CS:4330 Theory of Computation

Spring 2018

Computability Theory

P versus NP and NP-Completeness

Haniel Barbosa

The University of Iowa

Readings for this lecture

Chapter 7 of [Sipser 1996], 3rd edition. Section 7.4.

The P versus NP question

- ▷ **P**: the class of languages for which membership can be *decided* quickly

- ▷ **NP**: the class of languages for which membership can be *verified* quickly

So, is $\mathbf{P} = \mathbf{NP}$?

- ▷ This is one the greatest unsolved problems in theoretical computer science and contemporary mathematics.

Observations

- If **P** is equal to **NP** then any polynomially verifiable problem would be polynomially decidable
- Most researchers believe that $\mathbf{P} \neq \mathbf{NP}$ because people have invested enormous effort to find polynomial time algorithms for some problems in **NP** without success
- A proof that $\mathbf{P} \neq \mathbf{NP}$ would mean that no fast algorithm exists to replace brute-force search for some problems. This may be beyond scientific research
- Best method known for solving problems in **NP** deterministically is based on deterministic simulation of NTM, which is exponential, i.e.

$$\mathbf{NP} \subseteq \mathrm{EXPTIME} = \bigcup_k \mathrm{TIME}(2^{n^k})$$

But we don't know whether **NP** is contained in a smaller deterministic time complexity class

NP-Completeness

- Certain problems in **NP** have their individual complexity related to that of the entire class (Stephen Cook and Leonid Levin, 1970s)

- If a polynomial time algorithm exists for any of these problems, all problems in **NP** would be polynomial time solvable

- These problems are called NP-complete and the phenomenon of NP-completeness is important for both theoretical and practical reasons

Theoretical importance

- ▷ Research trying to show that $\mathbf{P} \neq \mathbf{NP}$ may focus on an NP-complete problem
- ▷ If any problem in **NP** requires more than polynomial time, an NP-complete one also does
- ▷ Research trying to show that $\mathbf{P} = \mathbf{NP}$ only needs to find a polynomial time algorithm for *one* NP-complete problem

Practical importance

- ▷ The phenomenon of NP-completeness may prevent wasting time searching for a non-existent polynomial time algorithm to solve a particular problem

- ▷ Even though we may not have the necessary mathematics to prove that the problem is unsolvable in polynomial time, showing that it is NP-complete is enough evidence, since we believe that $\mathbf{P} \neq \mathbf{NP}$

Example NP-complete problem

- A Boolean formula is an expression involving Boolean variables and operations. For example

$$\varphi = (\neg x \wedge y) \vee (x \wedge \neg z) \text{ is a Boolean formula}$$

- A Boolean formula is satisfiable if some assignments of 1 and 0 (*true* and *false*) to its variables makes the formula evaluate to 1. For example,

$$x = 0, y = 1, z = 0 \text{ makes } \varphi \text{ evaluate to } 1$$

- The satisfiability problem (SAT) is to test whether a Boolean formula is satisfiable, i.e.

$$\text{SAT} = \{\varphi \mid \varphi \text{ is a satisfiable Boolean formula}\}$$

This is the equivalent for propositional logic of the satisfiability problem for first-order logic we have seen before. Here models have as *universe* the set $\{1, 0\}$ and only Boolean variables require an interpretation function.

Why is SAT fundamental?

Theorem

$\mathrm{SAT} \in \mathbf{P}$ *if and only if* $\mathbf{P} = \mathbf{NP}$

- ▷ This theorem links the complexity of SAT to the complexities of all problems in **NP**

- ▷ To prove this result we develop the polynomial time reducibility method

Polynomial time reducibility

- ▷ When problem A reduces to problem B a solution to B can also solve A
- ▷ Polynomial time reducibility is a reducibility method that takes the efficiency of computation into account
- ▷ When problem A is efficiently reducible to problem B, an efficient solution to B can be used to solve A efficiently

Definition (Polynomial time computability)

A function $f : \Sigma^* \to \Sigma^*$ is a polynomial time computable function if some polynomial time TM M exists that halts with just $f(w)$ on its tape, when started on any input w.

Definition (Polynomial time reducibility)

A language A is *polynomial time mapping reducible*, or *polynomial time reducible* to a language B, written $A \leq_p B$, if a polynomial computable function $f : \Sigma^* \to \Sigma^*$, denoted the *polynomial time reduction* of A to B, exists such that for every $w \in \Sigma^*$, $w \in A \Leftrightarrow f(x) \in B$

Observations

- ▷ A polynomial time reduction of A to B provides an efficient way of converting membership testing in A to membership testing in B
- ▷ To test whether $w \in A$ we use the reduction f to map w into $f(w)$ and test $f(w) \in B$

Theorem

If $A \leq_p B$ *and* $B \in \mathbf{P}$ *then* $A \in \mathbf{P}$

Proof idea: polynomial time reducibility

Let M be a polynomial time algorithm deciding B and f be a polynomial time reduction from A to B. Then the polynomial time decider N to A is
$N =$"On input w:

1. Compute $f(w)$
2. Run M on input $f(w)$ and output whatever M outputs."

NP-Completeness

Definition

A language B is *NP-complete* if it satisfies two conditions:

1. $B \in \mathbf{NP}$
2. Every $A \in \mathbf{NP}$ is polynomial time reducible to B

Theorem

If B is NP-complete and $B \in \mathbf{P}$ then $\mathbf{P} = \mathbf{NP}$

Theorem

If B is NP-complete, $B \leq_p C$ and $C \in \mathbf{NP}$, then C is NP-complete.

Proof idea

Since $C \in \mathbf{NP}$ we only need to show that every $A \in \mathbf{NP}$ is polynomial time reducible to C. Since every NP problem A is polynomial time reducible to B and $B \leq_p C$, then $A \leq_p C$ since we can first reduce A to B and then reduce the result into C.

SAT is fundamental

Theorem (Cook-Levin Theorem)

SAT is NP-complete.

Proof idea

- ▷ Show that $\mathrm{SAT} \in \mathbf{NP}$, which is fairly easy
- ▷ Show that any language $A \in \mathbf{NP}$ is polynomial time reducible to SAT
- ▷ The reduction of A takes an NTM N and a string w and produces a Boolean formula φ_w that simulates the NTM N that decides A operating on w
- ▷ If N accepts, φ_w has a satisfying assignment that corresponds to that computation; if N doesn't accept, no assignment satisfies φ_w
- ▷ Therefore $w \in A$ iff φ_w is satisfiable

Proving SAT is NP-complete

1. $\mathrm{SAT} \in \mathbf{NP}$ since a nondeterministic polynomial time machine can guess an assignment to the variables of a given formula φ and accept if the assignment satisfies φ

2. Let $A \in \mathbf{NP}$: show that A is polynomial time reducible to SAT. For an NTM N that decides A in $\mathcal{O}(n^k)$ time for some constant k, construct a formula φ that simulates N

   The construction of φ is based on organizing the computation performed by N into an $n^k \times n^k$ tableau

   - Rows are the configurations of a branch of the computation of N on input w
   - Each configuration starts and ends with #
   - A tableau is *accepting* if any of its rows is an accepting configuration

Accepting Tableau(N,w)

- ▷ Every accepting tableau for N and w correspond to an accepting computation branch of N on w
- ▷ Problem of determining whether N accepts w is equivalent to the problem of determining whether an accepting tableau for N on w exists

Polynomial time $f : A \to \text{SAT}$

- ▷ On input w, f produces φ_w
- ▷ *Variables* of φ_w: Let $N = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$ and $C = Q \cup \Gamma \cup \{\#\}$. For each $1 \leq i,j \leq n^k$ and $s \in C$ we have a variable $x_{i,j,s}$
- ▷ *Cells*: each of the $(n^k)^2$ entries of a tableau is called a *cell*. The cell in row i and column j is called $cell[i,j]$ and contains a symbol from C.
- ▷ The contents of cells are represented with the variables of φ
- ▷ For every state $s \in C$, $x_{i,j,s} = 1$ if $cell[i,j] = s$
- ▷ Formula $\varphi_w = \varphi_{cell} \wedge \varphi_{start} \wedge \varphi_{move} \wedge \varphi_{accept}$ corresponds to Tableau(N,w)

Tableau(N, w) must be well formed

We first ensure that the tableau is well formed, i.e. it contains exactly one symbol per cell. Thus φ_w must guarantee that exactly one variable is true for each cell:

1. at least one variable that is associated with a cell is true, represented by

$$\bigvee_{s \in C} x_{i,j,s}$$

2. no more than one variable is true for each cell, i.e. for each pair of variables at least one is false, which is represented by

$$\bigwedge_{s,t \in C, s \neq t} (\neg x_{i,j,s} \vee \neg x_{i,j,t})$$

A satisfying assignment which makes true one variable for each cell is specified by

$$\varphi_{cell} = \bigwedge_{1 \leq i,j \leq n^k} \left[\bigvee_{s \in C} x_{i,j,s} \wedge \left(\bigwedge_{s,t \in C, s \neq t} (\neg x_{i,j,s} \vee \neg x_{i,j,t}) \right) \right]$$

$\text{Tableau}(N, w)$ must be an accepting tableau

The formulas $\varphi_{start}, \varphi_{move}, \varphi_{accept}$ guarantee that the symbols in the tableau actually correspond to an accepting tableau

- ▷ The first row of the tableau must be the starting configuration of N on w:

$$\varphi_{start} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge \cdots \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,\sqcup} \wedge \cdots \wedge x_{1,n^k-1,\sqcup} \wedge x_{1,n^k,\#}$$

- ▷ An accepting configuration must occur in the tableau, i.e. q_a must be in one of the cells:

$$\varphi_{accept} = \bigvee_{1 \leq i,j \leq n^k} x_{i,j,q_a}$$

- ▷ Each row of the tableau must correspond to a configuration that legally follows the preceding row's configuration according to N's transition rules. This is guaranteed by φ_{move}

Legal windows

A 2×3 window of cells is legal if that window does not violate the actions specified by N's transition function. In other words, a window is legal if it might appear when one configuration correctly follows another.

Consider the transitions:

$$\delta(q_1, \mathrm{a}) = \{(q_1, \mathrm{b}, R)\}, \delta(q_1, \mathrm{b}) = \{(q_2, \mathrm{c}, L), (q_2, \mathrm{a}, R)\}$$

Examples of legal windows for this machine are:

(a)

a	q_1	b
q_2	a	c

(b)

a	q_1	b
a	a	q_2

(c)

a	a	q_1
a	a	b

(d)

#	b	a
#	b	a

(e)

a	b	a
a	b	q_2

(f)

b	b	b
c	b	b

Legal windows

- ▷ Windows (a) and (b) are legal because the transition allows N to move this way
- ▷ Window (c) is legal because with q_1 appearing on the right side of the top row, we don't know what symbol the head is over
- ▷ Window (d) is legal because top and bottom are identical, which could happen if the head weren't adjacent to the location of the window
- ▷ Window (e) is legal because state q_1 reading a b might have been immediately to the right of the top row and would have moved to the left
- ▷ Window (f) is legal because state q_1 might have been immediately to the left of the top row changing b to c and moving left

Illegal windows

The following windows aren't legal for machine N:

(a)

a	b	a
a	a	a

(b)

a	q_1	b
q_1	a	a

(c)

b	q_1	b
q_2	b	q_2

- (a) is illegal because the central symbol cannot be changed without an adjacent state
- (b) is illegal because the transition function states that b gets changed to c, not to a
- (c) is illegal because two states appear in the bottom row

Construction of φ_{move}

Lemma

If the top row of the tableau is the start configuration and every window is legal then each row is a configuration that legally follows the configuration represented by the preceding row.

- ▷ Each window contains six cells, which may be set in a fixed number of ways to yield a legal window, according to N's transition function
- ▷ φ_{move} says that the setting of these six cells is legal by

$$\varphi_{move} = \bigwedge_{1 \leq i,j \leq n^k} (window[i,j] \text{ is legal})$$

  in which a $window[i,j]$ has a $cell[i,j]$ as the upper central position.
- ▷ A window being legal is computed in terms of the contents of its cells, as represented by

$$\bigvee_{a_1, \ldots, a_6} (x_{i,j-1,a_1} \wedge x_{i,j,a_2} \wedge x_{i,j+1,a_3} \wedge x_{i+1,j-1,a_4} \wedge x_{i+1,j,a_5} \wedge x_{i+1,j+1,a_6})$$

is a legal window

Complexity of the reduction

- ▷ Tableau is $n^k \times n^k$ and thus contains n^{2k} cells; each has $|C| = l$ variables associated with it where l depends on N. Hence the total number of variables $\mathcal{O}(n^{2k})$
- ▷ Estimating the size of the four components of φ_w:
  - φ_{cell} contains a fixed fragment of φ_w for each cell of the tableau, so its size is $\mathcal{O}(n^{2k})$
  - φ_{start} has a fixed fragment for each cell in the top row, so has size $\mathcal{O}(n^k)$
  - φ_{move} and φ_{accept} contain a fixed fragment for each cell of the tableau, so their size is $\mathcal{O}(n^{2k})$
  - Therefore the total size of φ_w is polynomial in n
- ▷ Each component of φ_w can be produced in polynomial time. Therefore we can construct a polynomial time reduction from N into φ_w

In conclusion

- ▷ SAT is an NP-complete problem since it is NP and we can polynomially reduce any arbitrary NP problem into it
- ▷ Showing that other problems are NP-complete is generally much simpler than proving the above theorem, as it suffices to show a polynomial reduction to known NP-complete problem, such as SAT
- ▷ SAT was the first problem to be proven to be NP-complete
- ▷ All problems in **NP** are at most as difficult as SAT
- ▷ Providing a polynomial algorithm to SAT solves the **P** versus **NP** dilemma

What Knuth thinks

> *At the present time very few people believe that* ***P=NP****. In other words, almost everybody who has studied the subject thinks that satisfiability cannot be decided in polynomial time. The author of this book, however, suspects that* $N^{\mathscr{O}(1)}$-steps *algorithms do exist, yet that they're unknowable. Almost all polynomial time algorithms are so complicated that they lie beyond human comprehension, and could never be programmed for an actual computer in the real world.* ***Existence is different from embodiment****.*

Donald Knuth

The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability