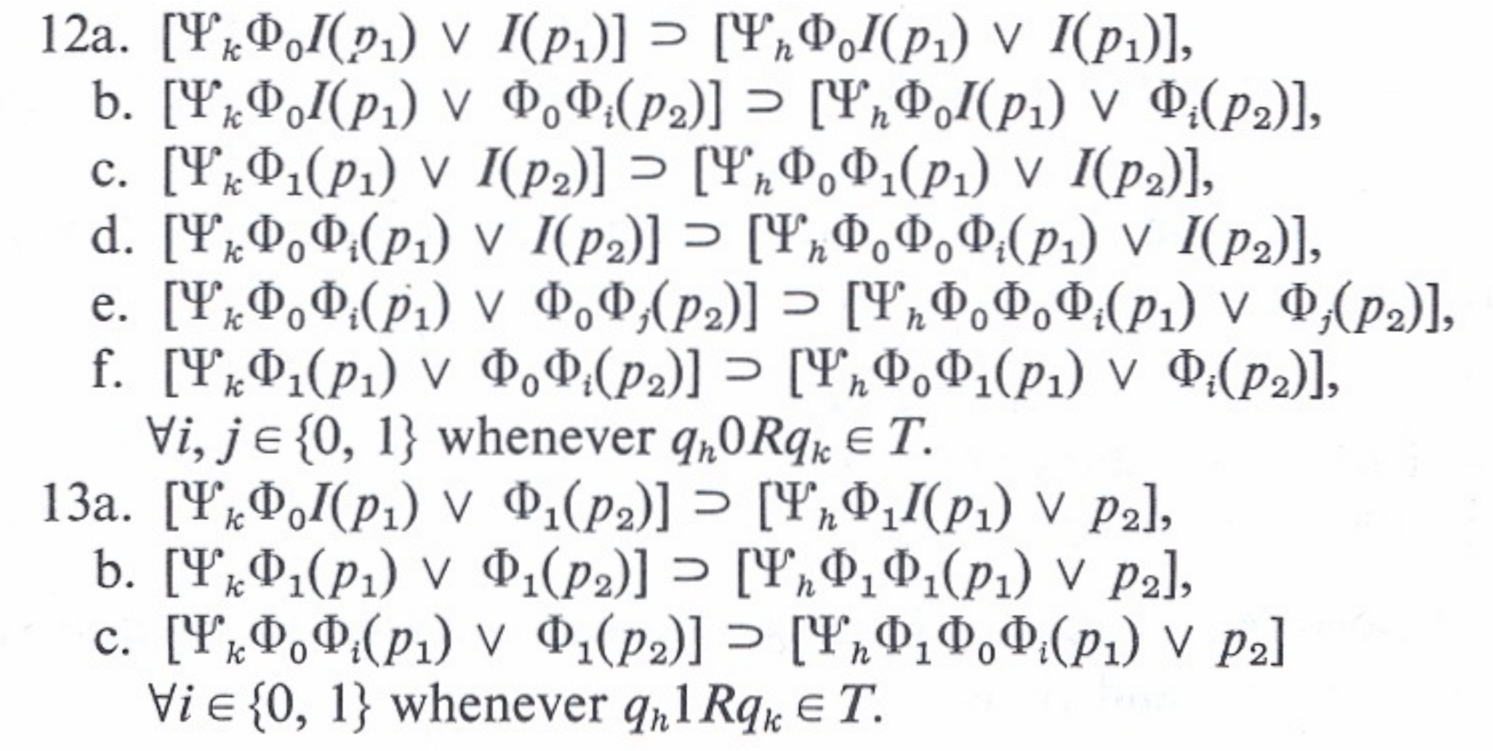
11a. $[\Psi'_k\Phi_0\Phi_1(p_1) \vee I(p_2)] \supset [\Psi'_h\Phi_1(p_1) \vee I(p_2)]$,

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 $q_h 1 R q_k \in T$.

ORDER ANALYSIS

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 **$O(f_x(n))$**.

Then, for algorithms **A** and **B** and their respective order functions, **$f_A(n)$** and **$f_B(n)$**, consider the limit of **$f_A(n)/f_B(n)$** 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.)

No 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 have not 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 **$O(n^k)$**, increasing the size of an instance by one gives a running time that is **$O((n+1)^k)$**

That's really not much more.

With an increase of one in an exponential algorithm, **$O(2^n)$** changes to **$O(2^{n+1}) = O(2*2^n) = 2*O(2^n)$** – 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 the 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.

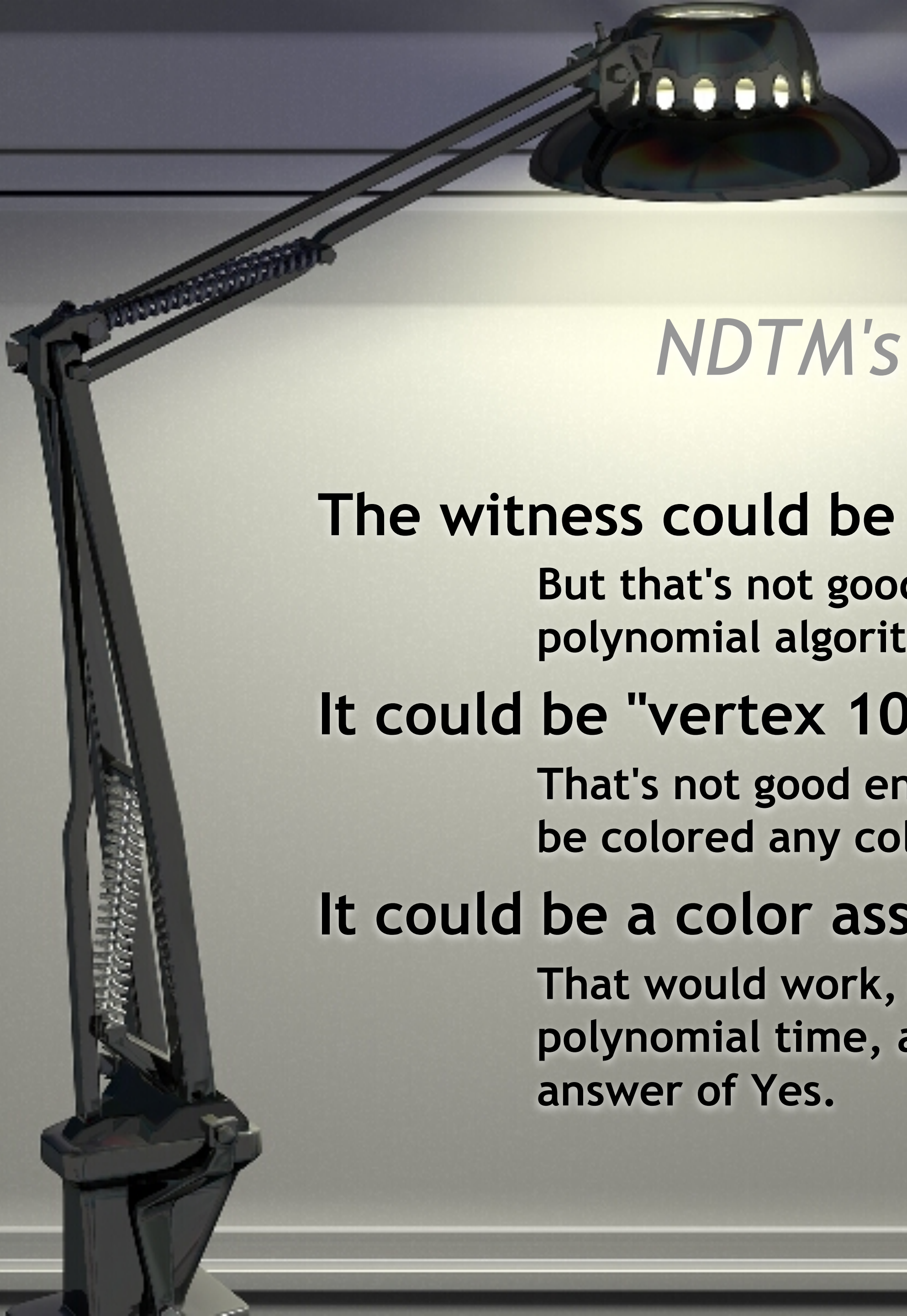
NDTM's - Witnesses

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:

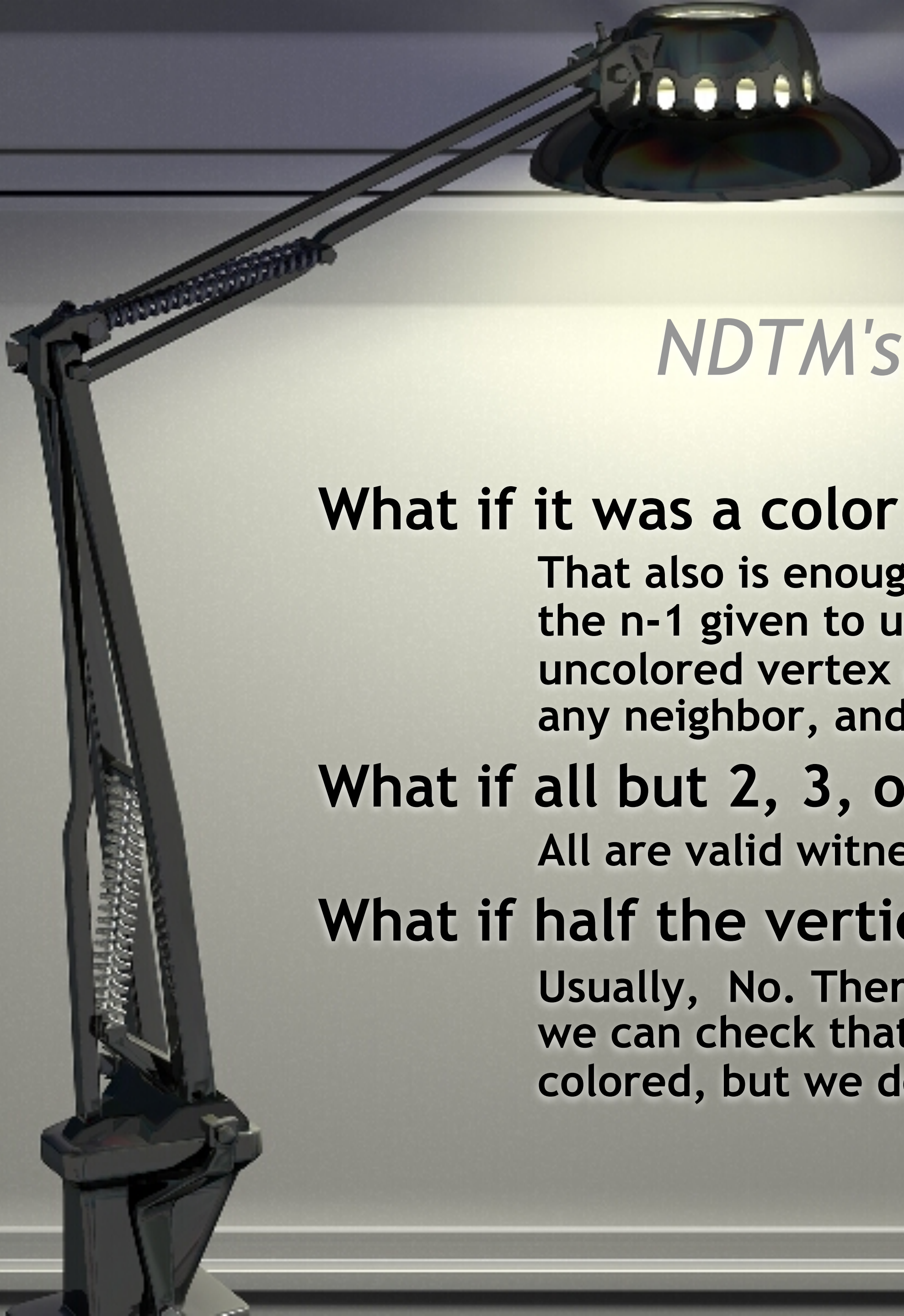
Given: A graph G = (V, E) and an integer k.

Question: Can the vertices of G be assigned colors so
that adjacent vertices have different
colors and use at most k colors?



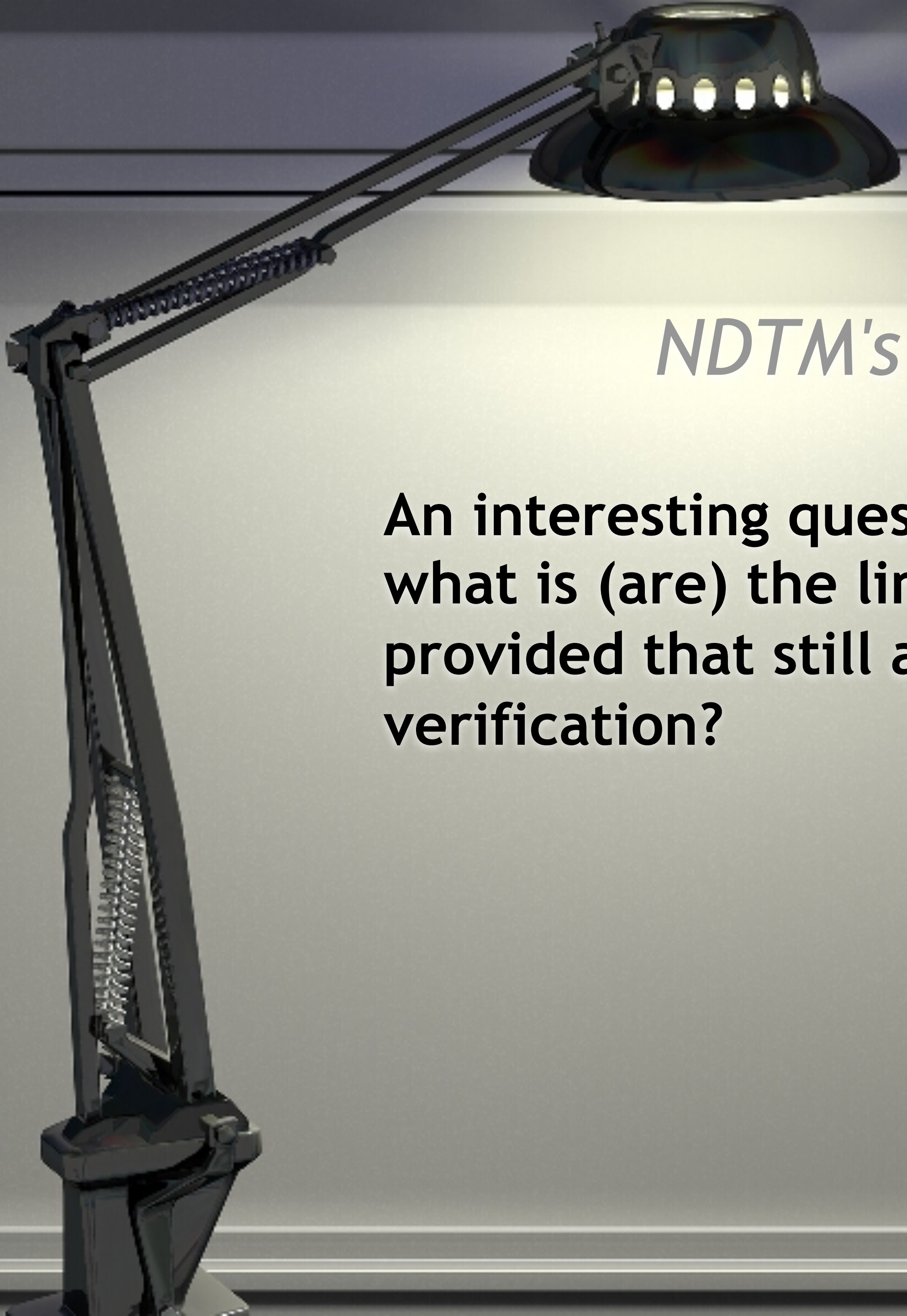
NDTM's - Witnesses

- 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 is (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 proven 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 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 Formal Languages and Automata 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 must 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.'

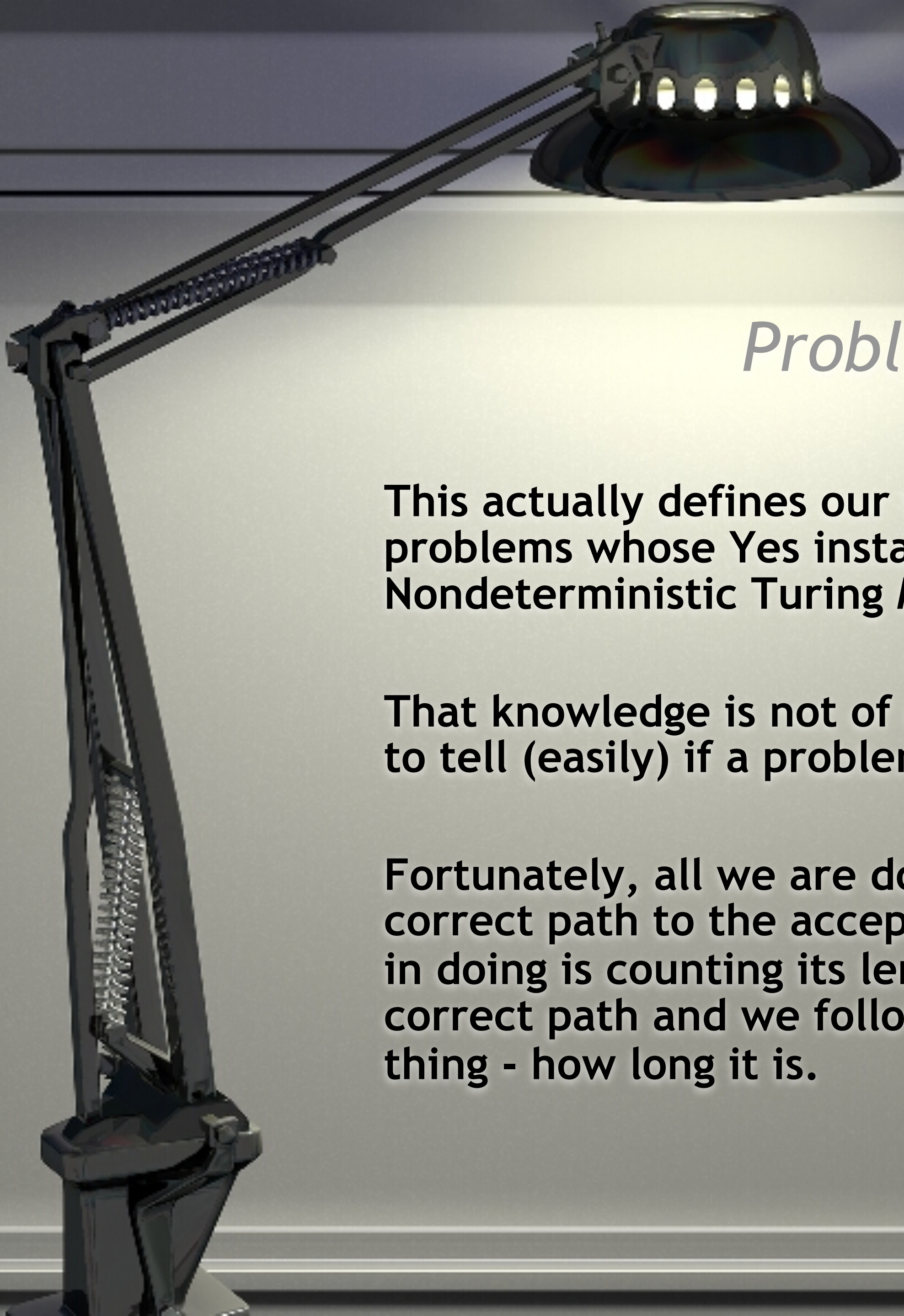
Problem Classes

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.

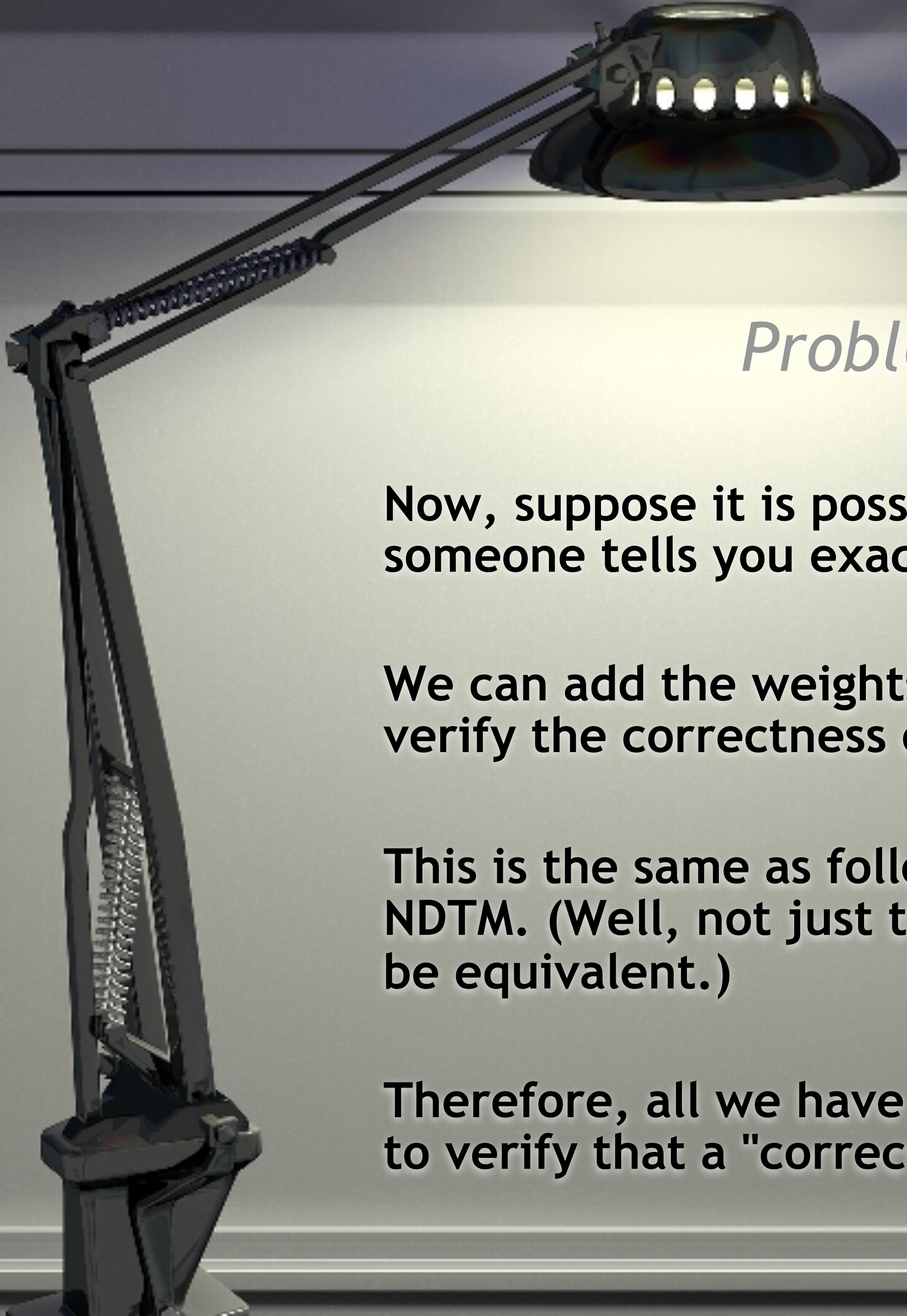
Fortunately, all we are doing with a NDTM is tracing the correct path to the accept state. Since all we are interested in doing is counting its length, if someone just gave us the correct path and we followed it, we could learn the same thing - how long it is.

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.

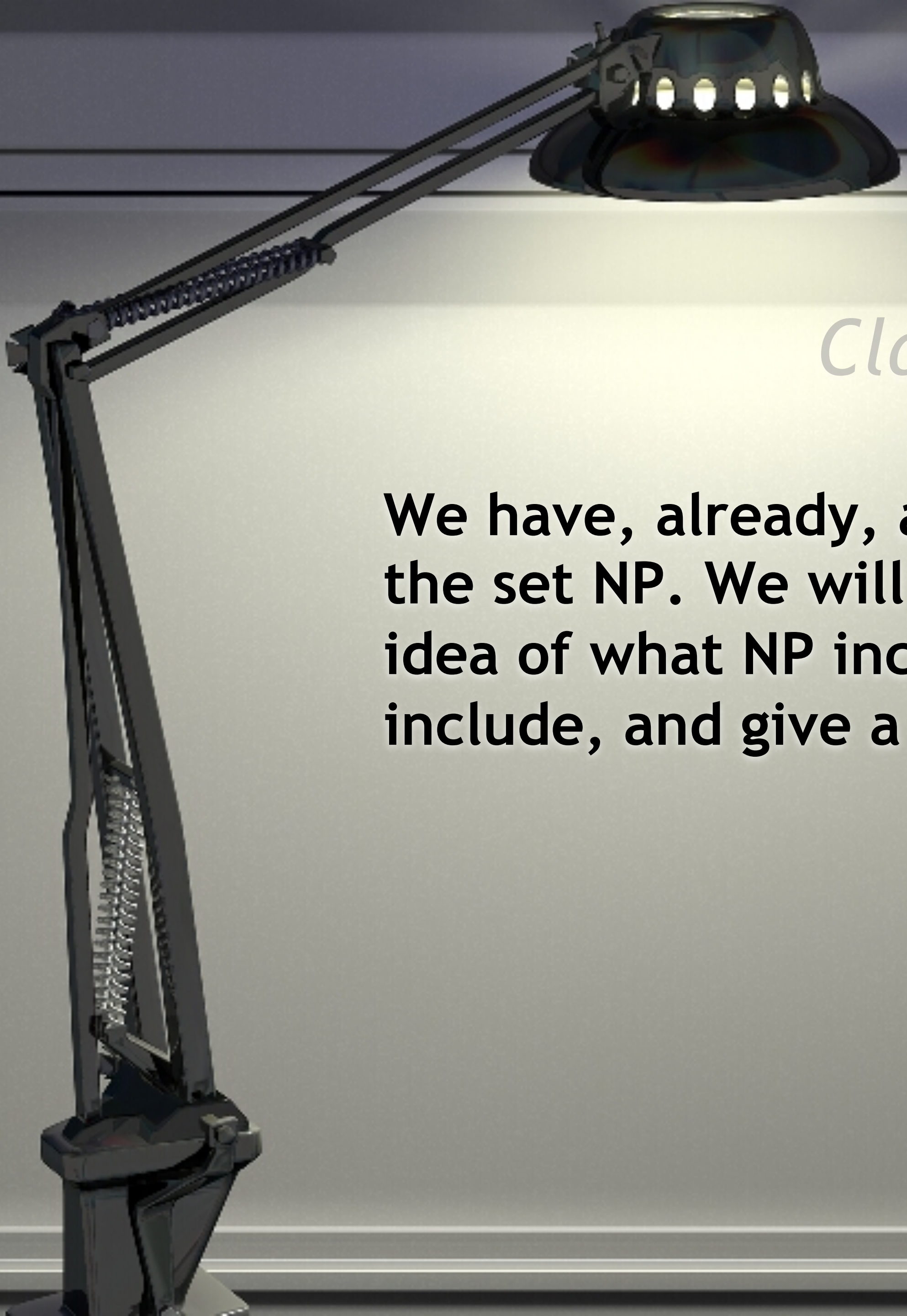
This is the same as following the correct path in a NDTM. (Well, not just the same, but it can be proven to be equivalent.)

Therefore, all we have to do is count how long it takes to verify that a "correct" answer" is in fact correct.

Class – NP

We are now ready for our

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

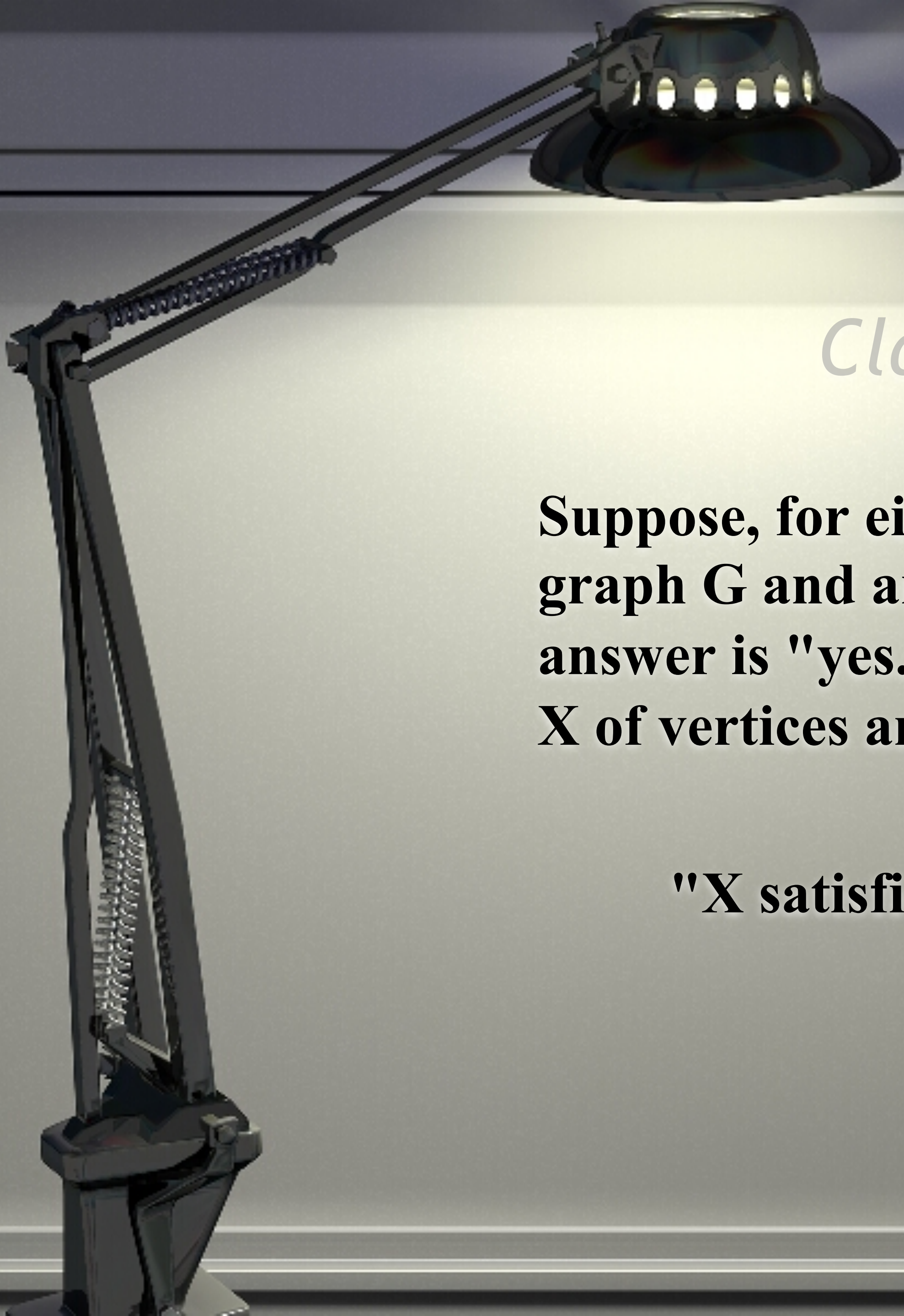
Consider two seemingly closely related statements (versions) of a single problem. We show they are actually very different. Let G = (V, E) be a graph.

Definition: X ⊆ 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

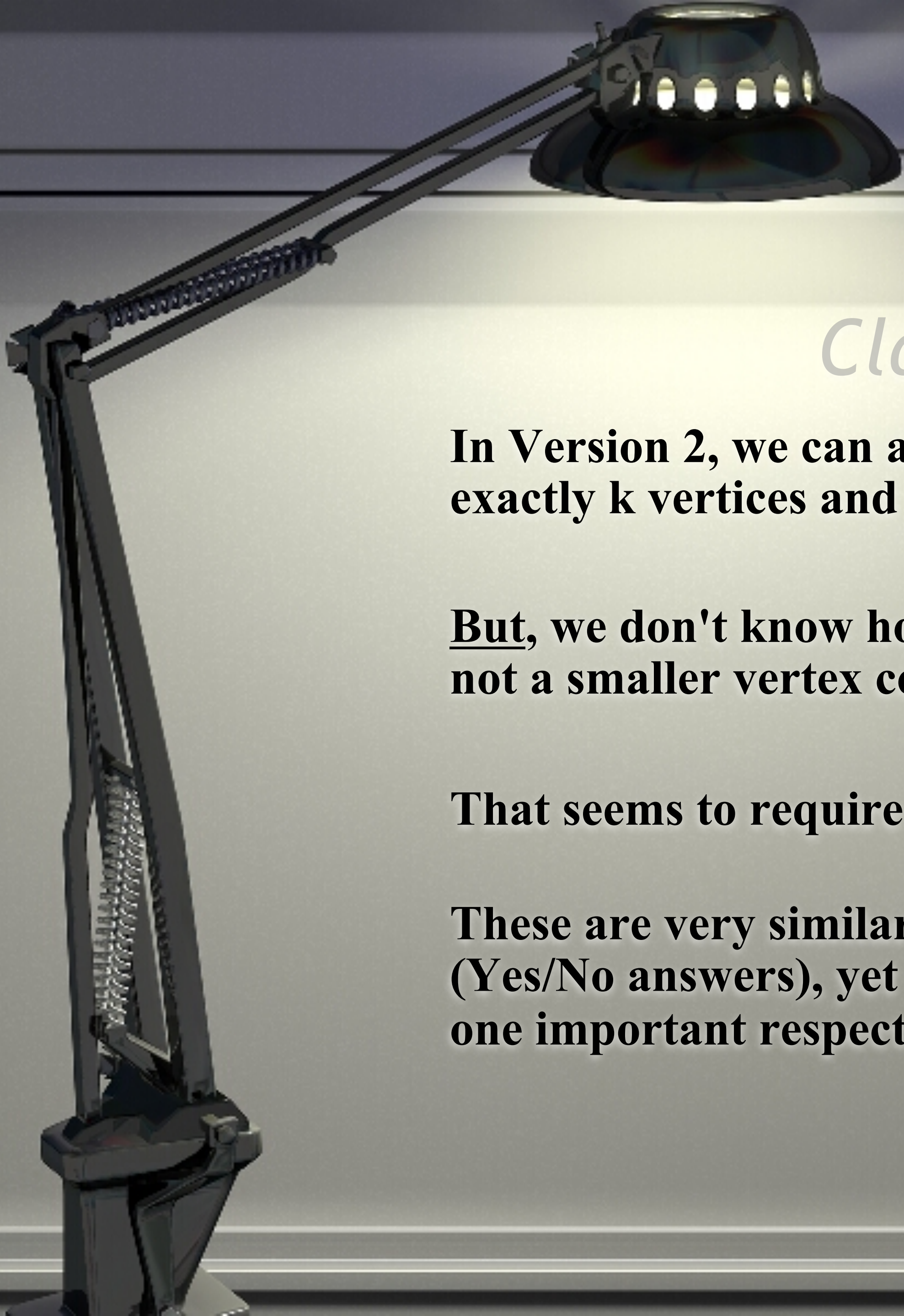
Suppose, for either version, we are given a graph G and an integer k for which the answer is "yes." Someone also gives us a set X of vertices and claims

"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.

But, 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.

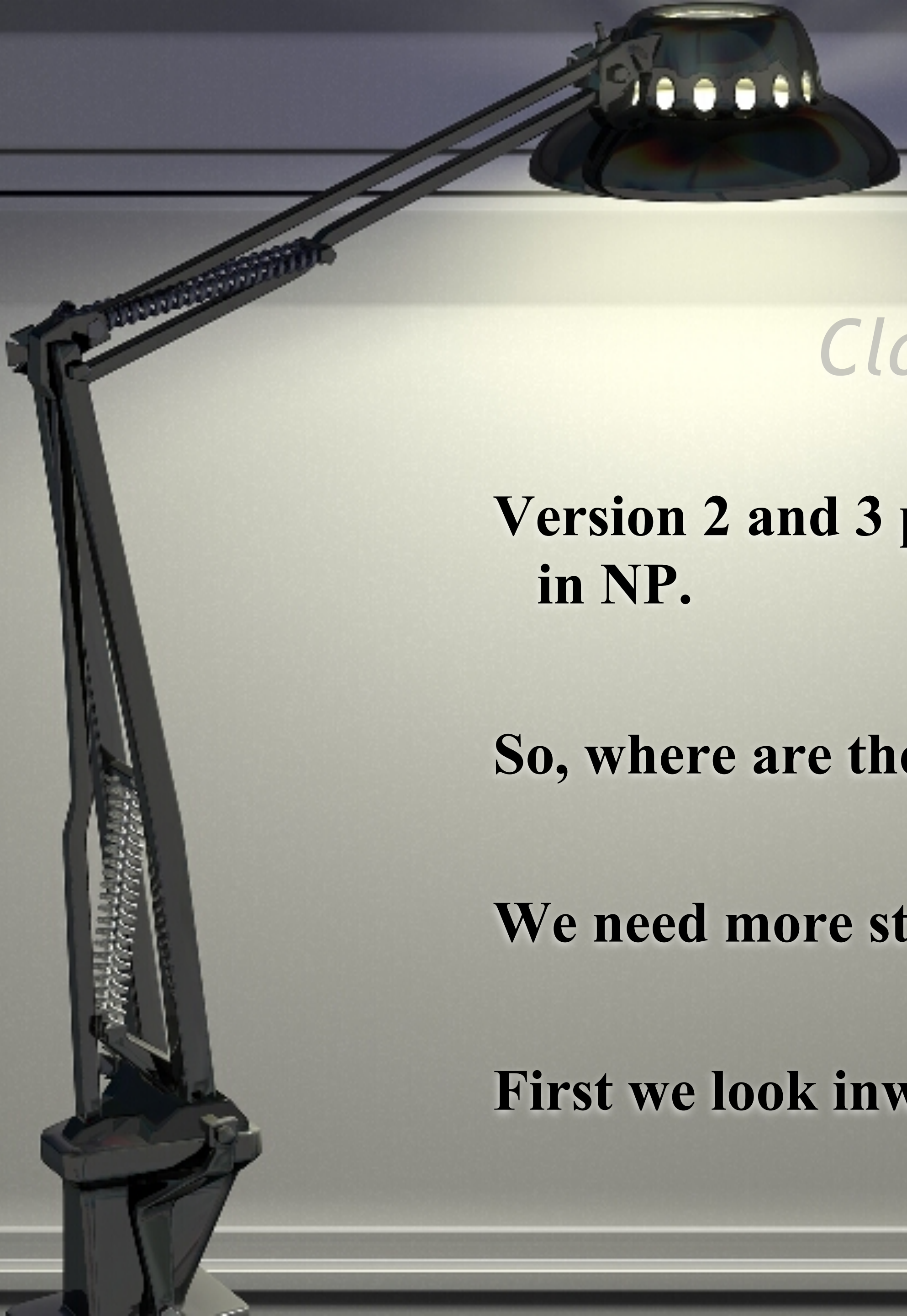
Further, this is not isolated to the Vertex Cover problem. Every decision problem has a 'Version 3,' also known as the 'complement' problem (we will discuss these further at a later point).

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 compute the correct answer in polynomial time - from "scratch."

These are the problems that make up the class P.

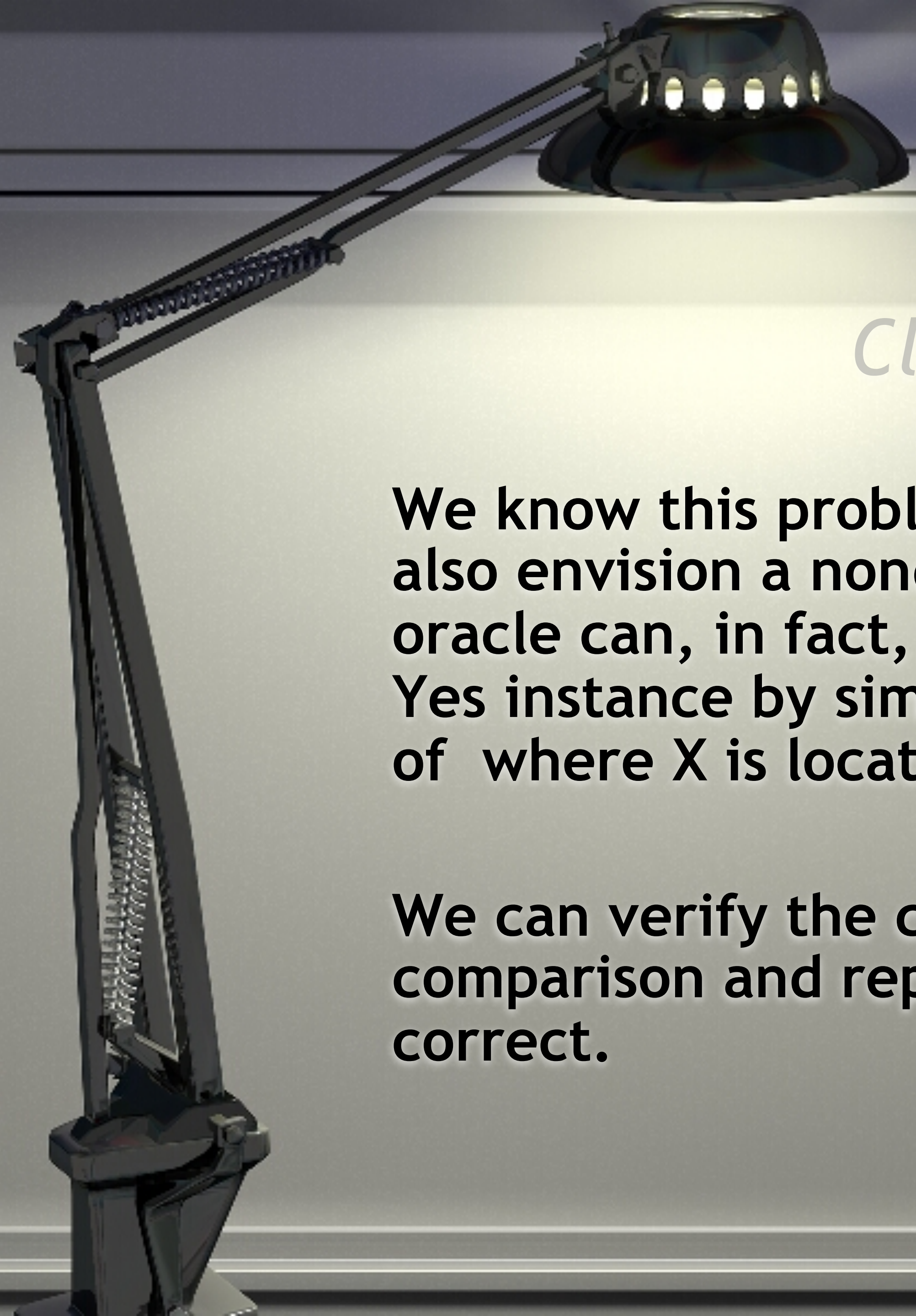
P is a subset of NP.

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 not 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,

$$f_A(n) \leq n^{\alpha} f_B(n),$$

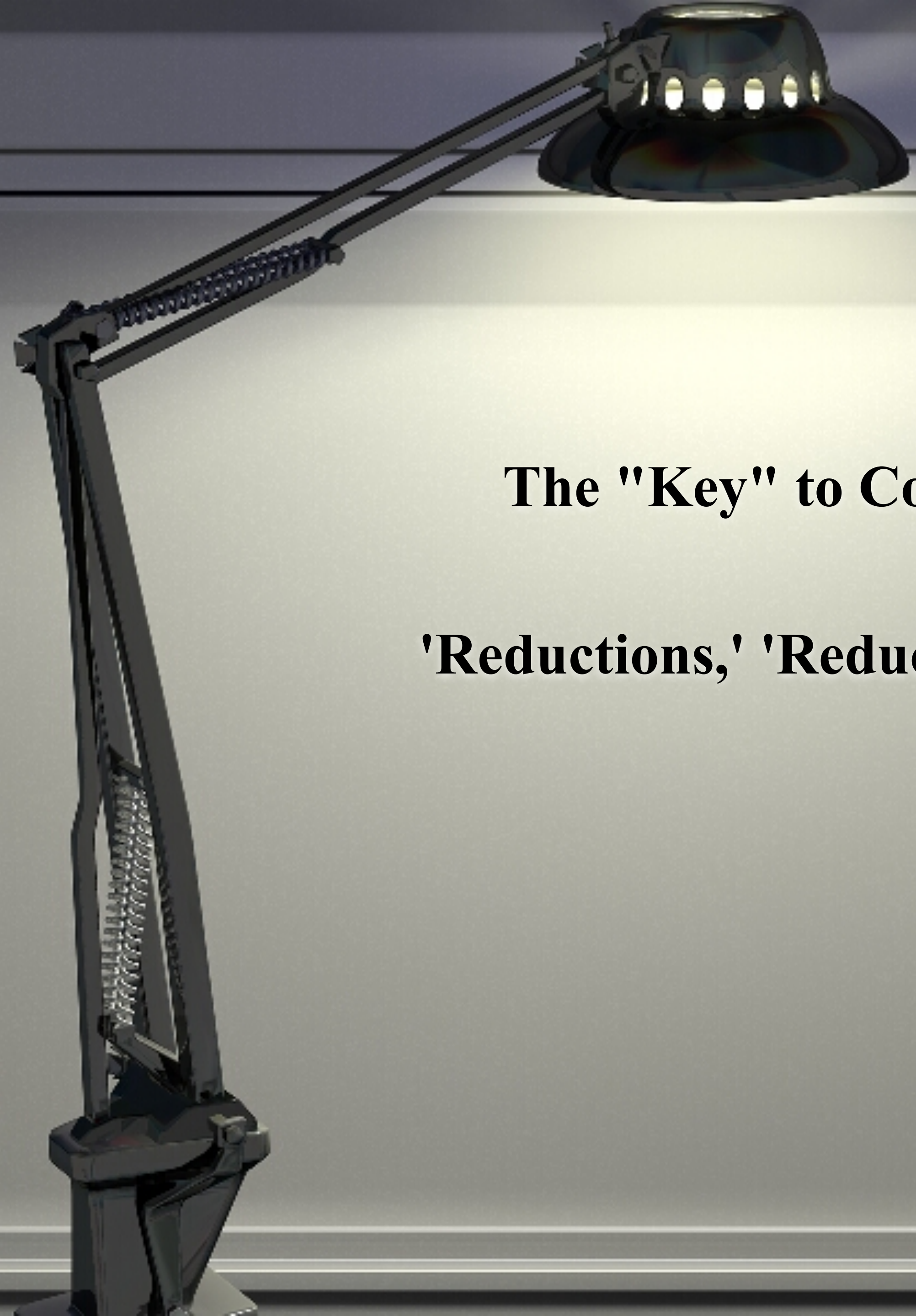
the time to solve A is no more than some polynomial multiple of the time required to solve B, i.e., A is 'no harder than' B.

"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 arbitray 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 assume 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.

Reductions

The existence of a Turing reduction is often stated as:

"Problem A *reduces* to problem B" or, simply,

"A ➤ B"

(Note: G & J use a symbol that I don't have.).

Reductions

Theorem. **If A ➤ B and problem B is polynomial, then problem A is polynomial.**

Corollary. **If A ➤ B and problem A is exponential, then problem B is exponential.**

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.

Reductions

Theorem. If A ➤ 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, $f_A(n) \leq t_A(n)$. Further, since we assume the best algorithm for B, $t_B(n) = f_B(n)$. Since A ➤ B, there is a constant k such that $t_A(n) \leq n^k t_B(n)$. Therefore, $f_A(n) \leq t_A(n) \leq n^k t_B(n) = n^k f_B(n)$. That is, A is no harder than B.

Reductions

Theorem.

If A ➤ B and B ➤ C then A ➤ C.

Definition.

If A ➤ B and B ➤ A, then A and B are *polynomially equivalent*.

Reductions

A ➤ B means:

'Problem A is *no harder than* problem B,' and

'Problem B is *as hard as* problem A.'

An Aside (Computability Theory)

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.

Special Type of Reduction

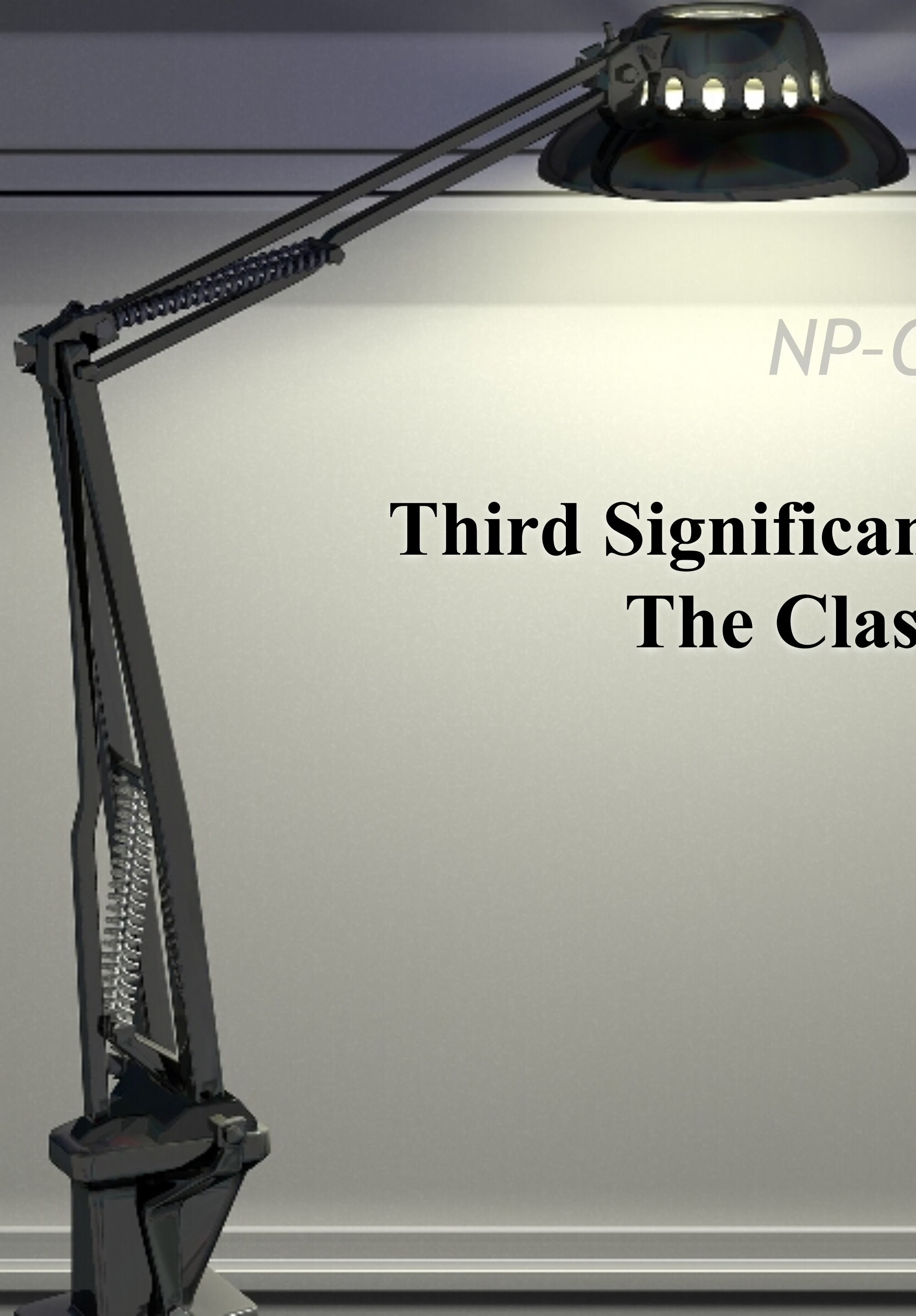
Polynomial Transformation

(Refer to the definition of Turing Reductions)

(1) Problems A and B must both be decision problems.

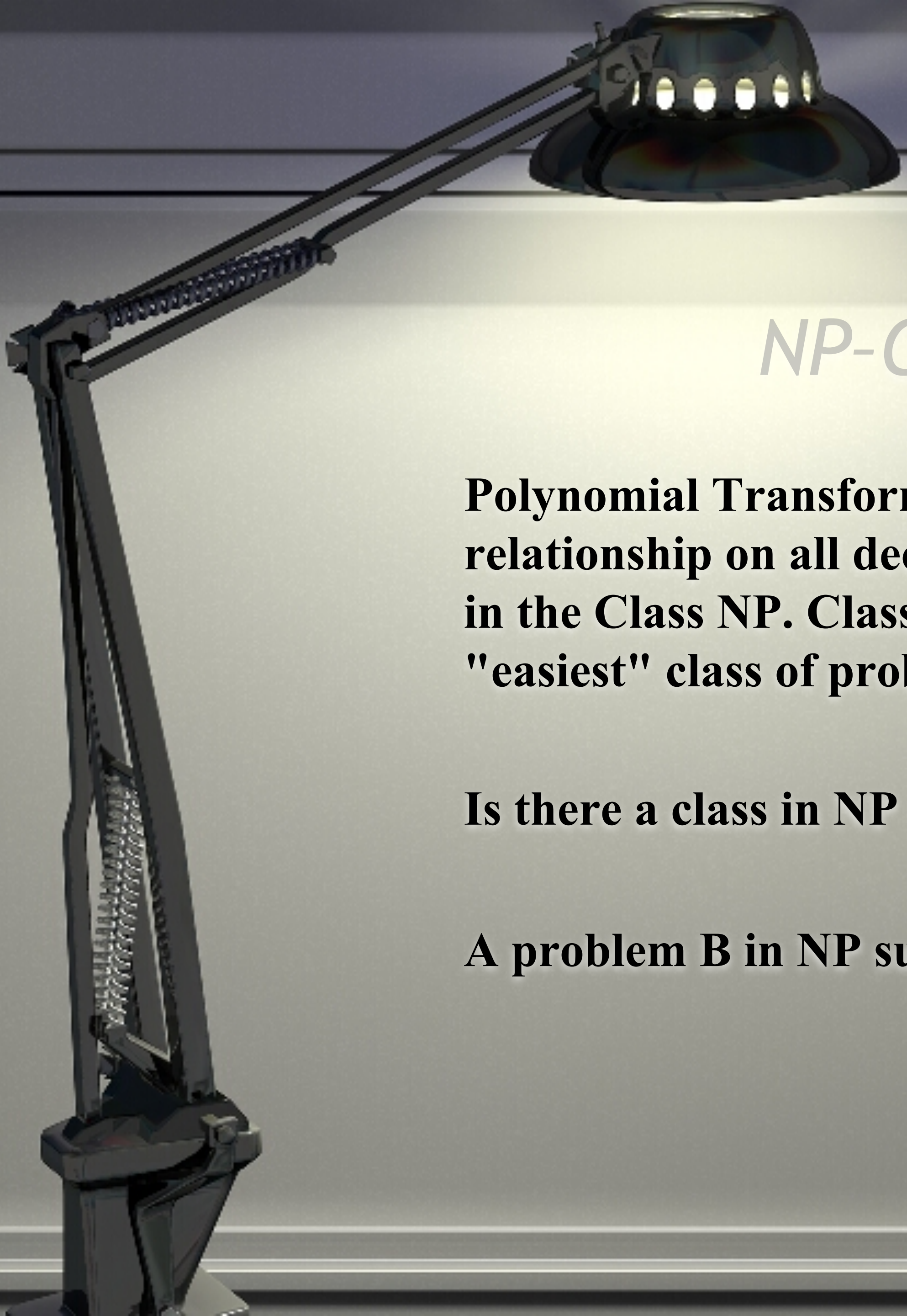
(2) A single instance, I_B, of problem B is constructed from a single instance, I_A, of problem A.

(3) I_B is true for problem B if and only if I_A is true for problem A.



NP-Complete

Third Significant Class of Problems: The Class NP–Complete

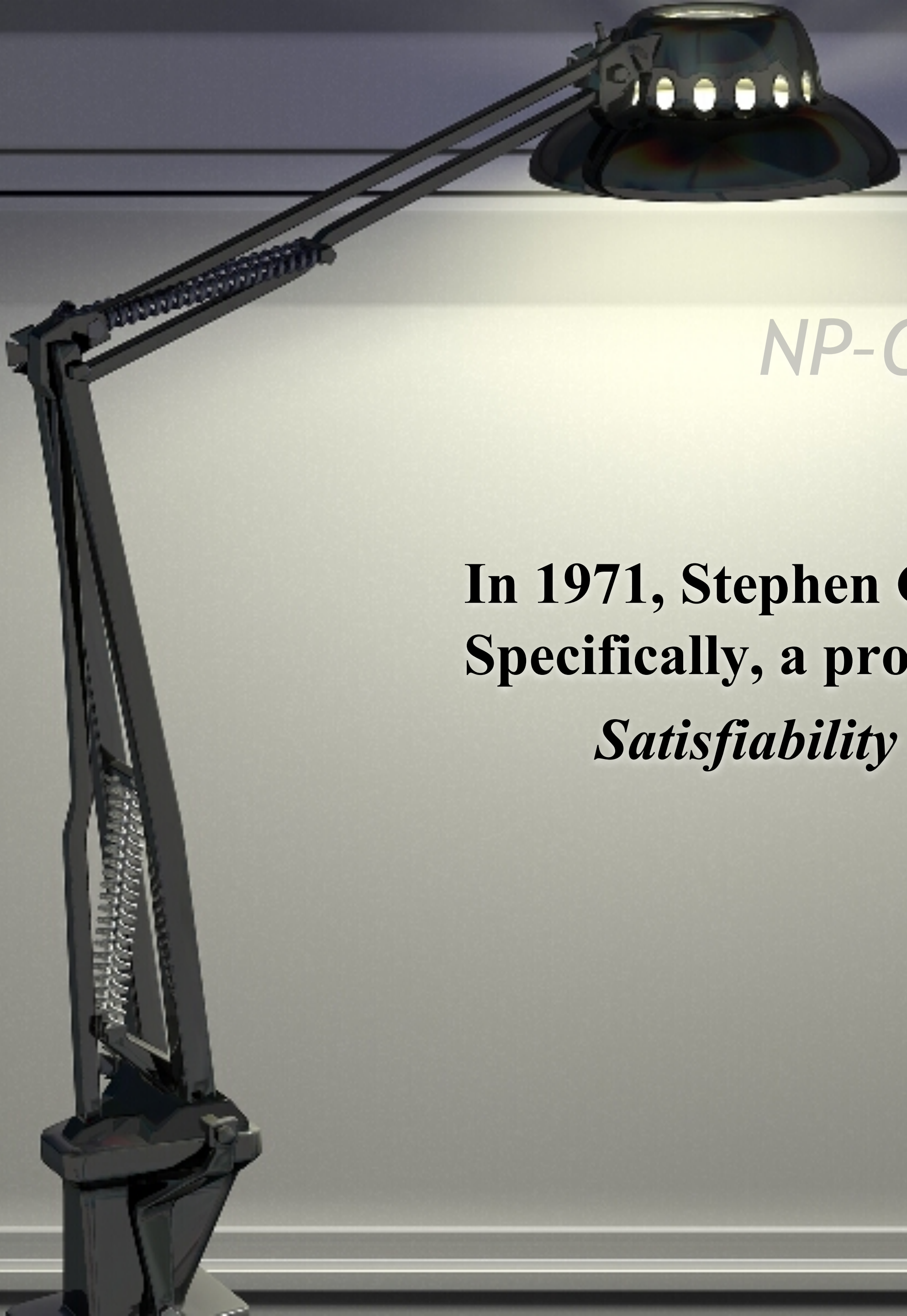


NP-Complete

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 ➤$_P$ 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 algorithm!!

NP-Complete

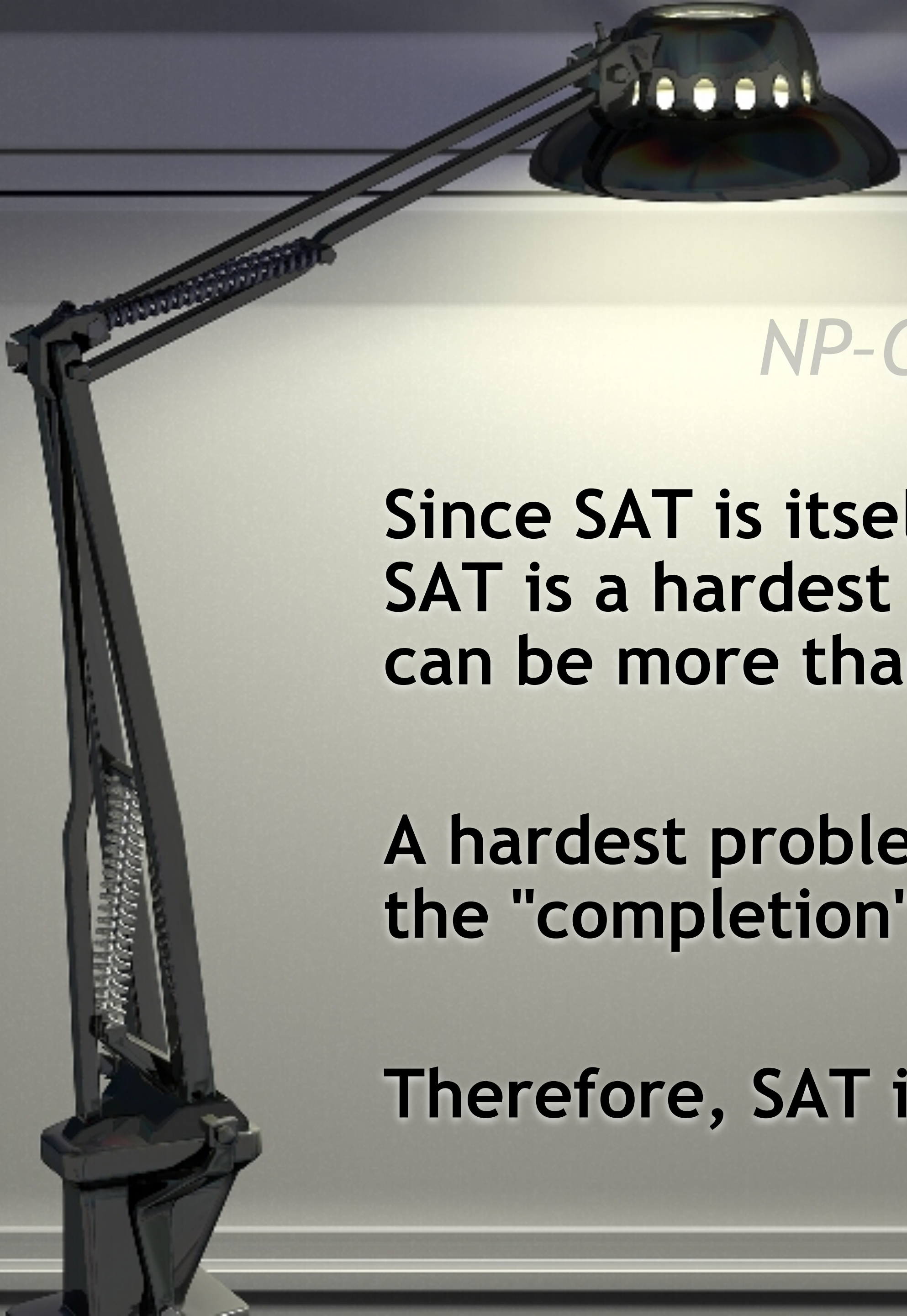
Cooks Theorem:

1) SAT is in NP
2) For every problem A in NP,

$$\text{A} \blacktriangleright_{\text{P}} \text{SAT}$$

Thus, SAT is as hard as every problem in NP.

(For a proof, see Garey and Johnson, pgs. 39 – 44)

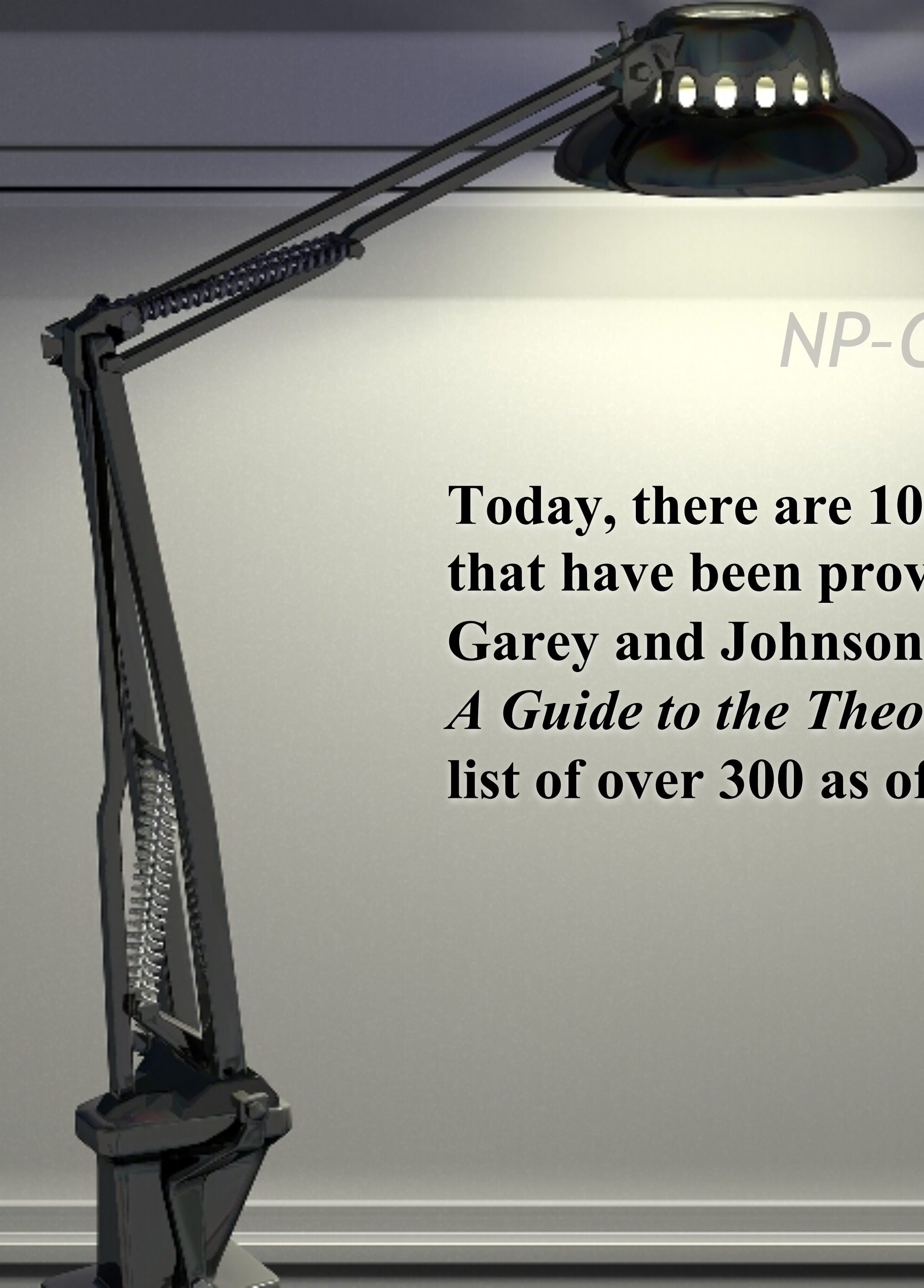


NP-Complete

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 100's, if not 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.

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.
- An awfully lot of 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.

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 algorithm!!

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 can show is that SAT is NP-complete, providing us our first concrete example of an NP-complete decision problem.

Simulating ND TM

- Given a TM, M, and an input w, we need to create a formula, $\varphi_{M,w}$, 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**

Cook’s Theorem

- $\varphi_{M,w} = \phi cell \wedge \phi start \wedge \phi move \wedge \phi accept$
- See the following for a 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.

These included such problems as:

3-SAT

3DM

Vertex Cover,

Independent Set,

Knapsack,

Multiprocessor Scheduling, and

Partition.

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

The following polynomial transformations have been shown to exist.(Later, we will see what these problems actually are.)

Theorem. SAT ➤$_{P}$ 3SAT

Theorem. 3SAT ➤$_{P}$ SubsetSum

Theorem. SubsetSum ➤$_{P}$ Partition

Example SubsetSum

Assuming a **3SAT expression (a + c + c) (b + b + ~c), the following shows the reduction from 3SAT to Subset-Sum.**

	a	**b**	**c**	**a + ~b + c**	**~a + b + ~c**
a	**1**			**1**	
~a	**1**				**1**
b		**1**			**1**
~b		**1**		**1**	
c			**1**	**1**	
~c			**1**		**1**
C1				**1**	
C1'				**1**	
C2					**1**
C2'					**1**
	1	**1**	**1**	**3**	**3**

Partition

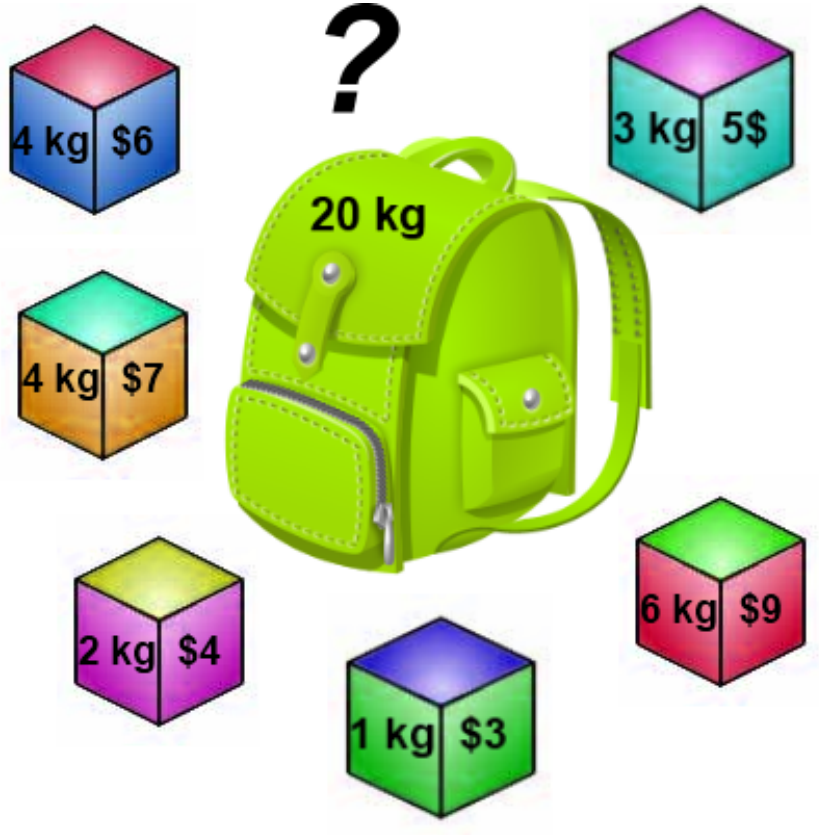
- Let i_1, i_2, .., i_n be a multi-set of positive natural numbers. Partition asks if this multi-set can be partitioned into two sub multi-sets whose union is the original multi-set and whose sums are equal.
- Partition is related to the optimization problem of dividing the multi-set into two partitions whose sums are as close to each other as possible.

Partition

- 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

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 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 with weight < W that reaches some fixed goal value, G.

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 is 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

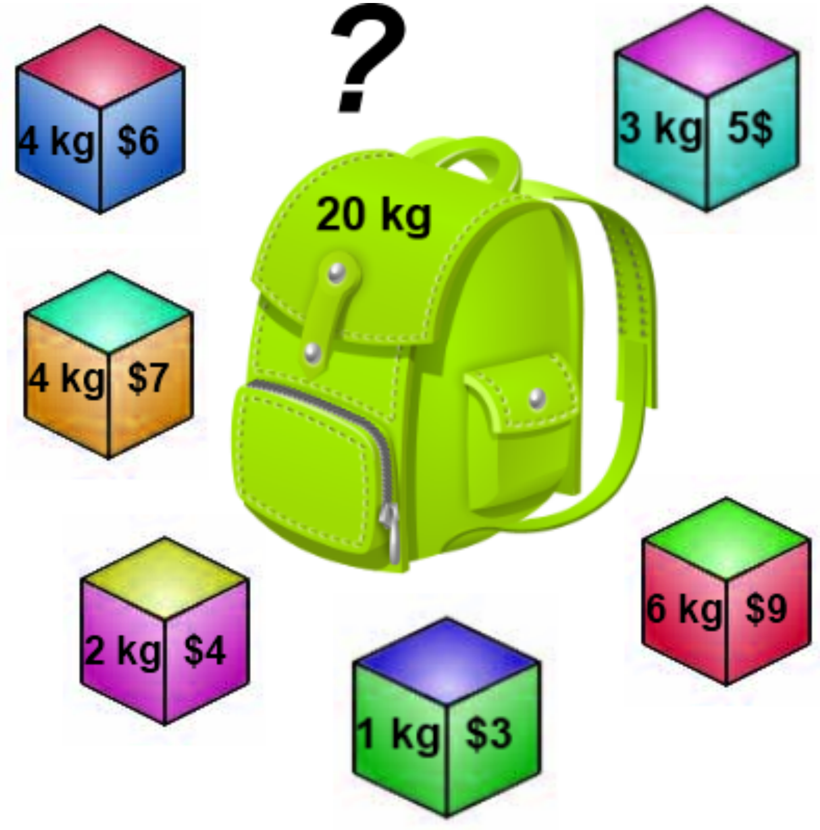
- The reduction from SubsetSum shows that the Knapsack decision problem is at least as hard as SubsetSum, so it is NP-Complete if it is in NP.
- Think about whether or not it is in NP.

Related Bin Packing

- Have a bin capacity of B.
- Have item set **$S = \{s_1, s_2, \ldots, s_n\}$**
- 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.

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 0-1 Problem

- Brute Force
  - The naïve way to solve this 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 USUALLY does better 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

- **The maximum weight the knapsack can hold is 20.**

- 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:

$$
\begin{aligned}
\mathbf{B[k, w]} &= \mathbf{B[k-1, w]}, \ \underline{\textbf{if } w_k > w} \\
&= \max \{ \mathbf{B[k-1, w]}, \mathbf{B[k-1, w-w_k]} + v_k \}, \ \underline{\textbf{otherwise}}
\end{aligned}
$$

- 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

$$B[k,w] = \begin{cases} B[k-1,w] & \text{if } w_k > w \\ \max\{B[k-1,w], B[k-1,w-w_k]+b_k\} & \text{else} \end{cases}$$

- The best subset of S_k that has the total weight w, either contains item k or not.

- **First case:** $w_k > w$
  - Item *k* can't be part of the solution! If it was the total weight would be > w, which is unacceptable.

- **Second case:** $w_k \le w$
  - Then the item *k* can 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 $w_i$ <= 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 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					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 2}$

$w-w_i = 0$

```
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					

$i = 1$

$v_i = 3$

$w_i = 2$

$\mathbf{w = 4}$

$w - w_i = 2$

```
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					
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					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 2}$

$w-w_i = -1$

```
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					

$i = 2$
$v_i = 4$
$w_i = 3$
$\mathbf{w = 3}$
$w\text{-}w_i = 0$

```
if $w_i \leq w$  //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					

$i = 2$

$v_i = 4$

$w_i = 3$

$\mathbf{w = 4}$

$w\text{-}w_i = 1$

```
if  $w_i <= w$   //item i can be in the solution
        if $v_i + B[i-1,w-w_i] > B[i-1,w]$
            $\mathbf{B[i,w] = v_i + B[i-1,w- w_i]}$
        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 $v_i + B[i-1,w-w_i] > B[i-1,w]$
            $B[i,w] = v_i + B[i-1,w- w_i]$
        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					

$i = 3$

$v_i = 5$

$w_i = 4$

$\mathbf{w = 1..3}$

$w\text{-}w_i = \text{-}3..\text{-}1$

```
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 $\mathbf{B[i,w] = B[i-1,w]}$ // $w_i > w$
```

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 $v_i + B[i-1,w-w_i] > B[i-1,w]$
                $B[i,w] = v_i + B[i-1,w- w_i]$
        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 $w_i \le w$   //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	3	3
2	0	0	3	4	4	7
3	0	0	3	4	5	7
4	0	0	3	4	5	

$i = 4$

$v_i = 6$

$w_i = 5$

$\mathbf{w = 1..4}$

$w\text{-}w_i = -4..-1$

```
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)
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

$i = 4$
$v_i = 6$
$w_i = 5$
$\mathbf{w = 5}$
$w - w_i = 0$

```
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	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)**

Polynomial Transformations

Polynomial transformations are also known as *Karp Reductions*

When a reduction is a polynomial transformation, we subscript the symbol with a "p" as follows:

$$\mathbf{A} \blacktriangleright_{\mathbf{P}} \mathbf{B}$$

Polynomial Transformations

Following Garey and Johnson, we recognize three forms of polynomial transformations.

(a) restriction,

(b) local replacement, and

(c) component design.

Polynomial Transformations

Restriction **allows nothing much more complex than renaming the objects in I_A so that they are, in a straightforward manner, objects in I_B.**

For example, objects in I_A could be a collection of cities with distances between certain pairs of cities. In I_B, these might correspond to vertices in a graph and weighted edges.

Polynomial Transformations

The term 'restriction' alludes to the fact that a proof of correctness often is simply describing the subset of instances of problem B that are essentially identical (isomorphic) to the instances of problem A, that is, the instances of B are restricted to those that are instances of A. To apply restriction, the relevant instances in Problem B must be identifiable in polynomial time.

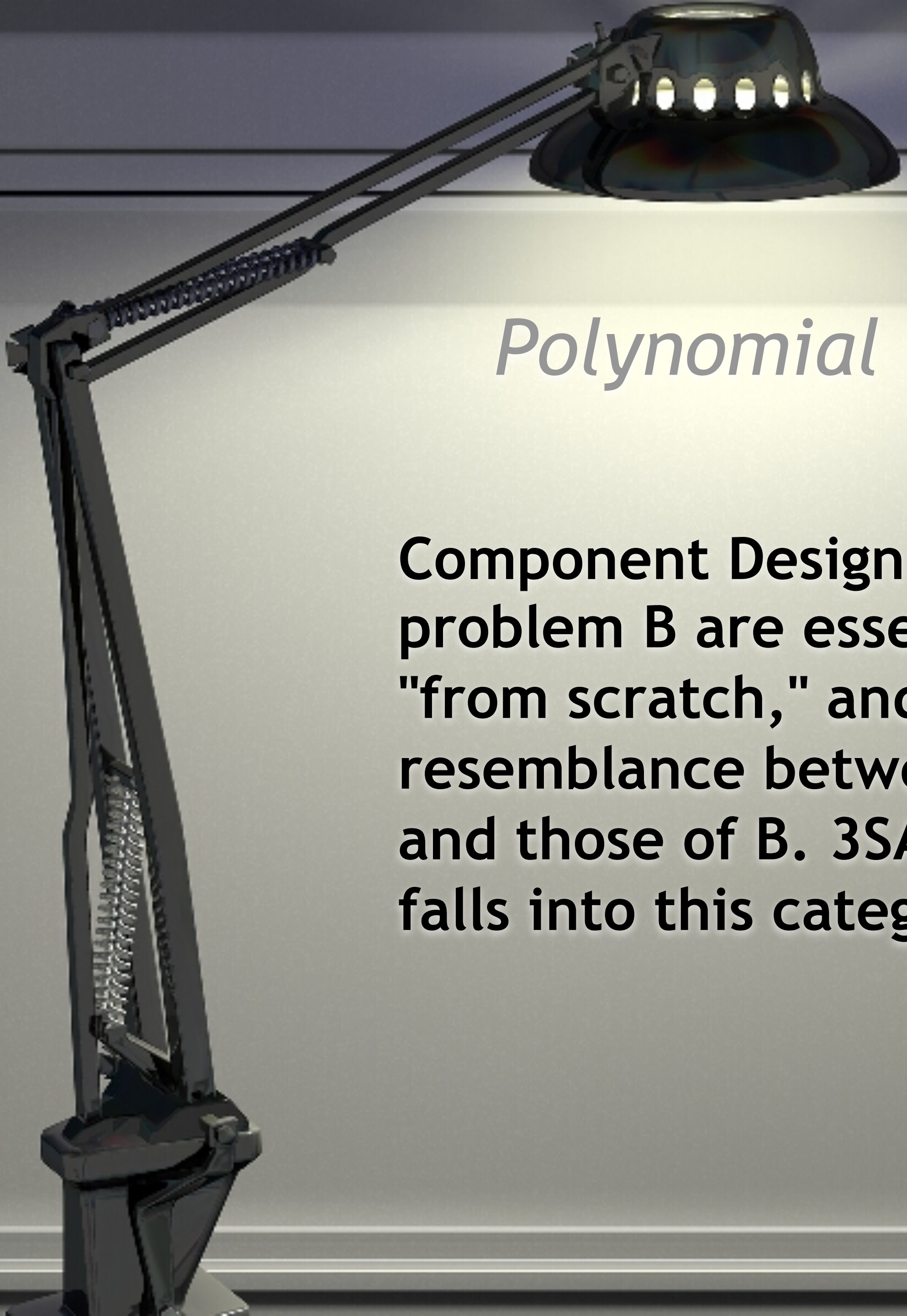
For example, if P ≠ NP and B is defined over the set of all graphs, we can not restrict to the instances that possess a Hamiltonian Circuit.

Polynomial Transformations

Local Replacement is more complex because there is usually not an obvious map between instance I_A and instance I_B. But, by modifying objects or small groups of objects a transformation often results. Sometimes the alterations are such that some feature or property of problem A that is not a part of all instances of problem B can be enforced in problem B. As in (a), the instances of problem B are usually of the same type as of problem A.

Polynomial Transformations

In a sense, Local Replacement might be viewed as a form of Restriction. In Local Replacement, we describe how to *construct* the instances of B that are isomorphic to the instances of A, and in Restriction we describe how to *eliminate* instances of B that are not isomorphic to instances of A.



Polynomial Transformations

Component Design is when instances of problem B are essentially constructed "from scratch," and there may be little resemblance between instances of A and those of B. 3SAT to SubsetSum falls into this category.

3SAT to Vertex Cover

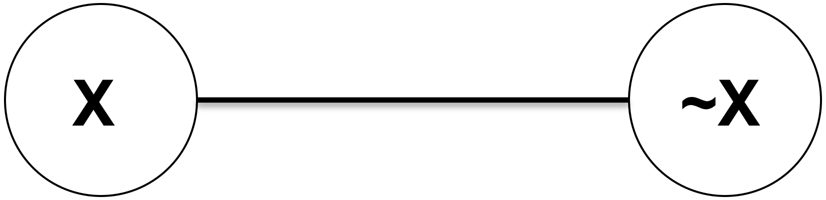
- Vertex cover seeks a set of vertices that cover every edge in some graph
- Let $I_{3\text{-SAT}}$ be an arbitrary instance of 3-SAT. For integers n and m, U = $\{u_1, u_2, \ldots, u_n\}$ and $C_i = [z_{i1}, z_{i2}, z_{i3}\}$ for $1 \le i \le m$, where each z_{ij} is either a u_k or u_k' for some k.

- 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.

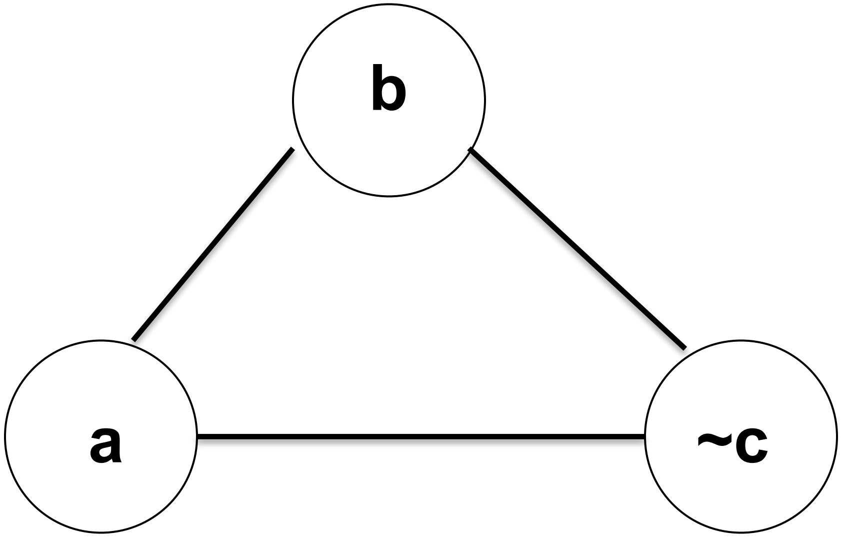
- **Theorem.** The given instance of 3-SAT is satisfiable if and only if the constructed instance of VC has a vertex cover with at most k vertices.

VERTEX COVERING (VC) DECISION PROBLEM IS NP-HARD

VC Variable Gadget



VC Clause Gadget

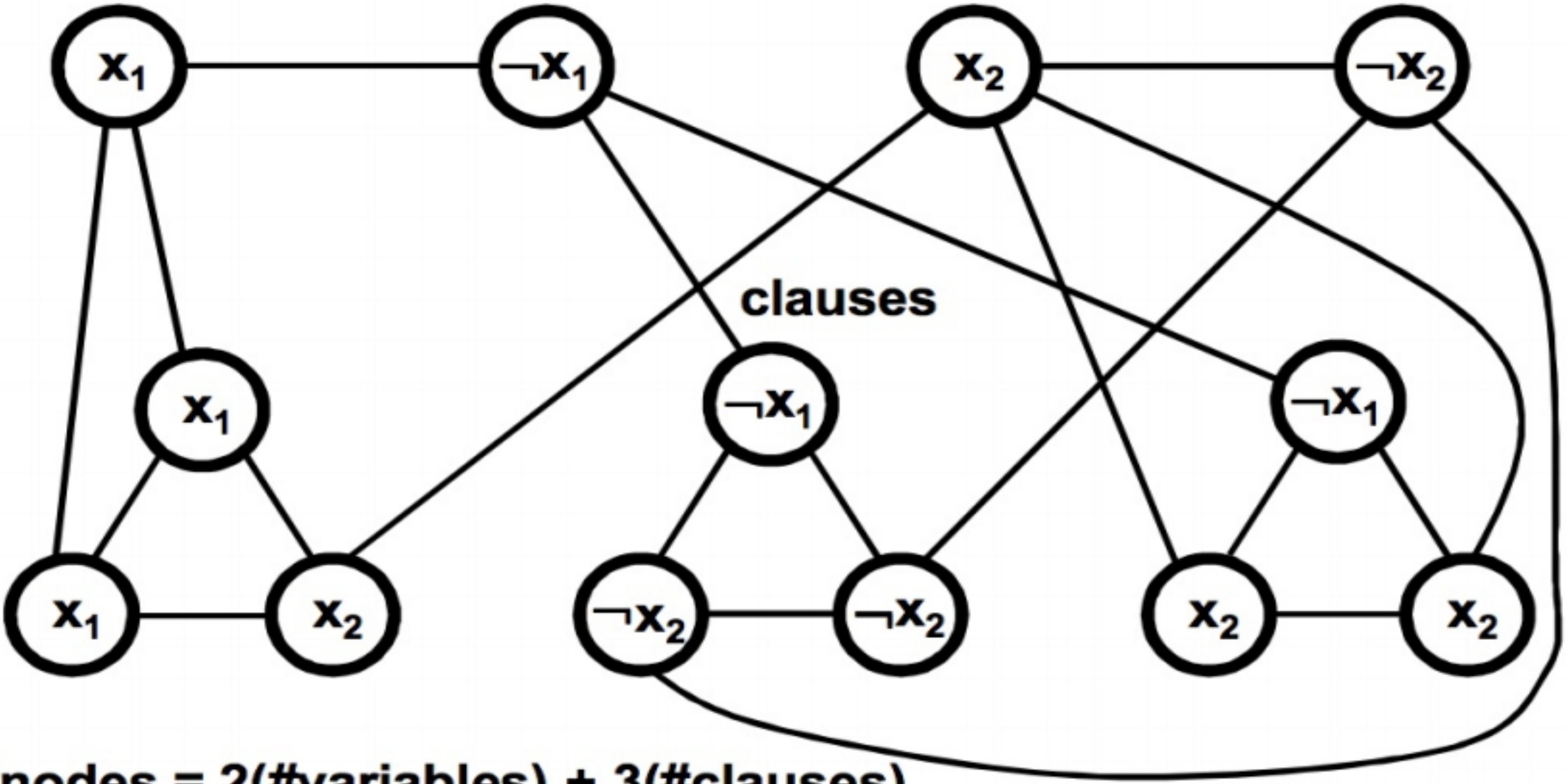


a + b + ~c

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

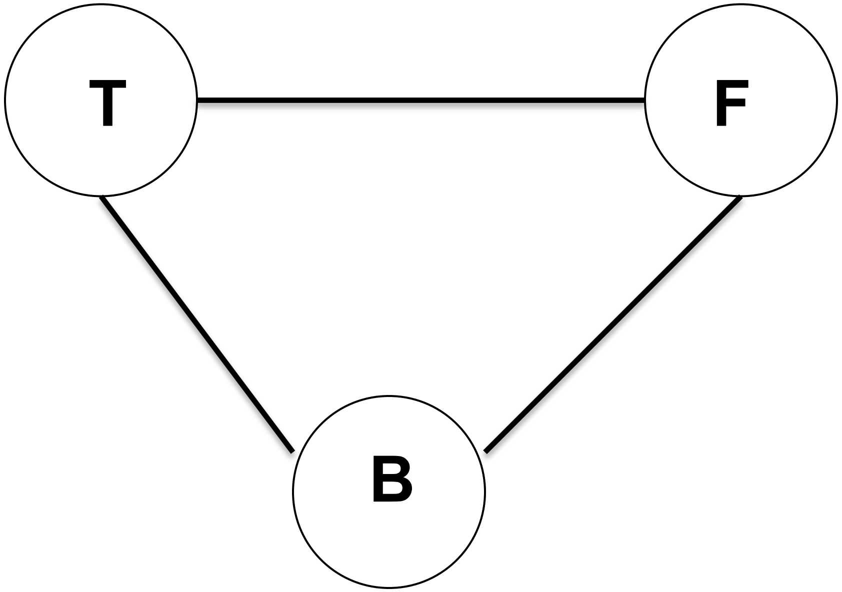


clauses

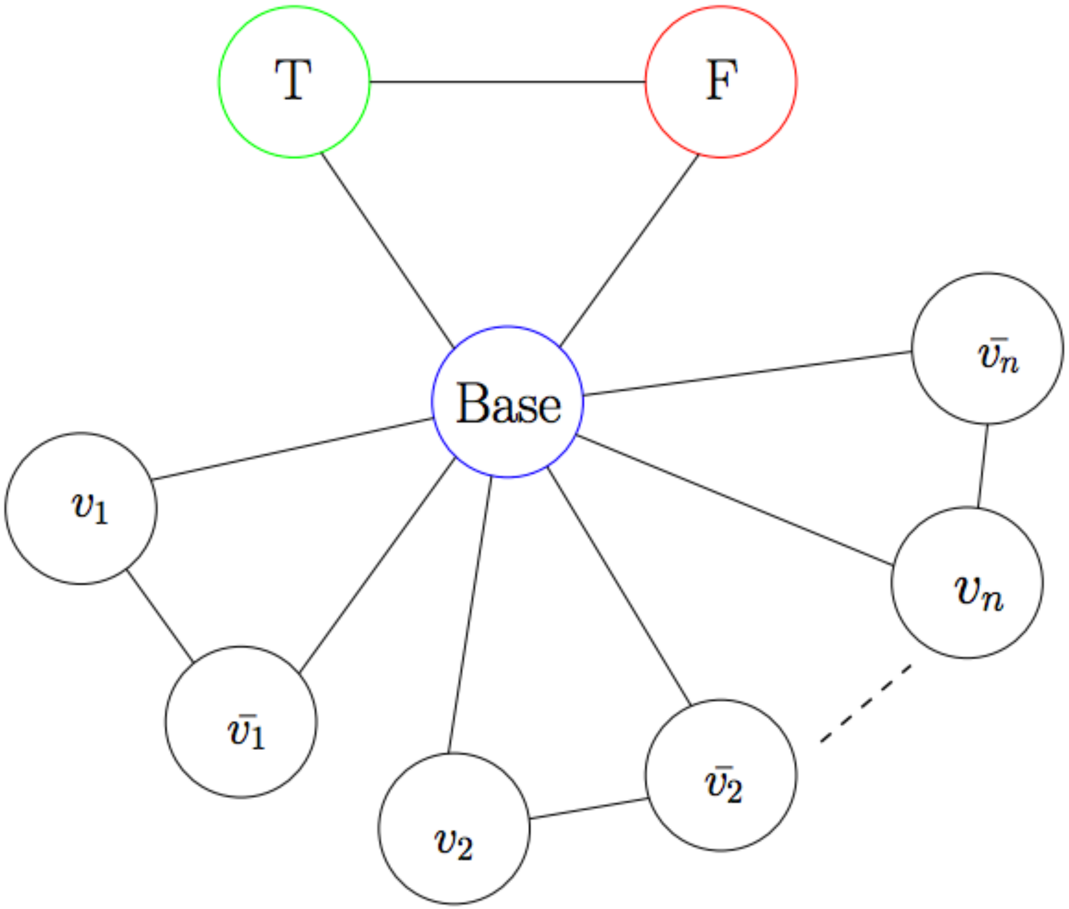
#nodes = 2(#variables) + 3(#clauses)

K-COLOR (KC) DECISION PROBLEM IS NP-HARD

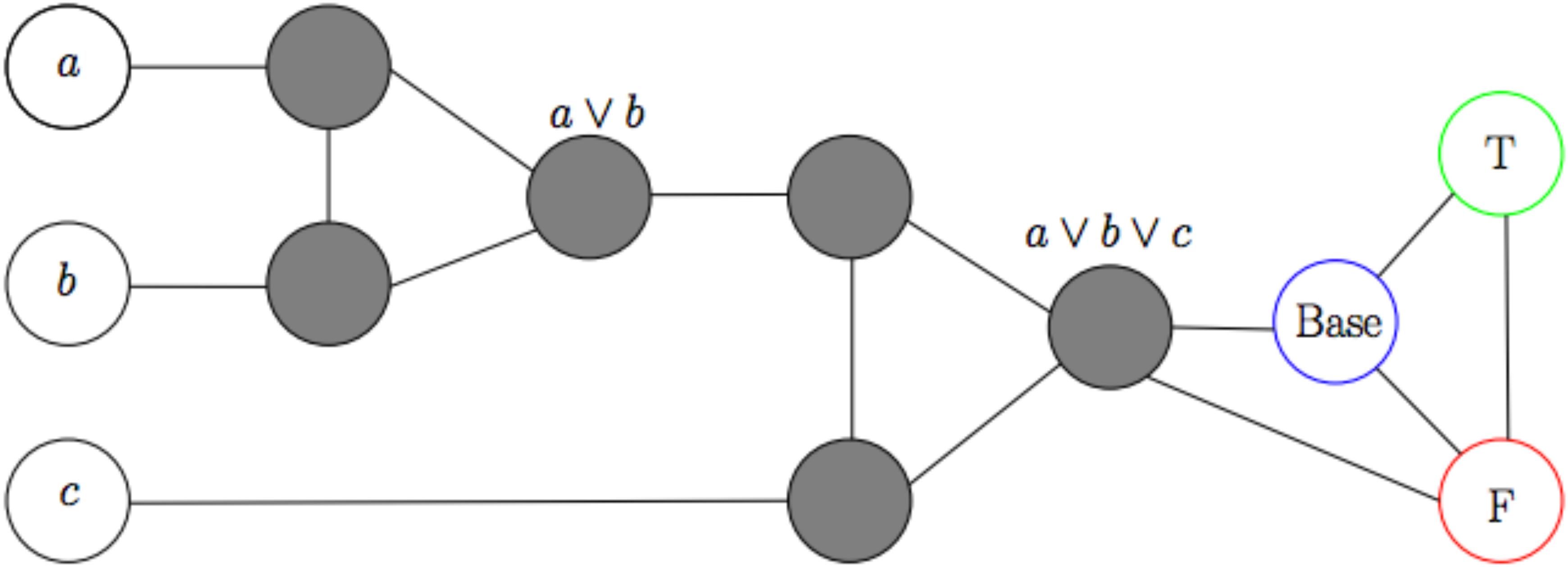
3C Super Gadget



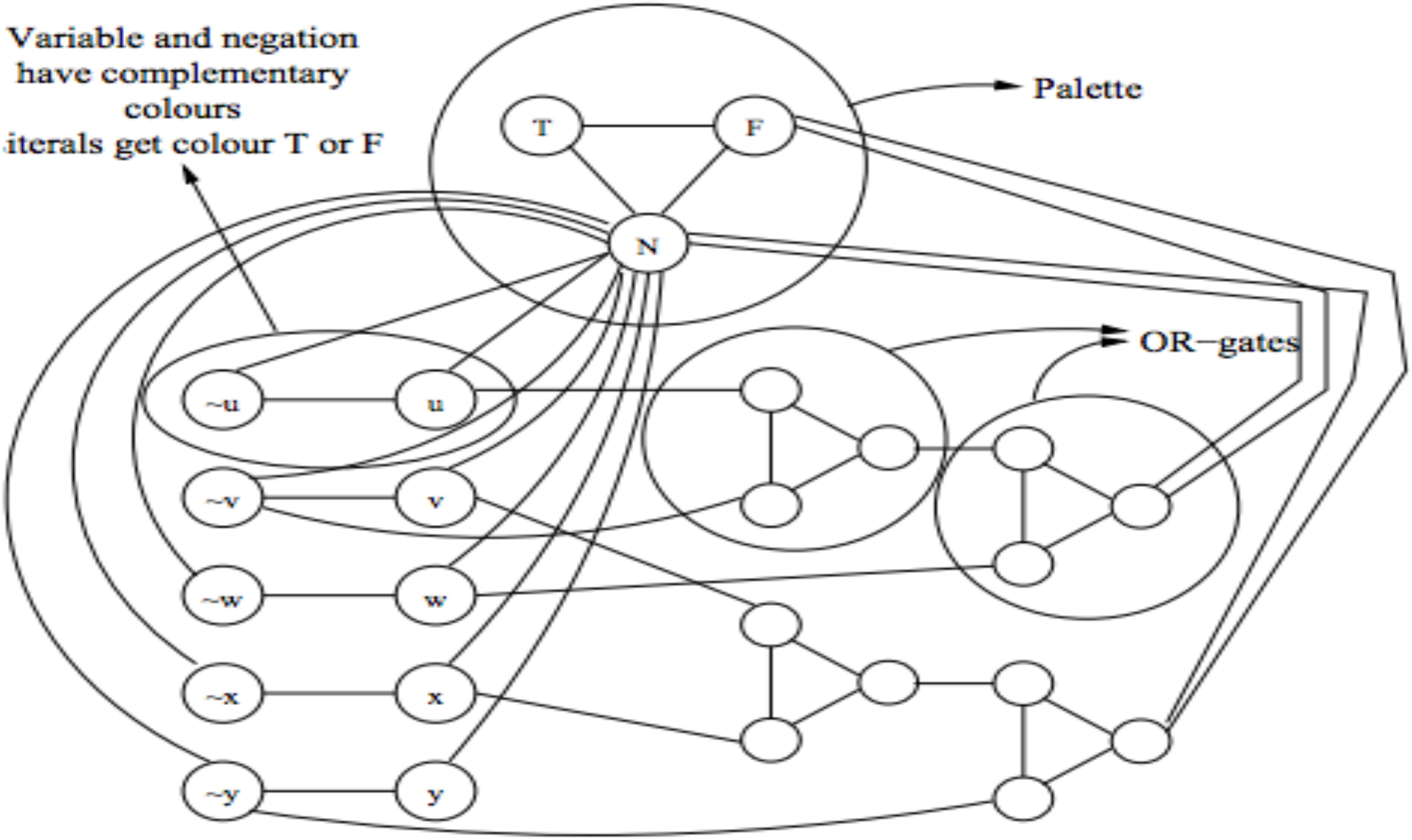
KC Super + Variables Gadget



KC Clause Gadget



KC Gadgets Combined



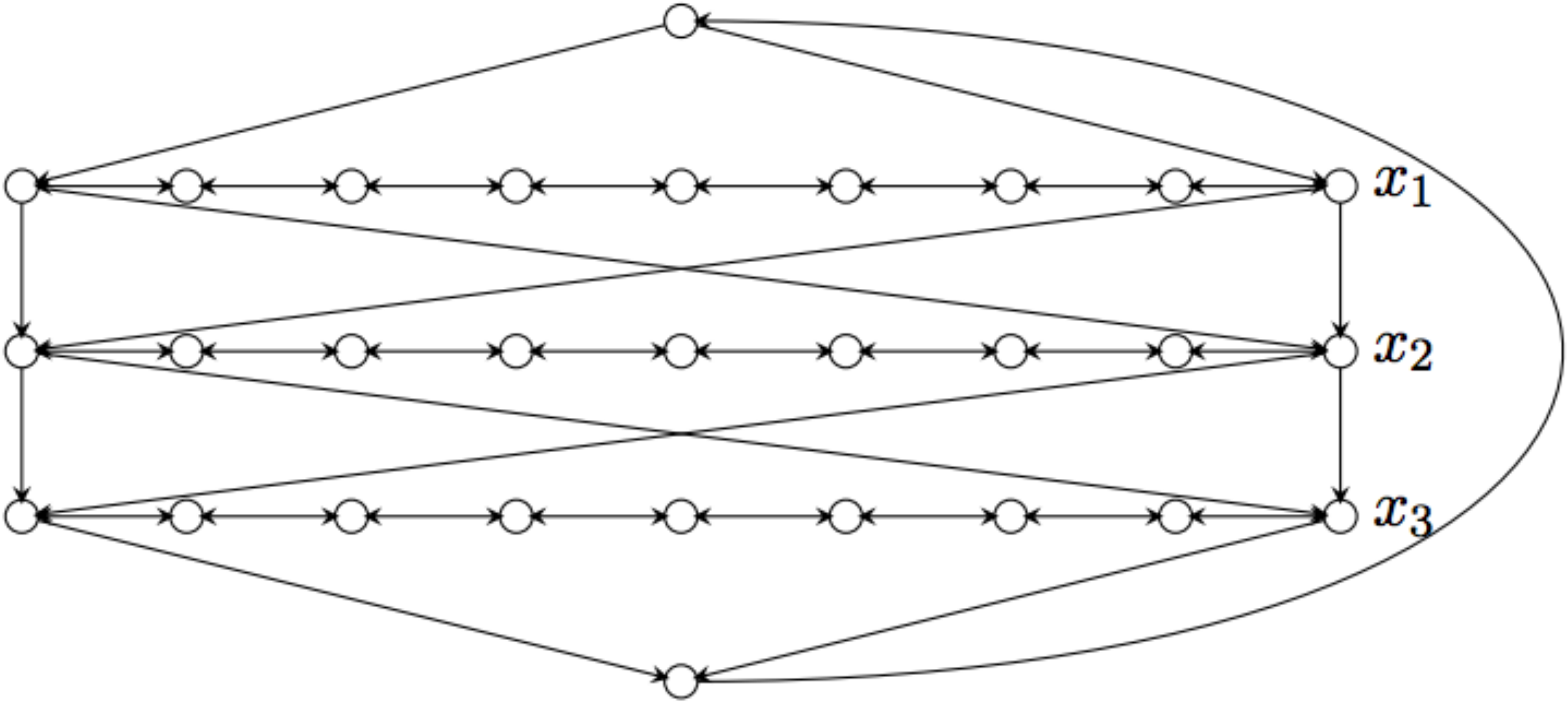
K = 3

Register allocation

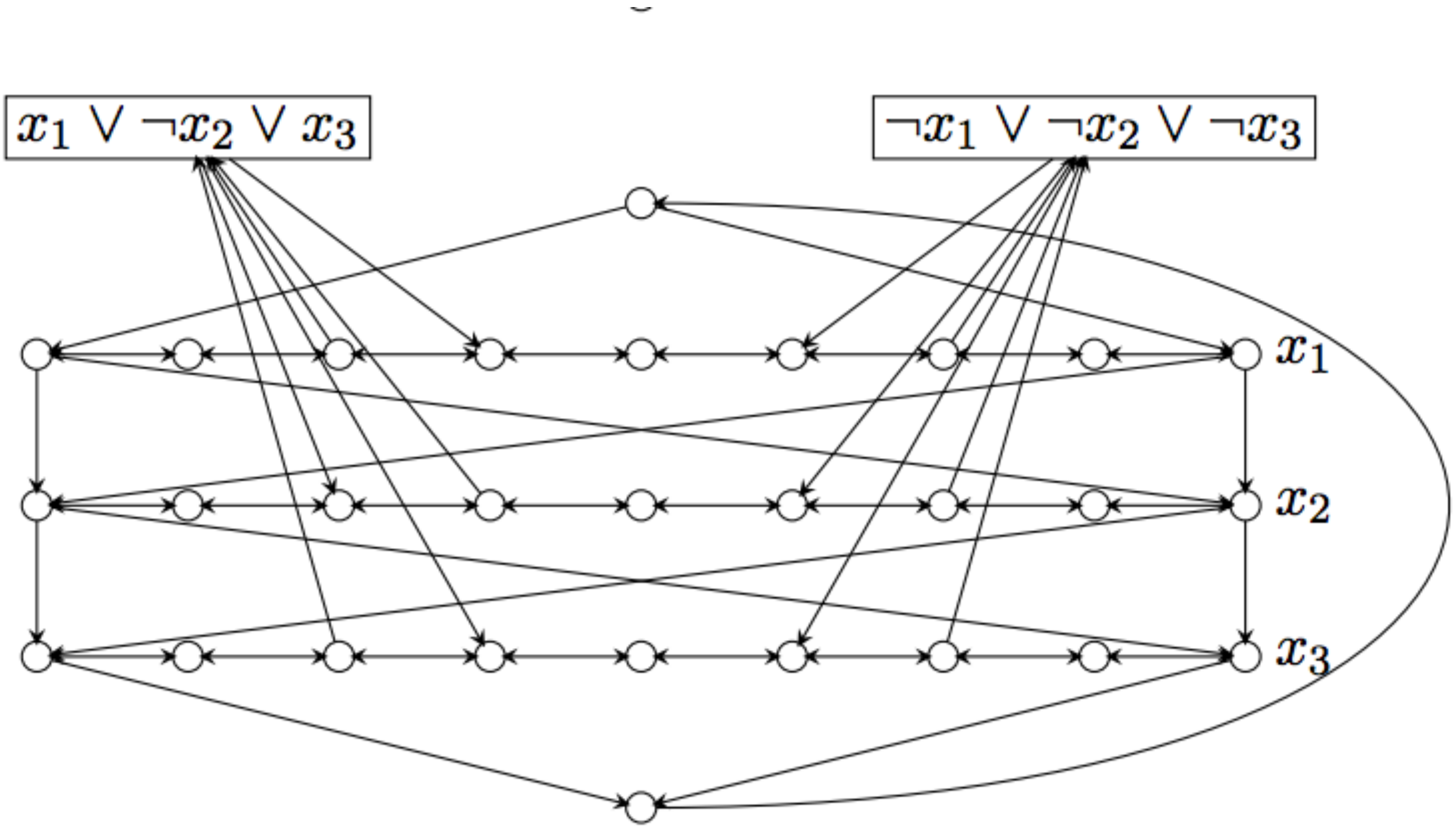
- Liveness: A variable is live if its current assignment may be used at some future point in a program's flow
- Optimizers often try to keep live variables in registers
- If two variables are simultaneously live, they need to be kept in separate registers
- Consider the K-coloring problem (can the nodes 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 because it's an isomorphism

HAMILTONIAN CIRCUIT (HC) DECISION PROBLEM IS NP-HARD

HC Variable Gadget



HC Gadgets Combined


$x_1 \vee \neg x_2 \vee x_3$
$\neg x_1 \vee \neg x_2 \vee \neg x_3$
x_1
x_2
x_3

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.

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

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} $(1 \leq i , j \leq m)$, 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} $(1 \leq i , j \leq n)$; 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. I will discuss this a bit more at a later time.

2 Processor scheduling

The problem of optimally scheduling n tasks T_1, T_2, …, 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

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

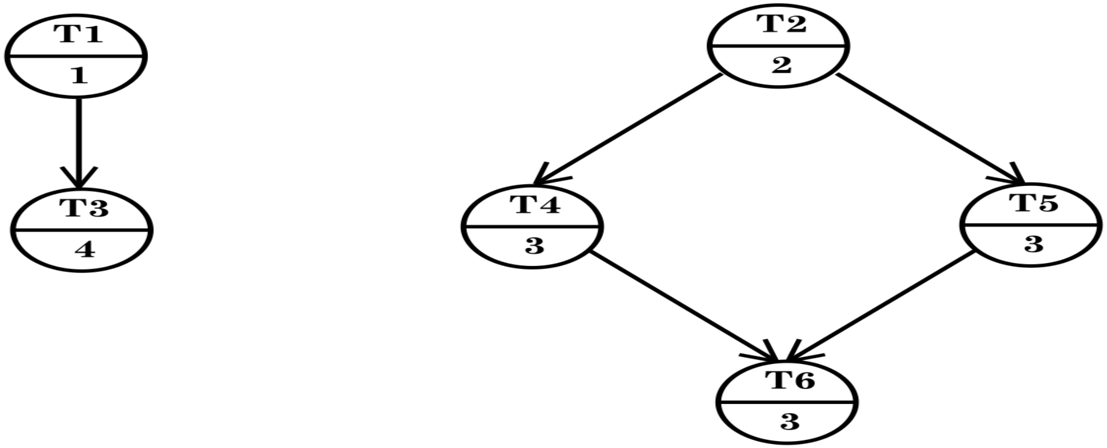
7, 7, 6, 6, 5, 5, 4, 4, 4

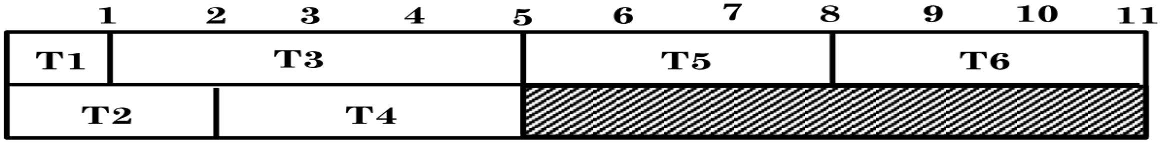
7, 6, 5, 4, 4 = 26

7, 6, 5, 4 = 22

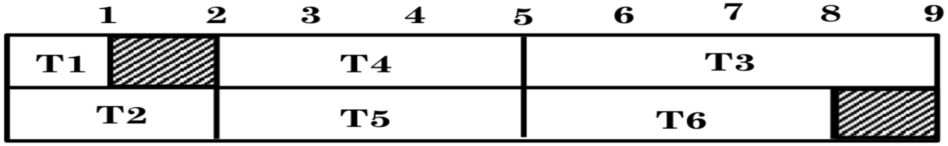
Even worse than greedy unsorted

2 Processor with partial order

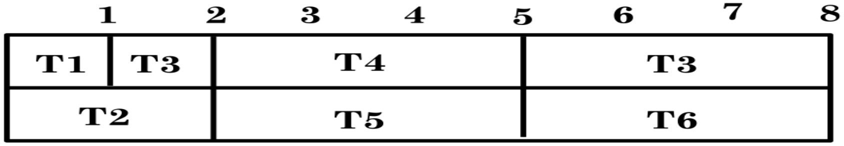




List Schedule with L = {T1,T2,T3,T4,T5,T6}

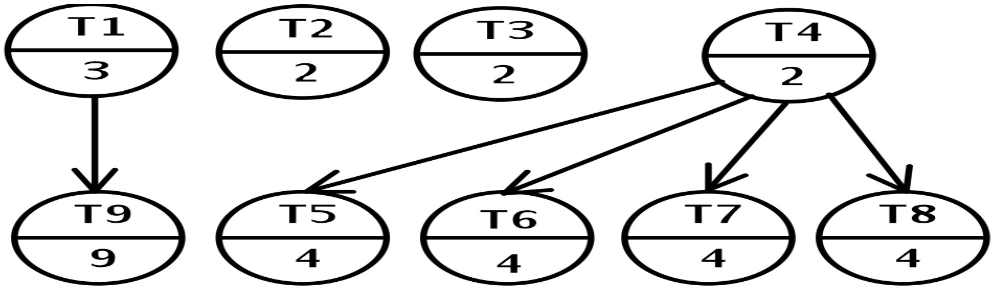


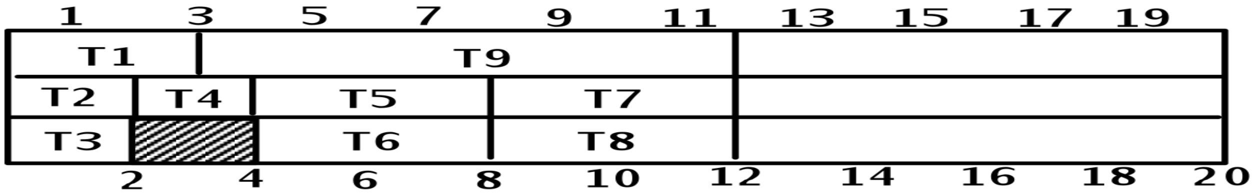
Non-Preemptive, Delays Allowed



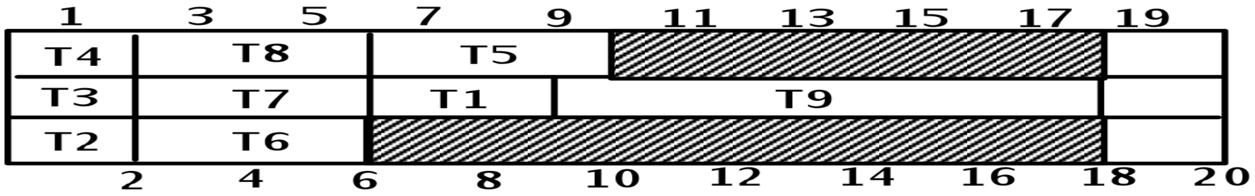
Preemptive

Anomalies everywhere

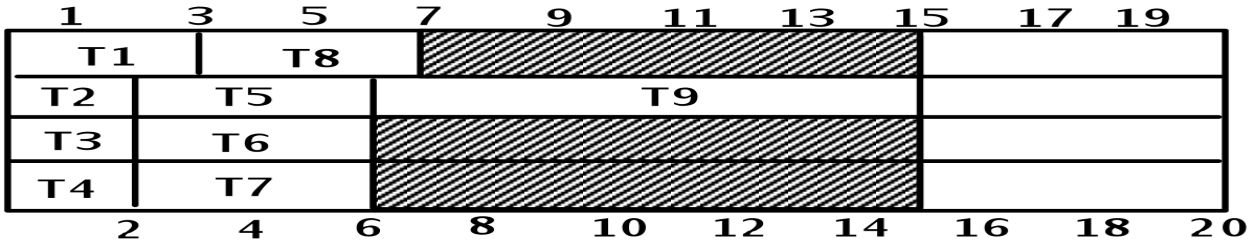




List Schedule with L = {T1,T2,T3,T4,T5,T6,T7,T8,T9

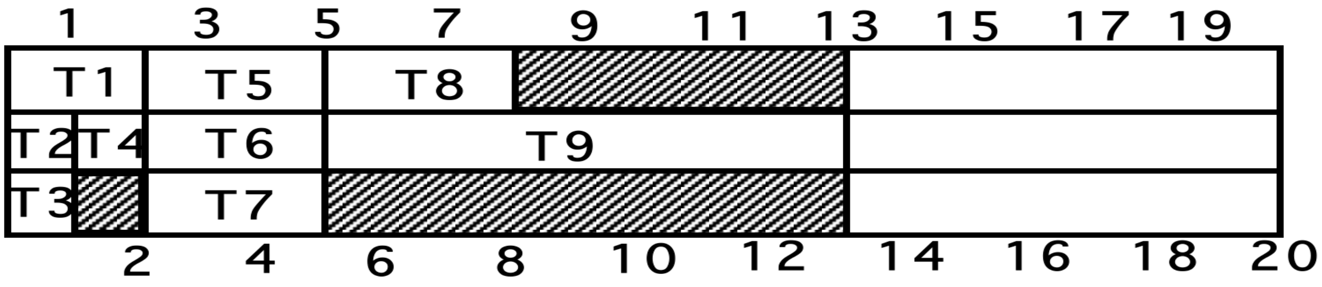


List Schedule with L = {T9,T8,T7,T6,T5,T4,T3,T2,T1

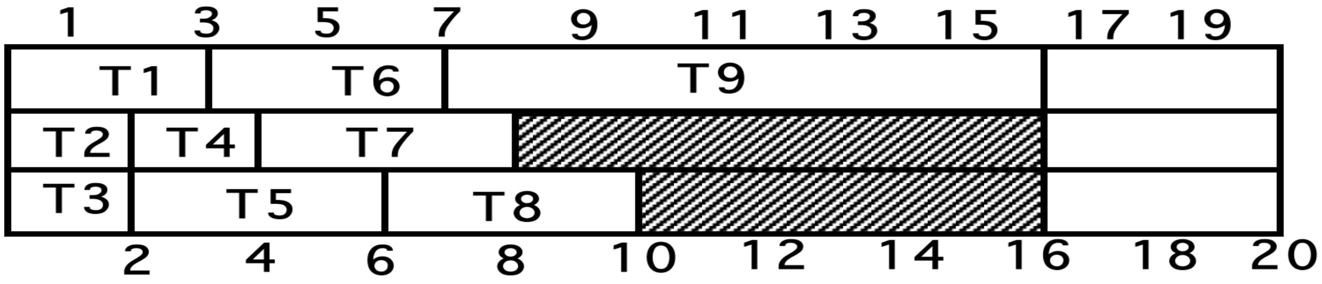


Use Original List with 4 Processors

More anomalies



Original List Schedule but with All Times Reduce



Original List Schedule but with T5 and T6 Inde

Heuristics

While it is not known whether or not P = NP?, it is clear that we need to "solve" problems that are NP-complete since many practical scheduling and networking problems are in this class. For this reason we often choose to find good "heuristics" which are fast and provide acceptable, though not perfect, answers. The First Fit and Best Fit algorithms we previously discussed are examples of such acceptable, imperfect solutions.

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.

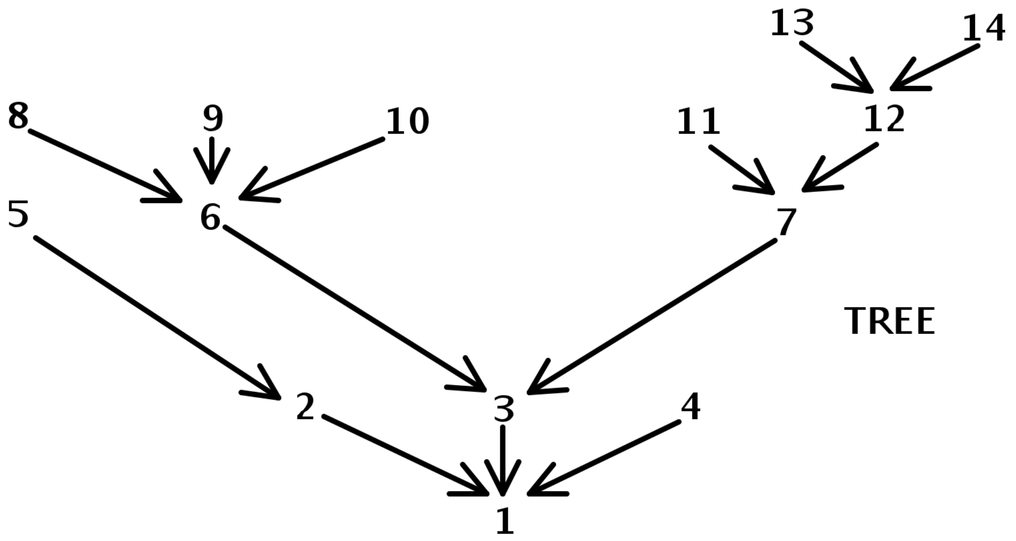
1. Assign L(T) = 1, for the leaf task T
2. Let labels 1, …, k-1 be assigned. If T is a task with lowest numbered immediate successor then define L(T) = k (non-deterministic)

   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, then 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#8.1

1. 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)

Now show what would happen if the times were sorted non-decreasing.

Now show what would happen if the times were sorted non-increasing.

Assignment#8.2

2. Looking back at Page 518, 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) Write the Gantt chart down that shows the new schedule associated with this enhanced tree
   b) Write down the Gantt chart that is associated with the corresponding anti-tree, in which all arc are turned in the opposite direction. Use the technique of reversing the schedule from (a)
   c) Write down 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 (2) versus (3), providing insight as to why they are different and why one is better than the other.

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,…, vn and clauses c1,…, cm
- For each variable vi, have constraint that 0 ≤ vi ≤ 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 ⋁ ~v3 ⋁ v5 ⋁ v6 then add the constraint
  v2 + (1-v3) + v5 + v6 ≥ 1
- A solution to this set of integer linear constraints implies a solution to the instance of SAT and vice versa
- Can also start with SubsetSum – think about it.

Fourth Significant Class of Problems:

The Class Co–NP

(The COmplement problems of NP)

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 a complement 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?

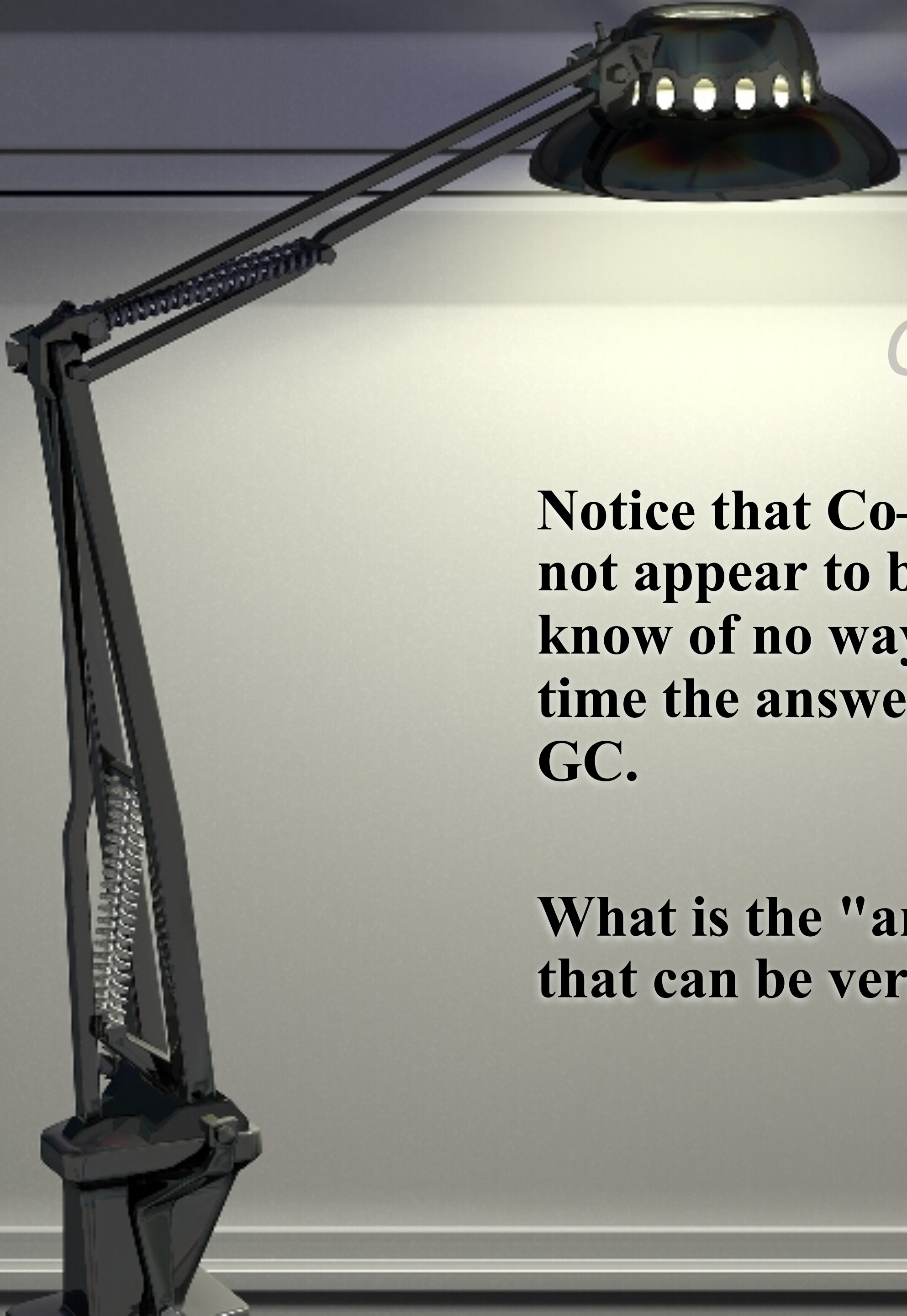
Co-NP

The complement problem of GC

Co–GC

Given: A graph G and an integer k.

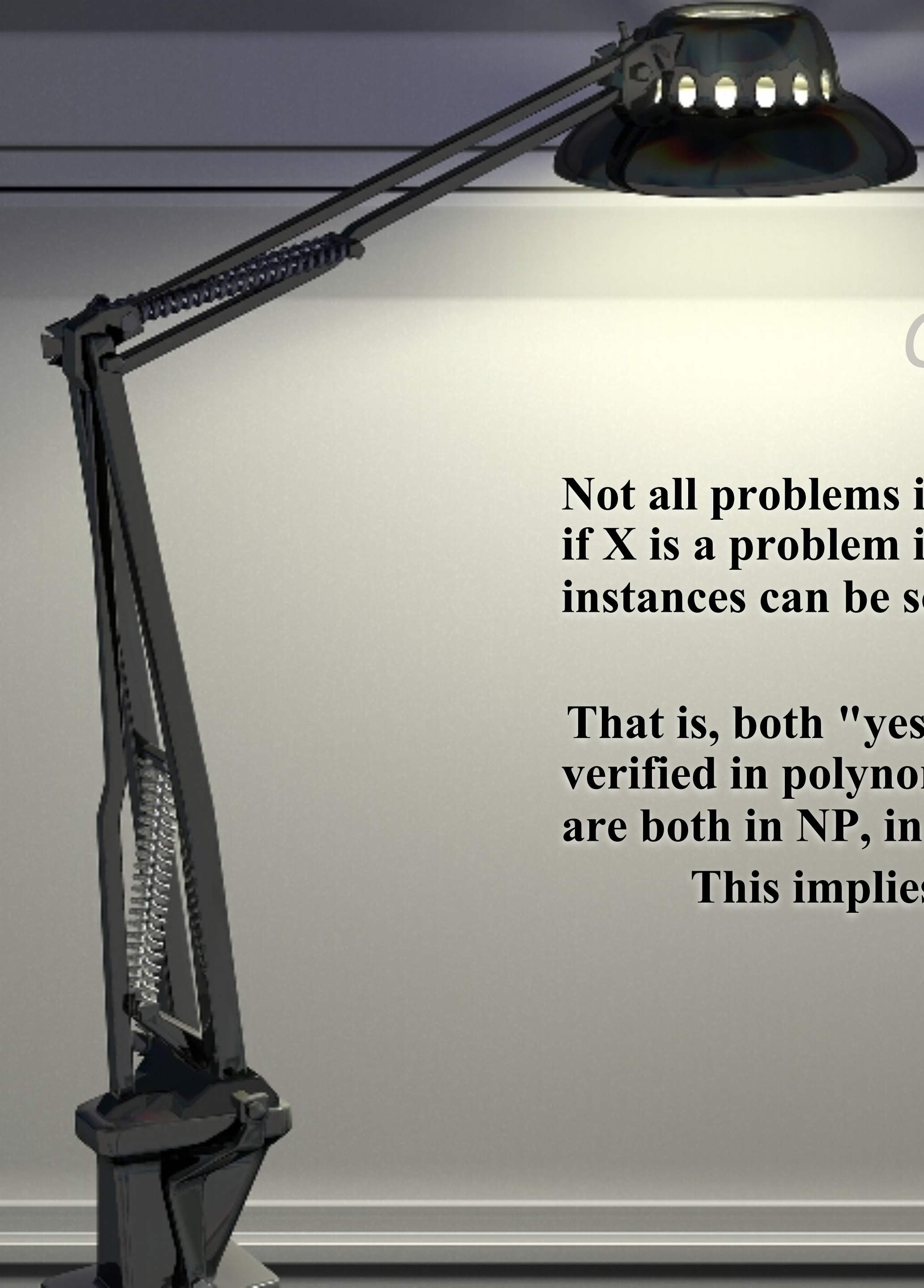
Question: Do all proper colorings of G require more than k colors?



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?



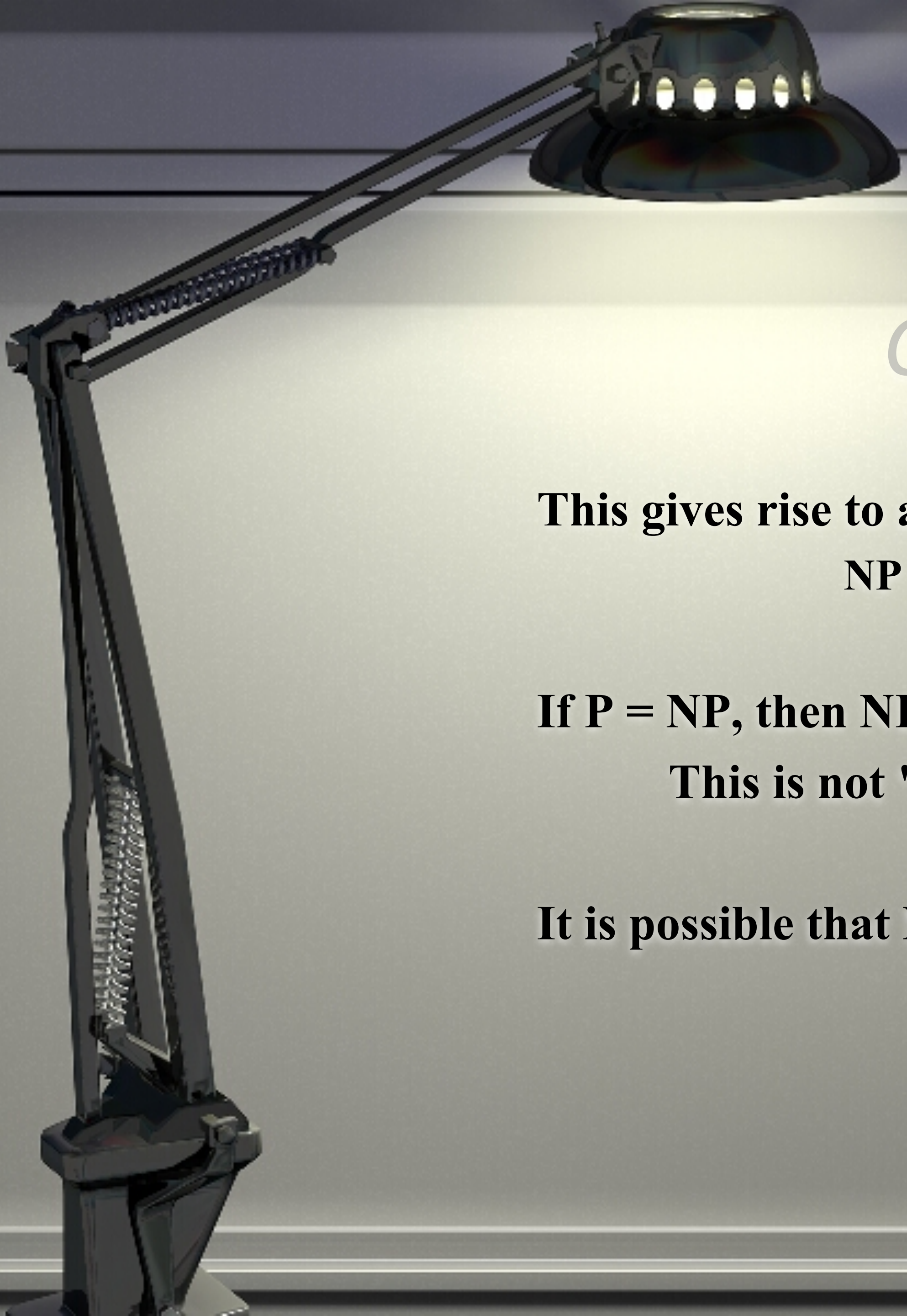
Co-NP

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 ≠ NP.

Co-NP

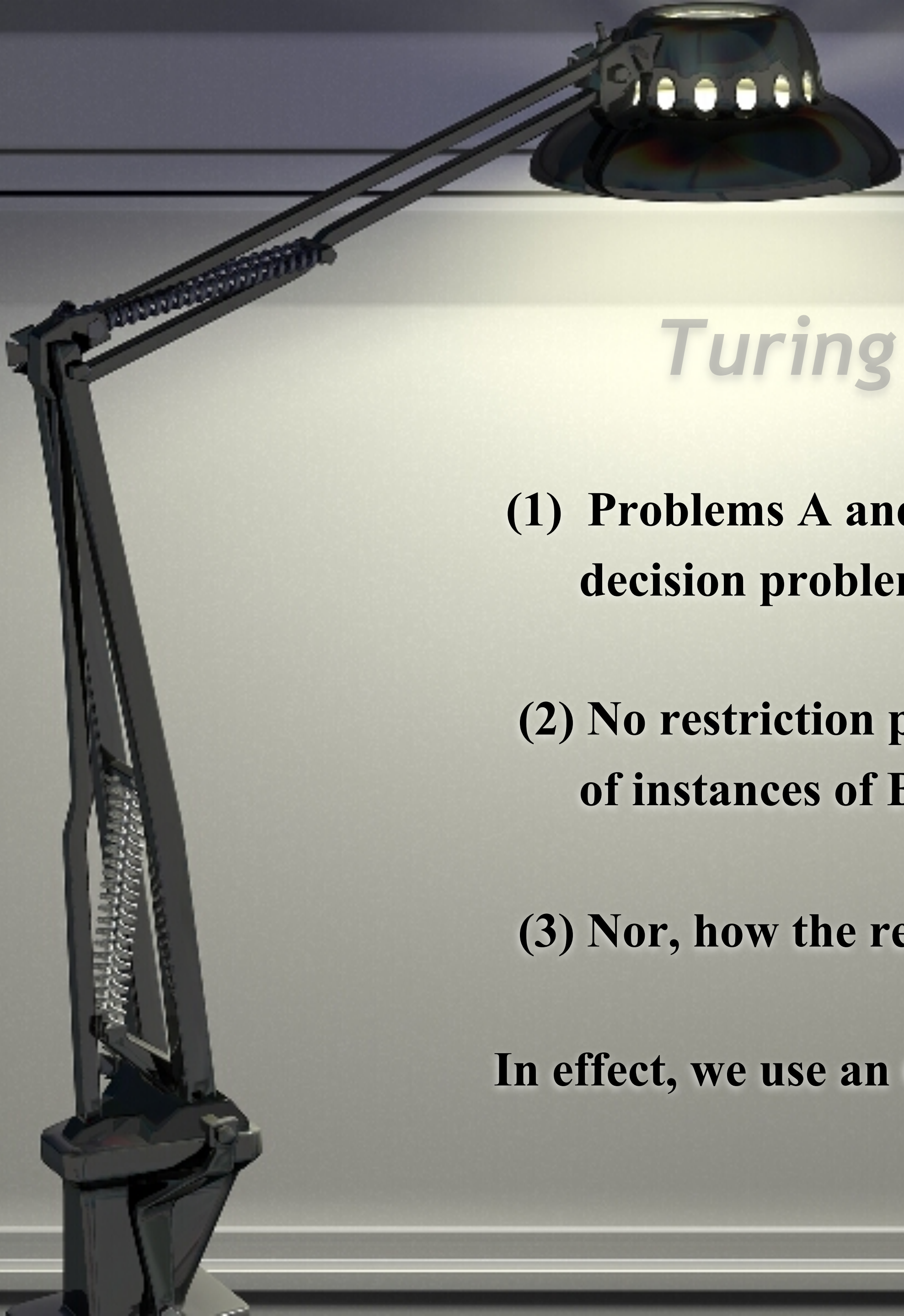
If A ➤$_P$ B and both are in NP, then the same polynomial transformation will reduce Co-A to Co–B. That is, Co–A ➤$_P$ Co–B. Therefore, Co–SAT is 'complete' in Co–NP.

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

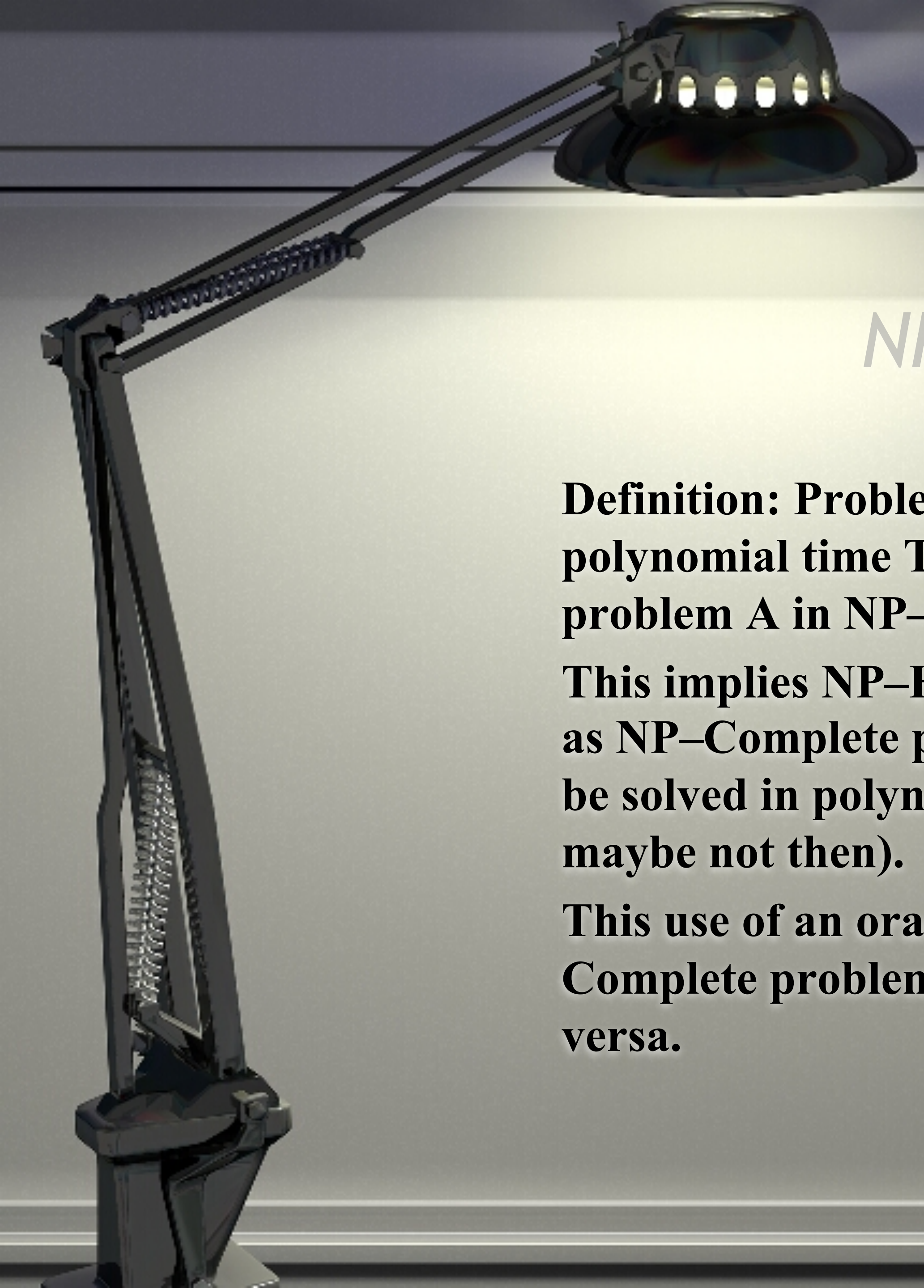
Fifth Significant Class of Problems:

The Class NP–Hard

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 ➤ B for some problem A in NP–Complete.

This implies NP–Hard problems are at least as hard as NP–Complete problems. Therefore, they can not 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-Equivalent

Co-NP problems are solvable in polynomial time if and only if their complement problem in NP is solvable in polynomial time.

Due to the existence of Turing reductions reducing either to the other.

Other problems not known to be in NP can also have this property (besides those in Co-NP).

NP-Equivalent

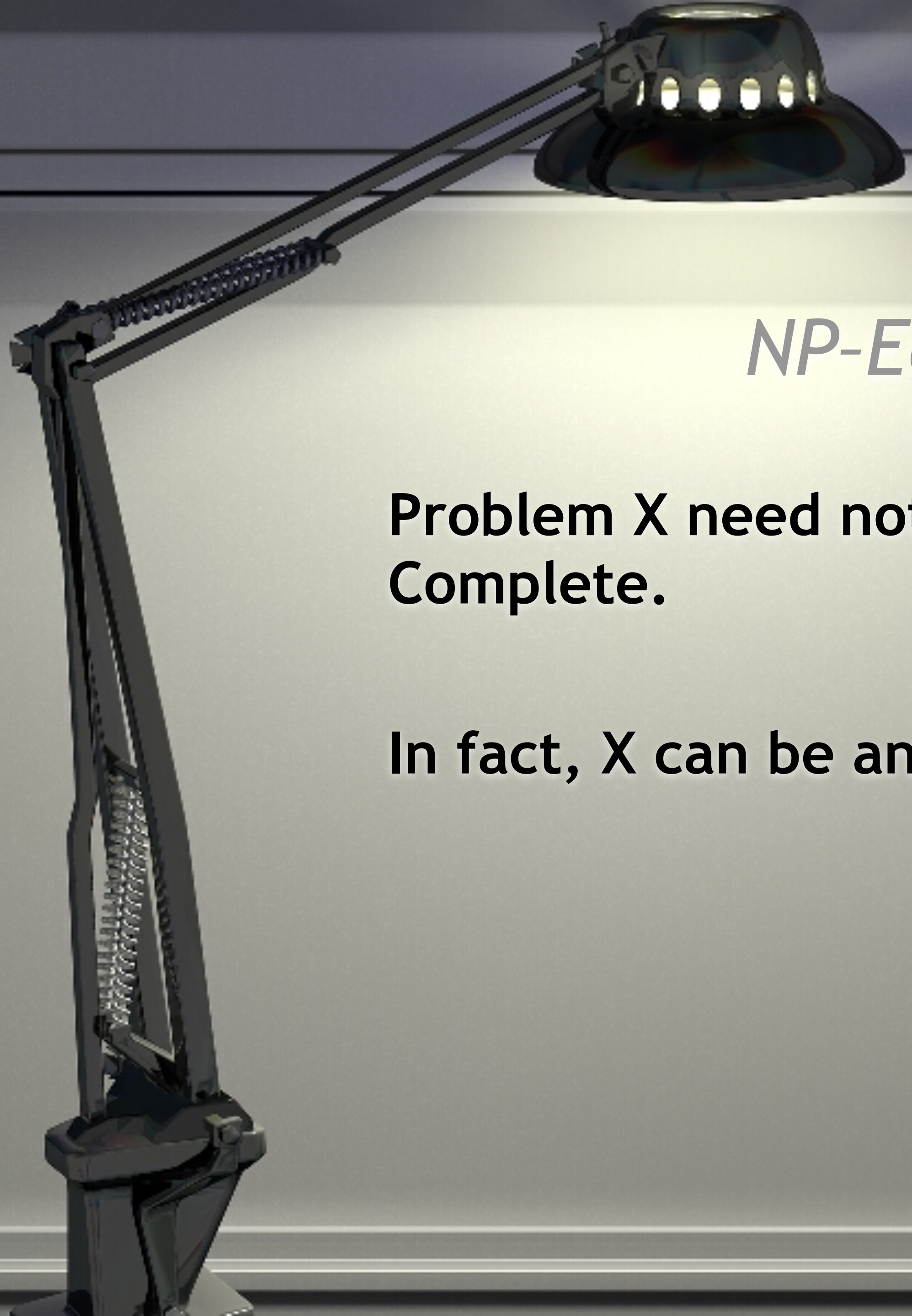
Problem B in NP-Hard is *NP-Equivalent* when B reduces to any problem X in NP, That is, B ➤ X.

Since B is in NP-Hard, we already know there is a problem A in NP-Complete that reduces to B. That is, A ➤ B.

Since X is in NP, X ➤ A. Therefore, X ➤ A ➤ B ➤ X.

Thus, X, A, and B are all polyomially equivalent, and we can say

Theorem. Problems in NP-Equivalent are polynomial if and only if P = NP.



NP-Equivalent

Problem X need not be, but often is, NP-Complete.

In fact, X can be any problem in NP or Co-NP.

NP-Hard

- NP-Hard has two competing definitions
- Some think of NP-Hard as involving many-one reductions and thus being decision problems
- Others think of NP-Hard as involving Turing reductions and thus being functional problems
- We will choose the latter which includes the former. Note, there is no conflict with NP-Complete since an NP-Complete problem is in NP and is therefore a decision problem

FP and FNP

- FP is functional equivalent to P R(x,y) in FP if can provide value y for input x via deterministic polynomial time algorithm
- FNP is functional equivalent to NP; R(x,y) in FNP if can verify any pair (x,y) via 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 (≤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 (≤tp) that uses rather than mimics answers from its oracle

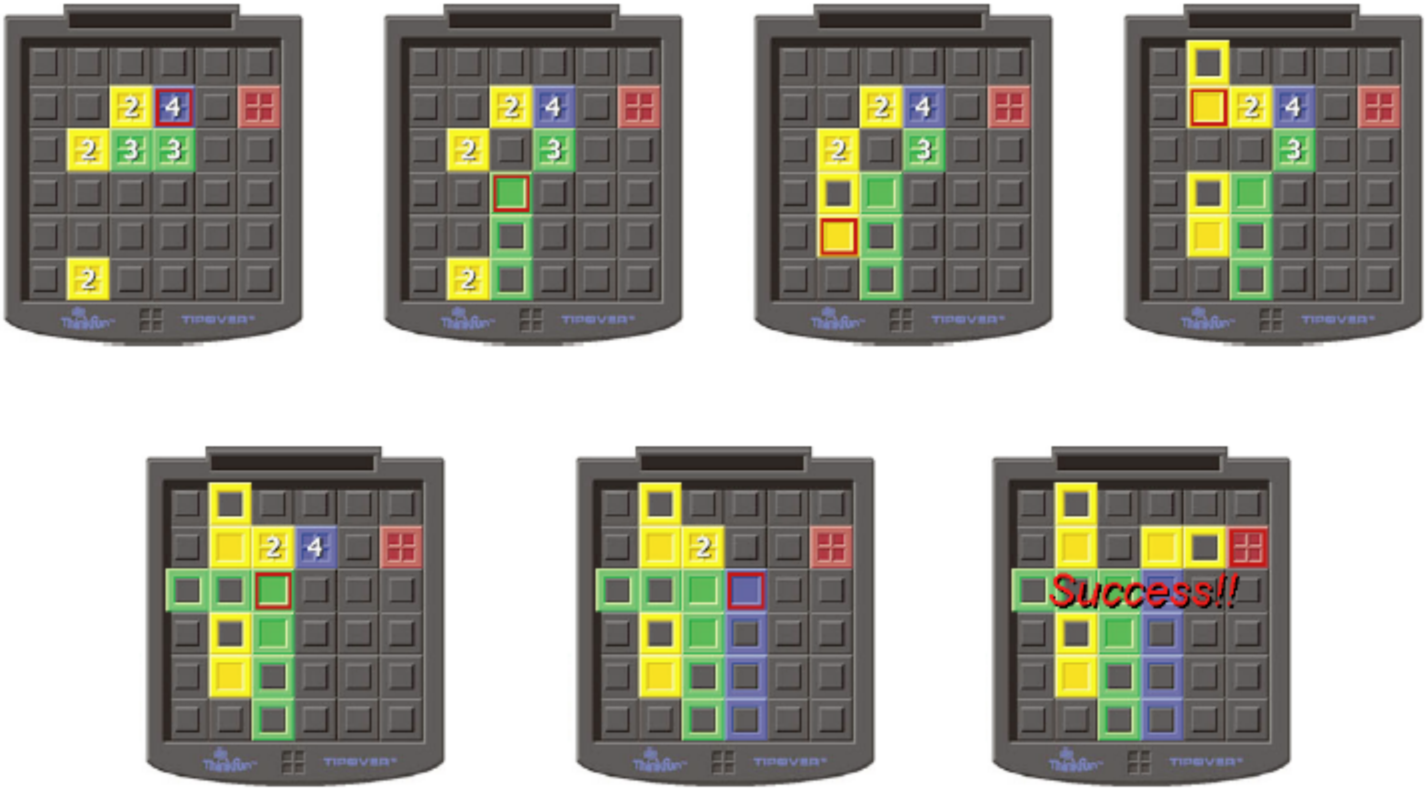
NP-Easy and Equivalent

- NP-Easy -- these are problems that are polynomial when using an NP oracle (≤tp)
- NP-Equivalent is the class of NP-Easy and NP-Hard problems (assuming Turing rather than many-one reductions)
  - In essence this is functional equivalent of NP-Complete but also of Co-NP-Complete since can negate answers

More Examples of NP Complete Problems

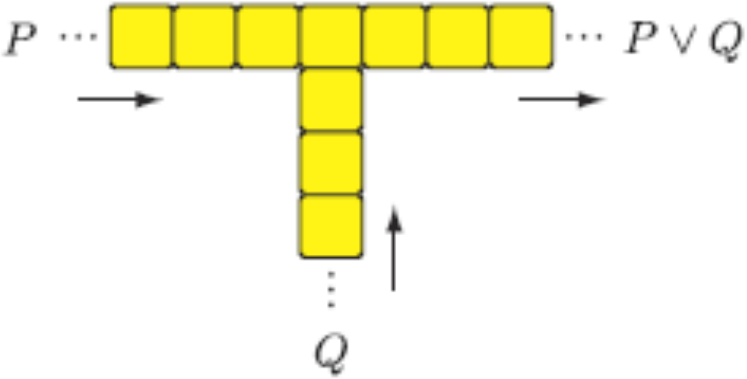
TipOver

Rules of Game



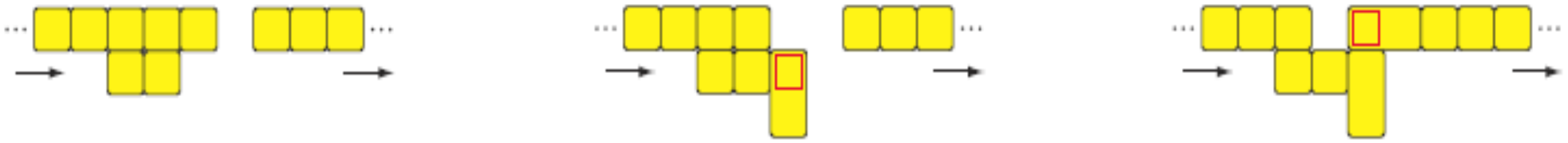
**Numbers are height of crate stack;
If could get 4 high out of way we can attain goal**

Problematic OR Gadget



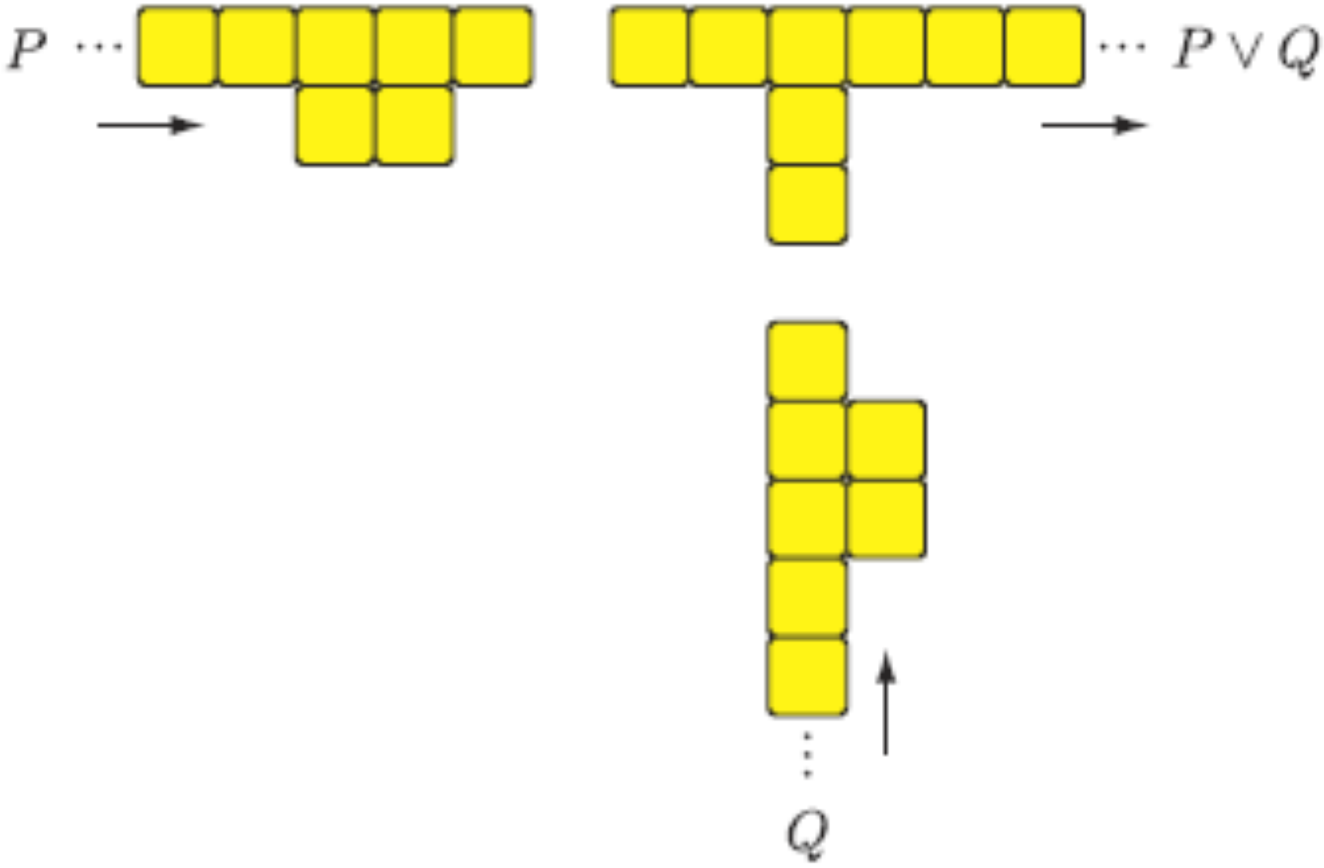
Can go out where did not enter

Directional gadget

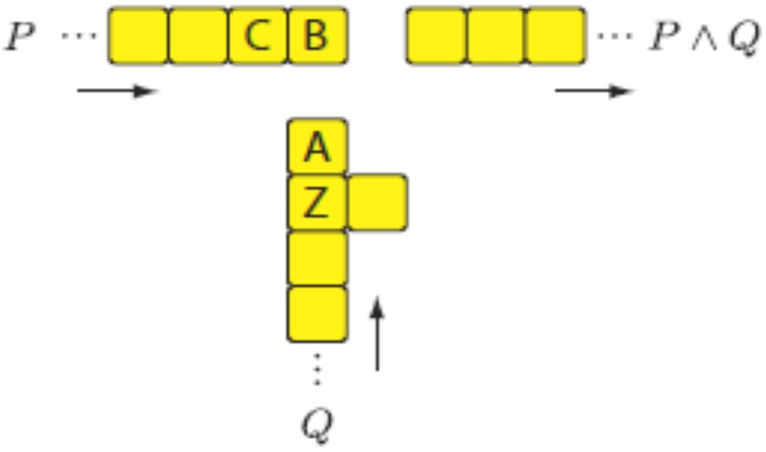


Single stack is two high;
tipped over stack is one high, two long;
red square is location of person travelling the towers

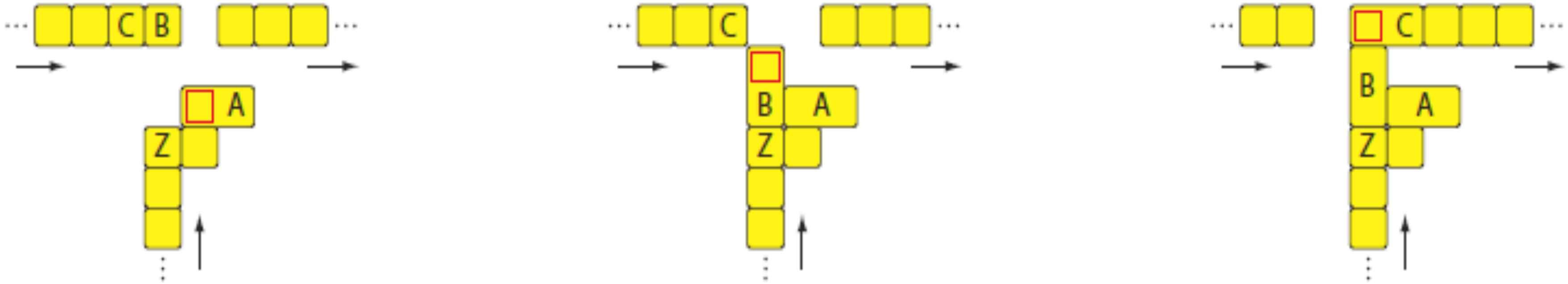
One directional Or gadget



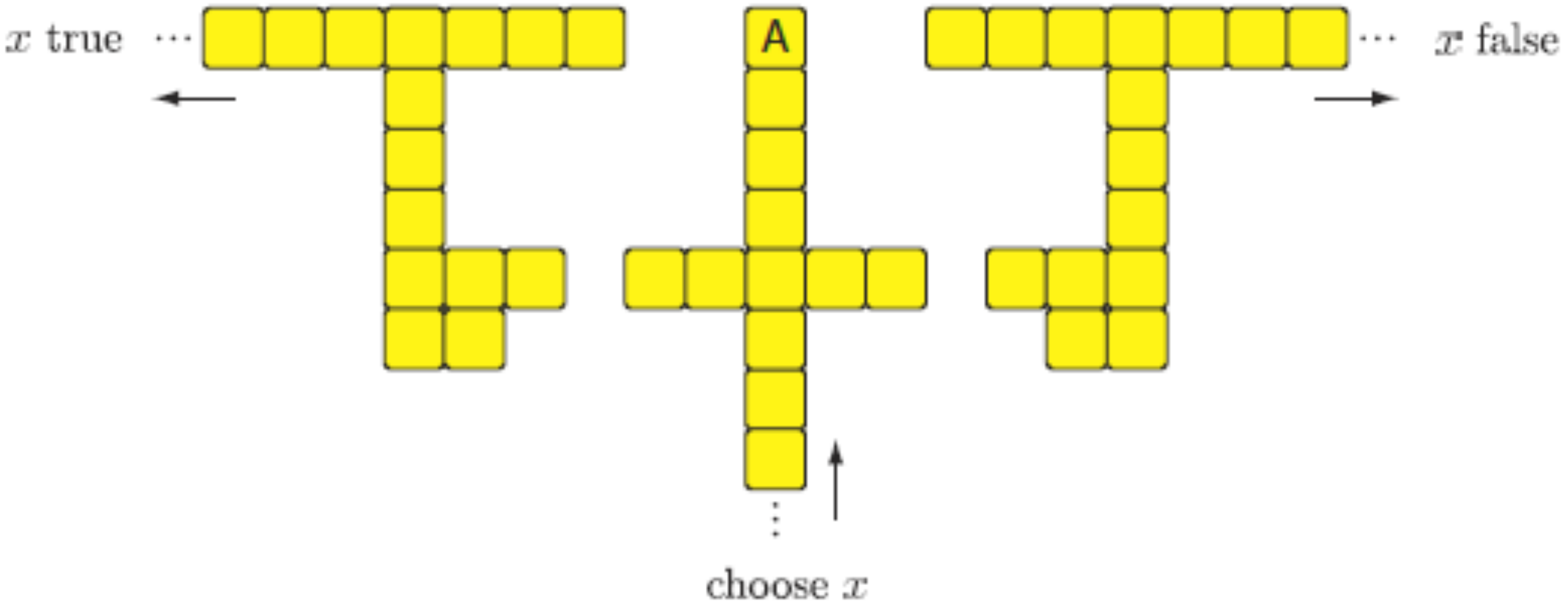
AND Gadget



How AND Works

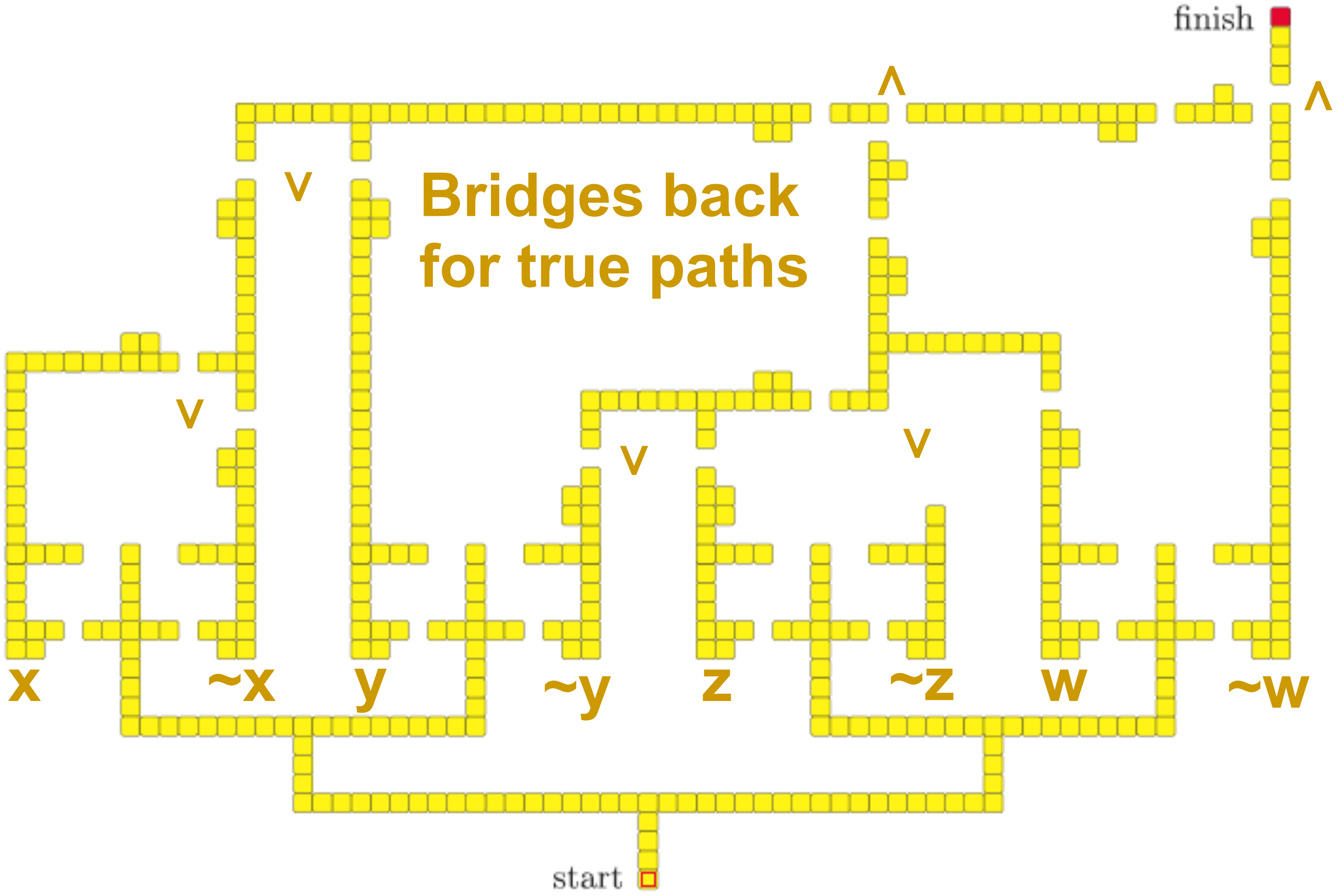


Variable Select Gadget



Tip A left to set x true; right to set x false
Can build bridge to go back but never to change choice

((x∨~x∨y)∧(~y∨z∨w)∧~w)



Bridges back for true paths

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

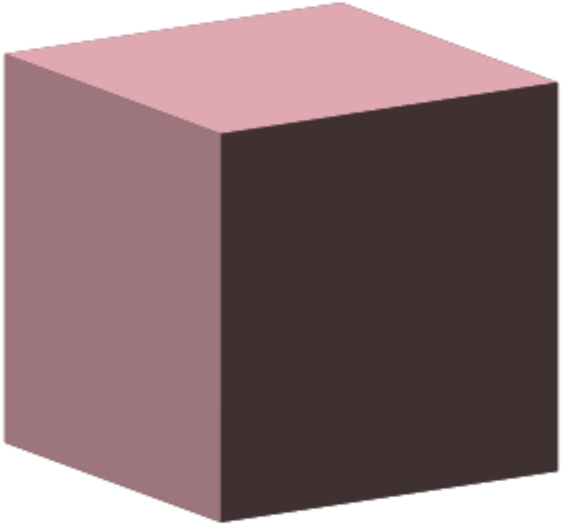
Adapted from presentation by
Ajit Hakke Patil
Spring 2010

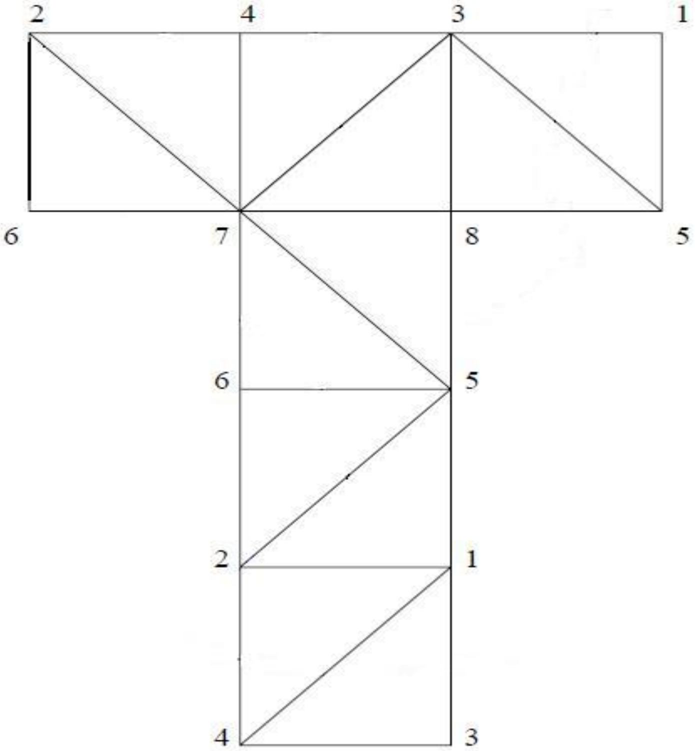
Graphics Subsystem

- The graphics subsystem (GS) receives graphics commands from the application running on CPU 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



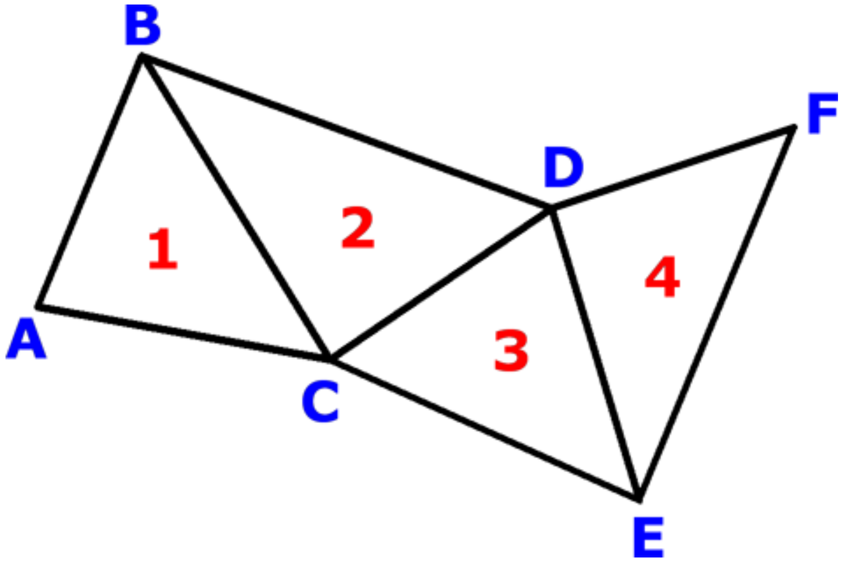


Triangle List vs Triangle Strip

- 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.

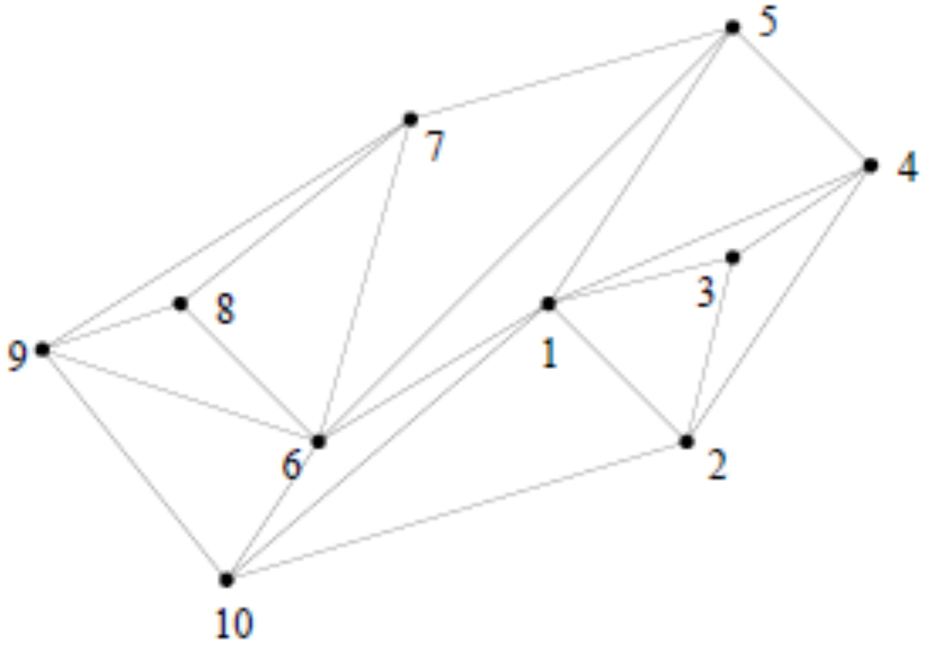
Triangle List vs Triangle Strip

- 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

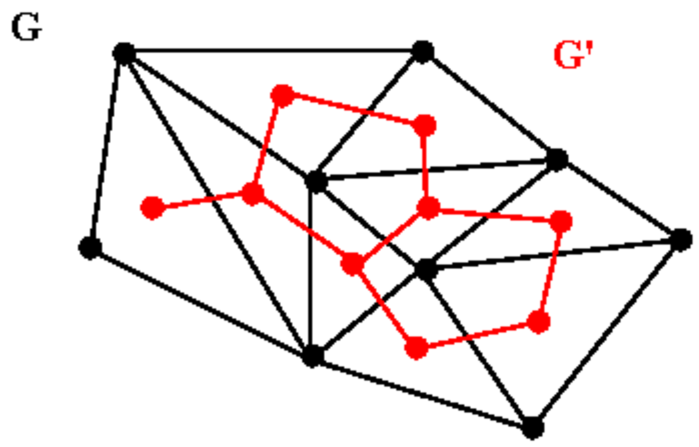
- Given a triangulation $T = \{t_1, t_2, t_3 ,.. t_n\}$. Find the triangle strip (sequential ordering) for it?
- Converting this to a decision problem.
- Formal Definition:
- Given a triangulation $T = \{t_1, t_2, t_3 ,.. t_N\}$. Does there exists a triangle strip?

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

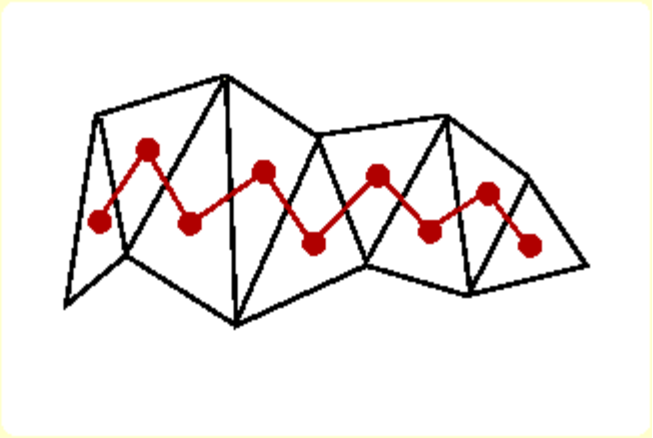
- To prove it’s NP-Complete we reduce a known NP-Complete problem to this one; the Hamiltonian Path Problem.
- Hamiltonian Path Problem:
  - Given: A Graph G = (V, E). Does G contains a path that visits every vertex exactly once?

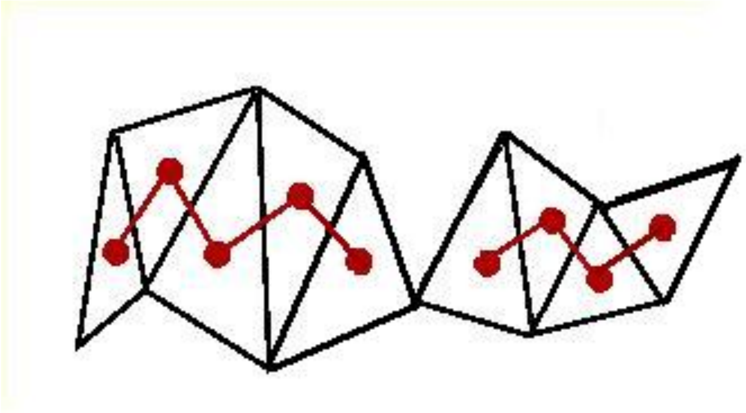
NP-Completeness proof by restriction

- 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