Computational Models

Computer Science & Engineering 235: Discrete Mathematics

Christopher M. Bourke
cbourke@cse.unl.edu

Introduction I

At the foundation of computer science, we have questions like

- What are the minimum amount of resources required to solve a given problem?
- Given enough time or memory, can an algorithm be designed to solve *any* problem?
- Can algorithms solve anything and everything?

In short, the answer is *no*. Even given an infinite amount of resources, there are problems that *no* algorithm can solve.

Introduction II

Up to now, we have studied algorithms and problems from a more *intuitive* approach.

In fact, the study of algorithms, and more generally, *computation* and *computational complexity theory* have a much more rigorous theoretical basis.

In particular, we must have a rigorous mathematical model of computation.

Introduction III

In truth, these topics deserve entire courses unto themselves, covering specific topics such as

- Language Theory
- Computational Models
- Turing Machines
- Computability Theory
- Reductions: Halting Problem
- Complexity Classes P & NP
- NP-Completeness
- Reductions

We will attempt to give a broad overview of some of these topics.

Languages

As with anything that we want to study, we must always establish a mathematical framework, a *model* by which to work within.

For instance, *problems* and algorithms come in many forms;

- Numerical
- Graph
- Sorting
- Logic

We need a *unified* model that captures all different types of problems and algorithms.

Rather than looking at problems, we look at *languages*.

Languages

Definition

- An *alphabet*, Σ is a finite, nonempty set of symbols. Examples: [A-Za-z], [0–9], $\{0,1\}$.
- A *string* (or *word*) over Σ is a finite combination of symbols from Σ. Example: any integer is a string over $\Sigma = [0-9]$.
- The *empty string*, denoted ϵ is a string containing no symbols from Σ.
- The set of all finite strings over Σ is denoted Σ^*.
- A *language* over Σ is a set (finite or infinite), $L \subseteq \Sigma^*$.

Language Operations

We restrict our consideration to a binary alphabet, $\Sigma = \{0, 1\}$ and consider binary strings.

Operations can be performed on languages:

- **Union** – $L_1 \cup L_2$
- **Concatenation** – $L_1 \circ L_2$ (or just $L_1 L_2$)
- **Kleene Star** – L^*

Many other operations exist.

Concatenation

The *concatenation* of two languages is the concatenation of all strings in each language.

$$L_1 L_2 = \{xy \mid x \in L_1, y \in L_2\}$$

Example

Let $L_1 = \{0, 10\}, L_2 = \{1, 01\}$. Then

$$L_1 L_2 = \{01, 001, 101, 1001\}$$

and

$$L_1 L_1 = \{00, 010, 100, 1010\}$$

Kleene Star

The Kleene Star is a recursively defined operation. For a given language L, $L^0 = \{\lambda\}$ and for $n > 0$, we define

$$L^n = L^{(n-1)} L$$

Example

For $L = \{0, 10\}$,

$$\begin{aligned} L^0 &= \{\lambda\} \\ L^1 &= \{0, 10\} \\ L^2 &= \{00, 010, 100, 1010\} \\ L^3 &= \{000, 0010, 0100, 01010, 1000, 10010, 10100, 101010\} \end{aligned}$$

Kleene Star

The Kleene Star operation is then defined as the union of all such concatenations.

$$L^* = \bigcup_{n \geq 0} L^n$$

For the alphabet Σ itself, Σ^* is the set of all binary strings.

Sometimes it is useful to use the following notation to consider only *nonempty* strings.

$$L^+ = \bigcup_{n \geq 1} L^n = L^* - \{\lambda\}$$

Regular Expressions

We say that R is a *regular expression* if

- $R = b$ for some bit $b \in \Sigma$
- $R = \lambda$
- $R = \emptyset$
- $R = (R_1 \cup R_2)$ where R_1, R_2 are regular expressions.
- $R = (R_1 \circ R_2)$ where R_1, R_2 are regular expressions.
- $R = (R_1^*)$ where R_1 is a regular expression.

Regular expressions are used in `grep`, `sed`, `vi`, Java, Perl, and most other scripting languages.

Regular Languages

Regular languages are those that can be generated by a regular expression.

Example

- $0^* \cup 1^*$ is the language consisting of all strings with *either* all 1s or all 0s (plus the empty string).
- 0^*10^* is the language consisting of all strings with a single 1 in them.
- $(\Sigma\Sigma)^*$ the set of all even length strings
- $1\Sigma^*0$ the set of all canonical representation of even integers.

Exercise: Give a regular expression for the set of all strings where every 0 appears *before* any occurrence of a 1.

Decision Problems I

An *instance* of a *decision problem* involves a given configuration of data.

An algorithm answers

- *yes* if the data conforms to or has some property, and
- *no* if it does not.

Example

Given: an undirected graph G;
Question: does there exist an Euler cycle in G?

Decision Problems II

Though many natural problems (optimization, functional) are not decision problems, we can usually formulate the *decision version* of them.

Example

For any optimization problem: "what is the maximum (minimum) number of x such that property $\mathcal{P}$ holds?"

Can be reformulated as,

"Does property $\mathcal{P}$ hold for all $x \geq k$?"

Languages = Problems

It is not hard to see that languages are equivalent to problems.

That is, given a problem, you can define a language that represents that problem.

Problem (Sorting)

Given *elements* $x_0, \ldots, x_{n-1}$ *(properly encoded) and an ordering* $\preceq$.

Question: *is* $x_i \preceq x_{i+1}$ *for* $0 \leq i \leq n-2$*?*

Languages are Universal I

The language model is *robust*. Any problem $\mathcal{P}$ can be equivalently stated as a language L where

- (Encodings) x of *yes* instances are members of the language; $x \in L$.
- (Encodings) x of *no* instances are not members of the language; $x \notin L$.

The key is that we establish a proper *encoding* scheme.

Languages are Universal II

A proper encoding of graphs, for example, may be a string that consists of a binary representation of n, the number of vertices.

Using some delimiter (which can also be in binary), we can specify connectivity by listing pairs of connected vertices.

$$\langle G \rangle = \texttt{11:00:01:01:10}$$

We can then define a language,

$$L = \{\langle G \rangle \mid G \text{ is a connected graph}\}$$

Languages are Universal III

Graph connectivity is now a language problem;

- $\langle G \rangle \in L$ if G is a (properly encoded) graph that is connected.
- $\langle G \rangle \notin L$ if G is not connected.

Instead of asking if a given graph G is connected, we instead ask, is $\langle G \rangle \in L$?

Introduction

Now that we have a framework for problems, we need one for algorithms.

There are many different computational models corresponding to many *classes* of languages.

Some are provably more powerful than others. Here, we give a brief introduction to

- Finite State Automata
- Grammars
- Turing Machines

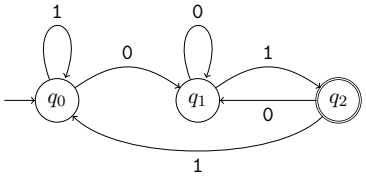
Finite-State Automata

Definition

A *finite automaton* is a 5-tuple, $\mathcal{A} = (Q, \Sigma, \delta, q_0, F)$ where

- Q is a nonempty finite set of *states*
- Σ is our alphabet
- $\delta : Q \times \Sigma \to Q$ is the *transition function*
- $q_0 \in Q$ is an *initial state*
- $F \subseteq Q$ is the set of *accept states*

Example I



- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{0, 1\}$
- q_0 is our initial state
- $F = \{q_2\}$

Example II

The transition function is specified by the labeled arrows.

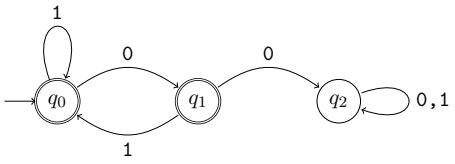
$$\begin{aligned}
\delta(q_0, 0) &= q_1 \\
\delta(q_0, 1) &= q_0 \\
\delta(q_1, 0) &= q_1 \\
\delta(q_1, 1) &= q_2 \\
\delta(q_2, 0) &= q_1 \\
\delta(q_2, 1) &= q_0
\end{aligned}$$

Acceptance = Language

Exercise

Exercise

Design a finite-state automaton to accept the language consisting of any string in which contain no contiguous 0s.



Acceptance = Language I

For the previous example its not hard to see that the set of strings $\mathcal{M}$ *accepts* is any string that ends in 01. An equivalent regular expression is simply

$$\Sigma^* 01$$

Acceptance = Language II

The set of strings that a finite-state automaton $\mathcal{M}$ accepts is its *language*:

$$L(\mathcal{M}) = \{x \in \Sigma^* \mid \mathcal{M}(x) \text{ accepts}\}$$

Conversely, any string that ends in a non-accept state is *rejected*. This also defines a language–the *compliment* language:

$$\overline{L(\mathcal{M})} = \Sigma^* - L(\mathcal{M})$$

Power of Computational Models I

Finite-state automata are one of the simplest computational models.

Unfortunately, they is also restrictive. The only types of languages it can *recognize* are those that can be defined by regular expressions.

Theorem

Finite-State Languages = Regular Languages = Regular Expressions

Power of Computational Models II

Recognition here means that a machine, given a finite string $x \in \Sigma^*$ can tell if $x \in L(\mathcal{M})$.

Such a computational model cannot, for example, recognize non-regular languages like

$$L = \{w \in \Sigma^* \mid w \text{ has an equal number of 0s and 1s}\}$$

Other Computational Models I

There are many other computational models that have been considered, each one defines a class of languages that it can recognize.

Context-Free Languages correspond to *Push-down automata*; finite state automatons that have access to a stack. These languages also correspond to certain types of *grammars*.

These languages are of fundamental importance to the theory of computer languages and compilers.

Though these models are interesting, we will limit further consideration to a more general computational model that considers general problems and algorithms rather than certain classes.

Other Computational Models II

For this we turn our attention to *Turing Machines*.

Though there are many variations (multi-tape, multi-head, randomized, RAM, etc) we will only consider the following *basic* definition.

Note: though there are many variations, from a *computability* point of view, the types of languages recognized by all Turing Machines are the same.

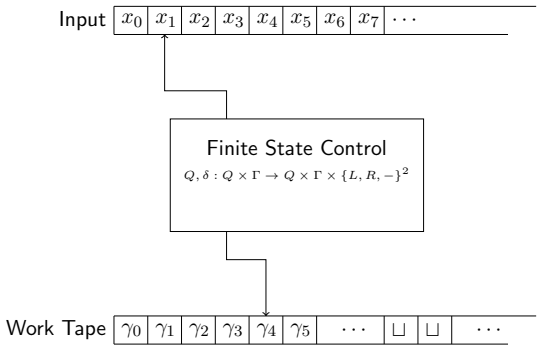
Turing Machines I

Definition

A *Turing Machine* is a 7-tuple, $\mathcal{M} = (Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$ where

- Q is a nonempty, finite set of states
- Σ is the input alphabet
- Γ is the tape alphabet
- $\delta : Q \times \Sigma \to Q \times \Gamma \times \{L, R, -\}^2$ is the transition function
- $q_0 \in Q$ is the initial state
- q_{accept} is the accept state
- q_{reject} is the reject state

Turing Machines II



Turing Machines III

A *Turing Machine* is a basic computational model which has an input tape that can be read from, an output tape that can be written on and a set of states.

A tape head moves left and right along the input/output tape and performs reads and writes according to what symbols it encounters.

Turing Machines IV

A definition of a given Turing Machine can be made precise by enumerating every possible transition on every possible input and output symbol for every state.

A state diagram similar to automatons can visualize this transition. However, it is much easier to simply *describe* a Turing Machine in high level English.

Turing Machine Example

The following Turing Machine decides the language

$$L = \{x\#x \mid x \in \Sigma^*\}$$

$\mathcal{M}(x)$ (read: on input x)

1. Scan the input to be sure that it contains a single $\#$, if not *reject*.
2. Zig Zag across the tape to corresponding positions on each side of $\#$. If symbols do not match, *reject*, otherwise cross them off (write a blank symbol, $\sqcup$) and continue.
3. After all symbols to the left of $\#$ have been crossed off, check to the right of $\#$, if any symbols remain, *reject* otherwise, *accept*.

Turing Machine Equivalence I

The *Church-Turing Thesis* gives a formal (though debatably not rigorous) definition of what an *algorithm* is.

It states that the intuitive notion of an algorithmic process is exactly equivalent to the computational model of Turing Machines.

This means that *any* rigorous computational model can be simulated by a Turing Machine. Moreover, *no* other computational model is *more* powerful (in terms of the *types* of languages it can accept) than a Turing Machine.

Turing Machine Equivalence II

A programming language is *Turing complete* if it can do anything that a Turing machine can do.

As a consequence, any two Turing complete programming languages are equivalent.

Intuitively, anything that you can do in Java, you can do in C++ (algorithmically, we're not talking about specific libraries), Perl, Python, PHP, etc.

R & RE I

As you may have experienced, a program does not necessarily have to *halt*. Infinite loops are possible in Turing machines too.

We say that a Turing machine $\mathcal{M}$ *recognizes* a language L if for every $x \in L$, $\mathcal{M}(x)$ halts and accepts.

A Turing machine $\mathcal{M}$ *decides* a language L if for every string x, $\mathcal{M}(x)$ halts and rejects or accepts (according to L).

R & RE II

A language L is in RE if some Turing machine recognizes it.

RE is the *class of recursively enumerable languages* (also called *computably enumerable*).

For a language in $L \in$ RE, if $x \in L$, then some machine will *eventually* halt and accept it.

If $x \notin L$ then the machine may or may not halt.

R & RE III

A language L is in R if some Turing machine decides it.

R is the *class of recursive languages* (also called *computable*).

Here, if $L \in$ R, then there is some machine that will *halt* on all inputs and is guaranteed to accept or reject.

Its not hard to see that if a language is decidable, it is also recognizable by definition, thus

$$\text{R} \subseteq \text{RE}$$

Church-Turing Thesis: Intuitive Notion

"There exists a Turing Machine $\mathcal{M}$ that *decides* a language L"

$=$

"There exists an Algorithm $\mathcal{A}$ that solves a problem $\mathcal{P}$"

The Halting Problem I

There are problems (languages) that are *not* Turing Decidable: languages $L \in$ RE, $L \notin$ R.

We take as our first example the *halting problem*.

Problem (Halting Problem)

Given: *A Turing Machine $\mathcal{M}$ and an input x.*

Question: *does $\mathcal{M}(x)$ halt?*

The Halting Problem II

This indeed would be a very useful program—once you've compiled a program, you may want to determine if you've screwed up and caused an infinite loop somewhere.

We will show that the halting problem is undecidable.

That is, *no algorithm, program or Turing Machine exists* that could ever tell if another Turing Machine halts on a given input or not.

Halting Problem Proof I

By way of contradiction assume that there exists a Turing Machine $\mathcal{H}$ that decides the halting problem:

$$\mathcal{H}(\langle \mathcal{P}, x \rangle) = \begin{cases} 1 & \text{if } \mathcal{P} \text{ halts on } x \\ 0 & \text{if } \mathcal{P} \text{ does not halt on } x \end{cases}$$

We now consider $\mathcal{P}$ as an input *to itself*.

In case you may think this is invalid, it happens all the time. A text editor may open itself up, allowing you to look at its binary code. The compiler for `C` was itself written in `C` and may be called on to compile itself. An *emulator* opens machine code intended for another machine and simulates that machine.

From the encoding $\langle \mathcal{P}, \mathcal{P} \rangle$ we construct another Turing Machine, $\mathcal{Q}$ as follows:

$$\mathcal{Q}(\langle \mathcal{P} \rangle) = \begin{cases} \text{halts if} & \mathcal{H}(\langle \mathcal{P}, \mathcal{P} \rangle) = 0 \\ \text{does not halt if} & \mathcal{H}(\langle \mathcal{P}, \mathcal{P} \rangle) = 1 \end{cases}$$

Halting Problem Proof II

Now that $\mathcal{Q}$ is constructed, we can run $\mathcal{Q}$ on itself:

$$\mathcal{Q}(\langle \mathcal{Q} \rangle) = \begin{cases} \text{halts if} & \mathcal{H}(\langle \mathcal{Q}, \mathcal{Q} \rangle) = 0 \\ \text{does not halt if} & \mathcal{H}(\langle \mathcal{Q}, \mathcal{Q} \rangle) = 1 \end{cases}$$

Which is a contradiction because $\mathcal{Q}(\langle \mathcal{Q} \rangle)$ will halt if and only if $\mathcal{Q}(\langle \mathcal{Q} \rangle)$ doesn't halt and vice versa.

Therefore, no such $\mathcal{H}$ can exist.

Other Undecidable Problems

Many other problems, some of even practical interest have been shown to be undecidable. This means that no matter how hard you try, you can *never* solve these problems with *any* algorithm.

- Hilbert's 10th problem: Given a multivariate polynomial, does it have *integral roots*?
- Post's Correspondence Problem: Given a set of "dominos" in which the top has a finite string and the bottom has another finite string, can you produce a sequence of dominos that is a *match*—where the top sequence is the same as the bottom?
- Rice's Theorem: In general, *given* a Turing Machine, $\mathcal{M}$, answering any question about any non-trivial property of the language which it defines, $L(\mathcal{M})$ is undecidable.

Reductions

Showing that a problem is undecidable is relatively easy—you simply show a *reduction* to the halting problem. That is, given a problem $\mathcal{P}$ that you wish to show undecidable, you proceed by contradiction:

1. Assume that $\mathcal{P}$ is decidable by a Turing Machine $\mathcal{M}$.
2. Construct a machine $\mathcal{R}$ that uses $\mathcal{M}$ to decide the Halting Problem.
3. Contradiction – such a machine $\mathcal{M}$ cannot exist.

Halting Problem

Intuitive example: we can categorize all statements into two sets: lies and truths. How then can we categorize the sentence,

I am lying

The key to this seeming paradox is *self-reference*. This is where we get the terms *recursive* and *recursively enumerable*.

Complexity Classes

Now that we have a concrete model to work from: Problems as languages and Algorithms as Turing Machines, we can further delineate complexity classes *within* R (all decidable problems) by considering Turing Machines with respect to *resource bounds*.

In the computation of a Turing Machine $\mathcal{M}$, the amount of memory $\mathcal{M}$ uses can be quantified by how many *tape cells* are required in the computation of an input x. The amount of *time* $\mathcal{M}$ uses can be quantified by the number of transitions $\mathcal{M}$ makes in the computation of x.

Complexity Classes

Of course, just as before, we are interested in how much time and memory are used as a *function* of the input size. In this case,

$$T(|x|)$$

and

$$M(|x|)$$

respectively where $x \in \Sigma^*$. Again, the restriction to decisional versions of problems is perfectly fine—we could just consider languages and Turing Machines themselves.

Complexity Class P

Definition

The complexity class P consists of all languages that are decidable by a Turing Machine running in polynomial time with respect to the input $|x|$. Alternatively, P is the class of all decision problems that are solvable by a polynomial time running algorithm.

Non-Determinism I

A *nondeterministic* algorithm (or Turing Machine) is an algorithm that works in two stages:

1. It *guesses* a solution to a given instance of a problem. This set of data corresponding to an instance of a decision problem is called a *certificate*.
2. It *verifies* whether or not the guessed solution is valid or not.
3. It accepts if the certificate is a valid *witness*.

Non-Determinism II

As an example, recall the HamiltonianCycle problem.

A nondeterministic algorithm would guess a solution by forming a permutation π of each of the vertices.

It would then verify that $(v_i, v_{i+1}) \in E$ for $0 \leq i \leq n-1$.

It *accepts* if π is a Hamiltonian Cycle, otherwise it rejects.

Non-Determinism III

An instance is in the language if *there exists* a computation path that accepts.

Therein lies the *nondeterminism* – such an algorithm does *not* determine an actual answer.

Alternatively, a nondeterministic algorithm solves a decision problem if and only if for every *yes* instance of the problem it returns *yes* on *some* execution.

Non-Determinism IV

This is the same as saying that *there exists* a certificate for an instance.

A certificate can be used as a *proof* that a given instance is a *yes* instance of a decision problem. In such a case, we say that the certificate is *valid*.

If a nondeterministic algorithm produces an *invalid* certificate, it does NOT necessarily man that the given instance is a *no* instance.

NP

We can now define the class NP.

Definition

NP ("Nondeterministic Polynomial Time") is the class of all languages (problems) that can be decided by a Nondeterministic Turing Machine running in polynomial time with respect to the size of the input $|x|$.

That is, each stage, guessing and verification, can be done in polynomial time. HamiltonianCycle $\in$ NP since a random permutation can be generated in $\mathcal{O}(n)$ time and the verification process can be done in $\mathcal{O}(n^2)$ time.

P versus NP I

It is not hard to see that

$$\mathsf{P} \subseteq \mathsf{NP}$$

since any problem that can be deterministically solved in polynomial time can certainly be solved in nondetermistic polynomial time.

The most famous unanswered question so far then is

$$\mathsf{P} \stackrel{?}{=} \mathsf{NP}$$

P versus NP II

In fact, this is the most important open question in computer science today. The Clay Mathematics Institute has designated it as one of their *Millennium Problems*[1], a collection of 7 open scientific questions deemed important for the new millennium.

Who so ever is able to definitively solve this problem (or any of the other 6) and survives the review of jealous and skeptical peers will be the proud recipient of $1,000,000.

[1] http://www.claymath.org/millennium/

The Million Dollar Question

The very question itself is at the heart of computer science studies.

If the answer is yes (*very unlikely*), then *every* problem in NP could be solved in polynomial time. If the answer is no, then the hardest problems in NP could *never* be solved by a polynomial time algorithm. Such problems will forever remain *intractable*.

To understand this more fully, we need to explore the notion of NP-Completeness.

Polynomial Time Reductions

Just as we had to show a *reduction* from one problem to the halting problem to show that it was undecidable, so to can we make *polynomial time reductions* between decidable problems.

Definition

A decision problem $\mathcal{P}_1$ is said to be *polynomial time reducible* to a decision problem $\mathcal{P}_2$ if there exists a function f such that

- f maps all *yes* instances of $\mathcal{P}_1$ to all yes instances of $\mathcal{P}_2$. *no* instances likewise.
- f is computable by a polynomial time algorithm

In such a case we write

$$\mathcal{P}_1 \leq_{\mathsf{P}} \mathcal{P}_2$$

NP-Completeness

In general, an easy problem can always be trivially "reduced" to a harder problem. Conversely, it is not possible to reduce a hard problem to an easy one using a polynomial time reduction.

Definition

A problem $\mathcal{P}$ is said to be NP-*Complete* if

1. $\mathcal{P} \in$ NP and
2. For every problem $\mathcal{P}' \in$ NP, $\mathcal{P}' \leq_p \mathcal{P}$

Intuitively, NP-Complete problems are the *hardest* (most difficult computationally speaking) problems in NP (of course there are *provably* harder problems in classes such as EXP.

The First Reduction

The notion of NP completeness doesn't help us much by itself—given a problem $\mathcal{P}$ that we wish to show is NP-Complete, how does one prove that *every* problem in NP reduces to $\mathcal{P}$?

The key is to realize that polynomial time reductions are *transitive*:

$$\mathcal{P}_1 \leq_p \mathcal{P}_2 \leq_p \mathcal{P}_3 \Rightarrow \mathcal{P}_1 \leq_p \mathcal{P}_3$$

Thus, we need only show a reduction *from* a known NP-Complete problem to $\mathcal{P}$ to show that $\mathcal{P} \in \mathrm{NPC}$.

In 1971, Stephen Cook independently defined the notions of NP and NP-Completeness showing the first NP-Complete problem ever.

Notation

Recall the following notations:

- A *literal* is a boolean variable that can be set to 0 or 1
- $\vee$ denotes the logical *or* of 2 boolean variables
- $\wedge$ denotes the logical *and* of 2 boolean variables
- $\neg$ denotes the *negation* of a boolean variable
- A *clause* is the is the logical disjunction (*or*-ing) of a set of boolean variables. Ex: $(x_1 \vee \neg x_2 \vee x_5)$
- The *conjunction* of a collection of clauses is the logical *and* of all their values. The value is true only if *every* clause is true.

Satisfiability

SATISFIABILITY or simply just SAT is the following problem:

Problem (Satisfiability)

Given: *a set of boolean variables,* $\mathcal{V} = \{x_1, x_2, \ldots x_n\}$ *and a set of clauses,* $\mathcal{C} = \{C_1, C_2, \ldots C_m\}$.

Question: *Does there exist a satisfying assignment of boolean values to each literal* x_i, $1 \leq i \leq n$ *such that*

$$\bigwedge_{i=1}^{m} C_i = C_1 \wedge C_2 \wedge \ldots \wedge C_m = 1$$

Example

Let $n = 4$ and consider the following conjunction:

$$\mathcal{C} = (x_1 \vee x_2 \vee \neg x_4) \wedge (\neg x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_3 \vee \neg x_4)$$

This conjunction *is* satisfiable and is therefore a *yes* instance of SAT since if we set $x_1 = x_4 = 0$ and $x_2 = x_3 = 1$, $\mathcal{C} = 1$.

Let $n = 3$ and consider the following conjunction:

$$\begin{aligned} \mathcal{C} \quad = \quad & (x_1 \vee x_2) \wedge (\neg x_1 \vee \neg x_2) \wedge \\ & (x_1 \vee x_3) \vee (\neg x_1 \vee \neg x_3) \wedge \\ & (x_2 \vee x_3) \wedge (\neg x_2 \vee \neg x_3) \end{aligned}$$

This conjunction is *not* satisfiable since none of the $2^n = 8$ possible boolean assignments will ever make $\mathcal{C} = 1$.

Satisfiability

This perfectly illustrates the intuition behind the NP class. *Yes* instances are easily solved if we are lucky enough to guess a satisfying assignment. *No* instances require an exhaustive search of all possible assignments.

Theorem (Cook, 1971)

The problem SAT *is* NP*-Complete.*

NP-Complete Problems

First, let's take a look at a short list of standard NP-Complete problems. Of course, literally hundreds of problems have been shown to be NP-Complete since Cook showed the first reduction.

- 3-CNF – A more restrictive version of SAT where each clause is a disjunction of exactly 3 literals. CNF stands for *Conjunctive Normal Form*. Note that $\text{2-CNF} \in \mathrm{P}$.
- HAMILTONIANCYCLE – Determine if a given undirected graph G contains a cycle which passes through every vertex exactly once.
- TRAVELINGSALESMAN – Find the least weighted cycle in a graph G that visits each vertex exactly once.

NP-Complete Problems

- SubsetSum – Given a collection of integers, can you form a subset $\mathcal{S}$ such that the sum of all items in $\mathcal{S}$ is exactly p.
- GraphColoring – For a given graph G, find its chromatic number $\chi(G)$ which is the smallest number of colors that are required to color the vertices of G so that no two adjacent vertices have the same color.

Some good resources on the subject can be found in:

- *Computers and Intractability – A Guide to the Theory of NP Completeness* 1979
- http://www.csc.liv.ac.uk/~ped/teachadmin/COMP202/annotated_np.html – An annotated list of about 90 NP-Complete Problems.

Showing a Polynomial Reduction

There are 5 basic steps to show that a given problem $\mathcal{P}$ is NP-Complete.

1. Prove that $\mathcal{P} \in$ NP by giving an algorithm that guesses a certificate and an algorithm that verifies a solution in polynomial time.
2. Select a known NP-Complete problem $\mathcal{P}'$ that we will reduce to $\mathcal{P}$ ($\mathcal{P}' \leq_{\mathrm{P}} \mathcal{P}$)
3. Give an algorithm that computes $f : \mathcal{P}'_{\text{yes}} \mapsto \mathcal{P}_{\text{yes}}$ for every instance $x \in \{0,1\}^*$.
4. *Prove* that f satisfies $x \in \mathcal{P}'$ if and only if $f(x) \in \mathcal{P}$
5. Prove that the algorithm in step 3 runs in polynomial time

Clique problem

A *clique* in an undirected graph $G = (V, E)$ is a complete induced subgraph $G' = (V', E'), V' \subseteq V, E' \subseteq E$. That is, it is a subset of vertices V' of G where every vertex in V' is connected to each other.

Problem (Clique)

Given: *An undirected graph* $G = (V, E)$
Question: *Does there exist a clique of size* k*?*

In terms of languages we define

$$\textsc{Clique} = \{\langle G, k\rangle \mid G \text{ is a graph with a clique of size } k\}$$

Clique Reduction

We want to prove that Clique is NP-Complete. To do this we will go by our 5 step process.

1. Clique $\in$ NP. We can randomly select k vertices from a given graph's vertex set V in $\mathcal{O}(|V|)$ time. Further, we can check if, for each pair of vertices $v, v' \in V'$ if $(v, v') \in E$ in $\mathcal{O}(|V|^2)$ time.
2. We select the 3-CNF-SAT problem, a known NP-Complete problem for our reduction: 3-CNF-SAT $\leq_P$ Clique.

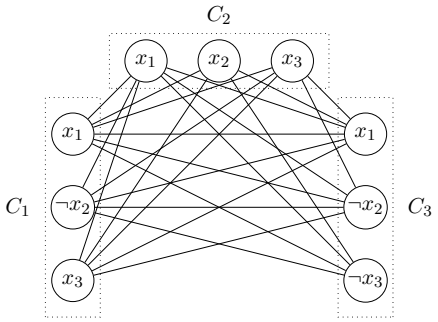
Clique Reduction

3. We define the following function. Let $\phi = C_1 \wedge \ldots \wedge C_k$ be a 3-CNF formula. We will construct a graph G that has a clique of size k if and only if ϕ is satisfiable. For each clause $C_i = (x_1^i \vee x_2^i \vee x_3^i)$ we define vertices $v_1^i, v_2^i, v_3^i \in V$. Edges are defined such that $(v_\alpha^i, v_\beta^j) \in E$ if *both* of the following hold:
   1. If the vertices are in different clauses, i.e. $i \neq j$
   2. Their corresponding literals are *consistent*: v_α^i is not the negation of v_β^j

Example

$$\mathcal{C} = (x_1 \vee \neg x_2 \vee x_3) \wedge (x_1 \vee \neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3)$$

Clique Reduction



Clique Reduction

4. We need to show that *yes* instances of the 3-CNF function ϕ are preserved with this function. That is we want to show that ϕ is satisfiable if and only if $f(\phi) = G$ has a clique of size k. ($\Rightarrow$) : Suppose that ϕ has a satisfying assignment. This implies that each clause C_i contains at least one true literal. Remember that each literal corresponds to some vertex in G. Choosing a true literal from each clause yields a set V' of size k. To see that V' is a clique we look at our two requirements from before: v_α^i and v_β^j are consistent and both are true, thus $(v_\alpha^i, v_\beta^j) \in E$.

Clique Reduction

($\Leftarrow$): Suppose that G has a clique of size k. No edges in G connect vertices in the same triple corresponding to a clause so V' contains exactly on vertex per triple. Without fear of causing inconsistencies, we can assign a 1 to each literal corresponding to some vertex in each triple thus each clause is satisfied and so ϕ is satisfied.

5. The computation of the function described in step 3 is clearly polynomial. The input size is something like $\mathcal{O}(nk)$ so building G takes at most $\mathcal{O}(2n^2k)$ time.

Independent Set Problem

Definition

An *independent set* of an undirected graph $G = (V, E)$ is a subset of vertices $V' \subseteq V$ such that for any two vertices $v, v' \in V'$, $(v, v') \notin E$

Problem (IndependentSet)

Given *an undirected graph* $G = (V, E)$
Question: *Does there exist an independent set of size* k*?*

Independent Set Reduction

Again, we'll follow our 5-step process to show that IndependentSet is NP-Complete.

1. Just like the Clique problem, we can formulate a certificate and verify it in polynomial time.
2. We will make the following reduction: Clique $\leq_p$ IndependentSet
3. The function is as follows $f : G \to \overline{G}$.
4. We want to show that a clique of size k in G is equivalent to an independent set of size k in $\overline{G}$.
5. The computation of f is certainly polynomial; $\mathcal{O}(|V|^2)$

Traveling Salesman Reduction

Exercise

Show that the TravelingSalesman *problem is* NP*-Complete by showing a reduction from* HamiltonianCircuit.

Beyond P and NP

There are, of course, *many* more complexity classes other than those we've looked at here. There are space (memory) classes, nondeterministic space classes, quantum classes, probabilistic classes etc.

For an almost complete listing and more recent results in complexity classes, check out the Complexity Zoo:
http://www.complexityzoo.com/

Beyond P and NP

We conclude this discussion by giving an overall view of the complexity classes we've seen. A few quick notes, first however:

- coRE is the compliment class of RE, that is it consists of all decision problems for which *no* instances can be verified by a Turing Machine in a finite amount of time. *yes* instances are not guaranteed to halt.
- coNP is the compliment class of NP. Rather than producing certificates, acceptance is defined as being able to produce a *disqualifier* (i.e. if some computation path produces a *no* answer, we accept. This is still a nondeterministic class. A good example: TAUTOLOGY.

Beyond P and NP

- NP $\cap$ coNP is the intersection of NP and coNP, P is contained in this class as is
- NPI (NP intermediate).
- If P $\neq$ NP, then NPI $\neq \emptyset$ is all we know.
- To date no one has shown that an actual problem is in NPI. The leading candidate is GRAPHISOMORPHISM.

One big note here, the following picture merely depicts the prevailing opinion of the Complexity Theory community. Again, no one has proven that P $\neq$ NP yet. Which of the separations in the following diagram *have been* proven though?

Complexity Hierarchy