Tractable Problems

We would like some convention for distinguishing tractable from intractable problems. A problem is said to be **tractable** if an algorithm exists to solve it with polynomial time complexity: \( O(p(n)) \).

- It is said to be **intractable** if the best known algorithm requires exponential time.

Examples:
- Sorting: \( O(n^2) \)
- Convex Hull: \( O(n^2) \)
- Single source shortest path: \( O(n^2) \)
- All pairs shortest path: \( O(n^3) \)
- Matrix multiplication: \( O(n^3) \)

Log-polynomial is \( O(n \log n) \)

Like any simple rule of thumb for categorizing, in some cases the distinction between polynomial and exponential could break down. For example, one can argue that, for practical problems, \( 1.01^n \) is preferable to \( n^{25} \). But the reality is that very few polynomial-time algorithms have high degree, and exponential-time algorithms nearly always have a constant of 2 or greater. Nearly all algorithms are either low-degree polynomials or “real” exponentials, with very little in between.

Tractable Problems (cont)

The technique we will use to classify one group of algorithms is based on two concepts:
- A special kind of reduction.
- Nondeterminism.

Decision Problems

(I, S) such that S(X) is always either “yes” or “no.”

- Usually formulated as a question.

Example:
- Instance: A weighted graph \( G = (V, E) \), two vertices \( s \) and \( t \), and an integer \( K \).

- Question: Is there a path from \( s \) to \( t \) of length \( \leq K \)? In this example, the answer is “yes.”
Decision Problems (cont)

Can also be formulated as a language recognition problem:
- Let $L$ be the subset of $I$ consisting of instances whose answer is “yes.” Can we recognize $L$?

The class of tractable problems $P$ is the class of languages or decision problems recognizable in polynomial time.

Polynomial Reducibility

Reduction of one language to another language.

Let $L_1 \subseteq I_1$ and $L_2 \subseteq I_2$ be languages. $L_1$ is polynomially reducible to $L_2$ if there exists a transformation $f : I_1 \rightarrow I_2$, computable in polynomial time, such that $f(x) \in L_2$ if and only if $x \in L_1$.

We write: $L_1 \leq_P L_2$ or $L_1 \leq L_2$.

Examples

- CLIQUE $\leq_P$ INDEPENDENT SET.
- An instance $I$ of CLIQUE is a graph $G = (V, E)$ and an integer $K$.
- The instance $I' = f(I)$ of INDEPENDENT SET is the graph $G' = (V, E')$ and the integer $K'$, were an edge $(u, v) \in E'$ iff $(u, v) \notin E$.
- $f$ is computable in polynomial time.

Transformation Example

- $G$ has a clique of size $\geq K$ iff $G'$ has an independent set of size $\geq K$.
- Therefore, CLIQUE $\leq_P$ INDEPENDENT SET.
- IMPORTANT WARNING: The reduction does not solve either INDEPENDENT SET or CLIQUE, it merely transforms one into the other.

Following our graph example: It is possible to translate from a graph to a string representation, and to define a subset of such strings as corresponding to graphs with a path from $s$ to $t$. This subset defines a language to “recognize.”

Or one decision problem to another.

Specialized case of reduction from Chapter 10.

Need a graph here.

If nodes in $G'$ are independent, then no connections. Thus, in $G$ they all connect.
Nondeterminism

Nondeterminism allows an algorithm to make an arbitrary choice among a finite number of possibilities.

Implemented by the "nd-choice" primitive:

\[
\text{nd-choice}(ch_1, ch_2, \ldots, ch_j)
\]
returns one of the choices \( ch_1, ch_2, \ldots \) arbitrarily.

Nondeterministic algorithms can be thought of as "correctly guessing" (choosing nondeterministically) a solution.

Alternatively, nondeterministic algorithms can be thought of as running on super-parallel machines that make all choices simultaneously and then reports the "right" solution.

Nondeterministic CLIQUE Algorithm

procedure nd-CLIQUE(Graph G, int K) {
    VertexSet S = EMPTY; int size = 0;
    for (v in G.V)
        if (nd-choice(YES, NO) == YES) then {
            S = union(S, v);
            size = size + 1;
        }
    if (size < K) then
        REJECT; // S is too small
    for (u in S)
        for (v in S)
            if ((u <> v) && ((u, v) not in E))
                REJECT; // S is missing an edge
    ACCEPT;
}

Nondeterministic Acceptance

\((G, K)\) is in the "language" CLIQUE iff there exists a sequence of nd-choice guesses that causes nd-CLIQUE to accept.

Definition of acceptance by a nondeterministic algorithm:

- An instance is accepted iff there exists a sequence of nondeterministic choices that causes the algorithm to accept.
- An unrealistic model of computation.
  - There are an exponential number of possible choices, but only one must accept for the instance to be accepted.
- Nondeterminism is a useful concept
  - It provides insight into the nature of certain hard problems.

Class \(\mathcal{NP}\)

- The class of languages accepted by a nondeterministic algorithm in polynomial time is called \(\mathcal{NP}\).
- There are an exponential number of different executions of nd-CLIQUE on a single instance, but any one execution requires only polynomial time in the size of that instance.
- Time complexity of nondeterministic algorithm is greatest amount of time required by any one of its executions.

Note that Towers of Hanoi is not in \(\mathcal{NP}\).
Class $\mathcal{NP}$ (cont)

Alternative Interpretation:
- $\mathcal{NP}$ is the class of algorithms that — never mind how we got the answer — can check if the answer is correct in polynomial time.
- If you cannot verify an answer in polynomial time, you cannot hope to find the right answer in polynomial time!

How to Get Famous

Clearly, $\mathcal{P} \subset \mathcal{NP}$.

Extra Credit Problem:
- Prove or disprove: $\mathcal{P} = \mathcal{NP}$.

This is important because there are many natural decision problems in $\mathcal{NP}$ for which no $\mathcal{P}$ (tractable) algorithm is known.

$\mathcal{NP}$-completeness

A theory based on identifying problems that are as hard as any problems in $\mathcal{NP}$.

The next best thing to knowing whether $\mathcal{P} = \mathcal{NP}$ or not.

A decision problem $A$ is $\mathcal{NP}$-hard if every problem in $\mathcal{NP}$ is polynomially reducible to $A$, that is, for all $B \in \mathcal{NP}$, $B \leq_p A$.

A decision problem $A$ is $\mathcal{NP}$-complete if $A \in \mathcal{NP}$ and $A$ is $\mathcal{NP}$-hard.

Satisfiability

Let $E$ be a Boolean expression over variables $x_1, x_2, \ldots, x_n$ in conjunctive normal form (CNF), that is, an AND of ORs.

$$E = (x_6 + x_7 + \overline{x_8} + x_{10}) \cdot (\overline{x_2} + x_3) \cdot (x_1 + \overline{x_5} + x_6).$$

A variable or its negation is called a literal. Each sum is called a clause.

SATISFIABILITY (SAT):
- Instance: A Boolean expression $E$ over variables $x_1, x_2, \ldots, x_n$ in CNF.
- Question: Is $E$ satisfiable?

Cook’s Theorem: SAT is $\mathcal{NP}$-complete.

Cook won a Turing award for this work.

This is worded a bit loosely. Specifically, we assume that we can get the answer fast enough — that is, in polynomial time non-deterministically.

SATISFIABILITY (SAT):

Question: Is $x$ satisfiable?

Instance: A Boolean expression $E$ over variables $x_1, x_2, \ldots, x_n$ in CNF.

Prove or disprove: $P \subseteq NP$.

Is there a truth assignment for the variables that makes $E$ true?

Cook won a Turing award for this work.
Proof Sketch

SAT ∈ NP:
- A non-deterministic algorithm guesses a truth assignment for \( x_1, x_2, \ldots, x_n \) and checks whether \( E \) is true in polynomial time.
- It accepts if there is a satisfying assignment for \( E \).

SAT is \( \mathcal{NP} \)-hard:
- Start with an arbitrary problem \( B \in \mathcal{NP} \).
- We know there is a polynomial-time, nondeterministic algorithm to accept \( B \).
- Cook showed how to transform an instance \( X \) of \( B \) into a Boolean expression \( E \) that is satisfiable if the algorithm for \( B \) accepts \( X \).

Implications

(1) Since SAT is \( \mathcal{NP} \)-complete, we have not defined an empty concept.

(2) If SAT ∈ \( \mathcal{P} \), then \( \mathcal{P} = \mathcal{NP} \).

(3) If \( \mathcal{P} = \mathcal{NP} \), then SAT ∈ \( \mathcal{P} \).

(4) If \( A \in \mathcal{NP} \) and \( B \in \mathcal{NP} \)-complete, then \( B \leq_p A \) implies \( A \) is \( \mathcal{NP} \)-complete.
Proof:
- Let \( C \in \mathcal{NP} \).
- Then \( C \leq_p B \) since \( B \) is \( \mathcal{NP} \)-complete.
- Since \( B \leq_p A \) and \( \leq_p \) is transitive, \( C \leq_p A \).
- Therefore, \( A \) is \( \mathcal{NP} \)-hard and, finally, \( \mathcal{NP} \)-complete.

Implications (cont)

(5) This gives a simple two-part strategy for showing a decision problem \( A \) is \( \mathcal{NP} \)-complete.
(a) Show \( A \in \mathcal{NP} \).
(b) Pick an \( \mathcal{NP} \)-complete problem \( B \) and show \( B \leq_p A \).

\( \mathcal{NP} \)-completeness Proof Template

To show that decision problem \( B \) is \( \mathcal{NP} \)-complete:
- \( B \in \mathcal{NP} \)
  - Give a polynomial time, non-deterministic algorithm that accepts \( B \).
  - Given an instance \( X \) of \( B \), guess evidence \( Y \).
  - Check whether \( Y \) is evidence that \( X \in B \). If so, accept \( X \).
- \( B \) is \( \mathcal{NP} \)-hard.
  - Choose a known \( \mathcal{NP} \)-complete problem, \( A \).
  - Describe a polynomial-time transformation \( T \) of an arbitrary instance of \( A \) to a [not necessarily arbitrary] instance of \( B \).
  - Show that \( X \in A \) if and only if \( T(X) \in B \).

The proof of this last step is usually several pages long. One approach is to develop a nondeterministic Turing Machine program to solve an arbitrary problem \( B \) in \( \mathcal{NP} \).

Implications (cont)

Proving \( A \in \mathcal{NP} \) is usually easy.

Don’t get the reduction backwards!
Choose SAT to be the known

A special case of SAT.

One might hope that 3SAT is easier than SAT.

3-SATISFIABILITY (3SAT)

Instance: A Boolean expression $E$ in CNF such that each clause contains exactly 3 literals.

Question: Is there a satisfying assignment for $E$?

3SAT is $\mathcal{NP}$-complete

(1) $3SAT \in \mathcal{NP}$.

procedure nd-3SAT($E$) { for (i = 1 to n) $x[i] = nd$-choice(TRUE, FALSE); Evaluate $E$ for the guessed truth assignment. if ($E$ evaluates to TRUE) ACCEPT; else REJECT; }

nd-3SAT is a polynomial-time nondeterministic algorithm that accepts 3SAT.

Proving 3SAT $\mathcal{NP}$-hard

Pros: Choose SAT to be the known $\mathcal{NP}$-complete problem. We need to show that SAT $\leq_{p} 3SAT$.

Let $E = C_1 \cdot C_2 \cdots C_k$ be any instance of SAT.

Strategy: Replace any clause $C_i$ that does not have exactly 3 literals with two or more clauses having exactly 3 literals.

Let $C_i = y_1 + y_2 + \cdots + y_j$ where $y_1, \cdots, y_j$ are literals.

(a) $j = 1$

- Replace ($y_i$) with

$$(y_1 + v + w) \cdot (y_1 + \overline{v} + \overline{w}) \cdot (y_1 + v + \overline{w}) \cdot (y_1 + \overline{v} + \overline{w})$$

where $v$ and $w$ are new variables.

(b) $j = 2$

- Replace ($y_1 + y_2$) with ($y_1 + y_2 + z$) \cdot ($y_1 + y_2 + \overline{z}$) where $z$ is a new variable.

(c) $j > 3$

- Replace ($y_1 + y_2 + \cdots + y_j$) with

$$(y_1 + y_2 + z_1) \cdot (y_3 + \overline{z_1} + z_2) \cdot (y_4 + \overline{z_2} + z_3) \cdots$$

$$(y_{j-2} + \overline{z_{j-3}} + \overline{z_{j-3}}) \cdot (y_{j-1} + y_j + \overline{z_{j-3}})$$

where $z_1, z_2, \cdots, z_{j-3}$ are new variables.

- After replacements made for each $C_i$, a Boolean expression $E'$ results that is an instance of 3SAT.

- The replacement clearly can be done by a polynomial-time deterministic algorithm.

Replacing ($y_i$) with ($y_1 + y_1 + y_i$) seems like a reasonable alternative. But some of the theory behind the definitions rejects clauses with duplicated literals.

SAT is the only choice that we have so far!
Proving 3SAT \( \mathcal{NP} \)-hard (cont)

(3) Show \( E \) is satisfiable iff \( E' \) is satisfiable.
- Assume \( E \) has a satisfying truth assignment.
- Then that extends to a satisfying truth assignment for cases (a) and (b).
- In case (c), assume \( y_m \) is assigned “true”.
- Then assign \( z_t, t \leq m - 2 \), true and \( z_k, t \geq m - 1 \), false.
- Then all the clauses in case (c) are satisfied.

We conclude SAT \( \leq \) 3SAT and 3SAT is \( \mathcal{NP} \)-complete.

Tree of Reductions

The reduction tree gives us a collection of 12 diverse \( \mathcal{NP} \)-complete problems. The complexity of all these problems depends on the complexity of any one:
- If any \( \mathcal{NP} \)-complete problem is tractable, then they all are.

This collection is a good place to start when attempting to show a decision problem is \( \mathcal{NP} \)-complete.

Observation: If we find a problem is \( \mathcal{NP} \)-complete, then we should do something other than try to find a \( \mathcal{P} \)-time algorithm.

Perspective

Hundreds of problems, from many fields, have been shown to be \( \mathcal{NP} \)-complete.

More on this observation later.