Reductions

A **reduction** is a transformation of one problem to another.

Purposes: To compare the difficulty of two problems.

- To use one algorithm to solve another problem (upper bound).
- To compare the relative difficulty of two problems (lower bound).

Notation: A problem is a mapping of inputs to outputs.

A definition looks as follows:

SORTING:

- Input: A sequence of integers $x_0, x_1, ..., x_{n-1}$.
- Output: A permutation $y_0, y_1, ..., y_{n-1}$ of the sequence such that $y_i \leq y_j$ whenever $i < j$.

PAIRING

PAIRING:

- Input: Two sequences of integers $X = (x_0, x_1, ..., x_{n-1})$ and $Y = (y_0, y_1, ..., y_{n-1})$.
- Output: A pairing of the elements in the two sequences such that the least value in X is paired with the least value in Y, and so on.

How can we solve this?

One algorithm:

- Sort X.
- Sort Y.
- Now, pair x_i with y_i for $0 \leq i < n$.

Terminology: We say that PAIRING is **reduced** to SORTING, since SORTING is used to solve PAIRING.

PAIRING Reduction Process

The reduction of PAIRING to SORTING requires 3 steps:

- Convert an instance of PAIRING to two instances of SORTING.
- Run SORTING (twice).
- CONVERT the output for the two instances of SORTING to an output for the original PAIRING instance.

What do we require about the transformations to make them useful?

What is the cost of this algorithm?

PAIRING Lower Bound

We have an upper bound for PAIRING equal to that of SORTING.

What is the lower bound for PAIRING?

Pretend that there is a $O(n)$ time algorithm for PAIRING.
Consider this algorithm for SORTING:

- Transform SORTING to PAIRING with X being the input sequence for SORTING, and Y a sequence containing the values 0 through $n-1$
- Run the $O(n)$ time PAIRING algorithm.
- Take the pairs output by PAIRING and use a simple binsort to order them by the second value of the pair. The first items of the pair will be the sorted list.

What is the cost of this algorithm?

What does this say about the existence of an $O(n)$ time algorithm for PAIRING?

Reduction Process

Consider any two problems for which a suitable reduction from one to the other can be found.

The first problem $P1$ takes input instance $\mathbf{I}$ and transforms that to solution $\mathbf{S}$.

The second problem $P2$ takes input instance $\mathbf{I}'$ and transforms that to solution $\mathbf{S}'$.

A **reduction** is the following three-step process:

- Transform an arbitrary instance $\mathbf{I}$ of problem $P1$ and transform it to a (possibly special) instance $\mathbf{I}'$ of $P2$.
- Apply an algorithm for $P2$ to $\mathbf{I}'$, yielding $\mathbf{S}'$.
- Transform $\mathbf{S}'$ to a solution for $P1$ ($\mathbf{S}$). Note that $\mathbf{S}$ MUST BE THE CORRECT SOLUTION for $\mathbf{I}$!

Reduction Process (Cont.)

Note that reduction is NOT an algorithm for either problem.

It does mean, given "cheap" transformations, that:

- The upper bound for $P1$ is at most the upper bound for $P2$.
- The lower bound for $P2$ is at least the lower bound for $P1$.

Another Reduction Example

How much does it cost to multiply two n-digit numbers?

- Naive algorithm requires $\Theta(n^2)$ single-digit multiplications.
- Faster (but more complicated) algorithms are known, but none so fast as to be $O(n)$.

Is it faster to square an n-digit number than it is to multiply two n-digit numbers?

- This is a special case, so might go faster.

Answer: No, because

$$X \times Y = \frac{(X+Y)^2 - (X-Y)^2}{4}.$$

If a fast algorithm can be found for squaring, then it could be used to make a fast algorithm for multiplying.

Matrix Multiplication

Standard matrix multiplication for two $n \times n$ matrices requires $\Theta(n^3)$ multiplications.

Faster algorithms are known, but none so fast as to be $O(n^2)$.

A **symmetric** matrix is one in which $M_{ij} = M_{ji}$.

Can we multiply symmetric matrices faster than regular matrices?

$$\begin{bmatrix} 0 & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} 0 & B^T \\ B & 0 \end{bmatrix} = \begin{bmatrix} AB & 0 \\ 0 & A^T B^T \end{bmatrix}.$$

Some Puzzles

1. A hiker leaves at 8:00 AM and hikes over the mountain. The next day, she again leaves at 8:00 AM and returns to her starting point along the same path. Prove that there is a point on the path such that she was at that point at the same time on both days.

2. Take a chessboard and cover it with dominos (a domino covers two adjacent squares of the board). Now, remove the upper left and lower right corners of the board. Now, can it still be covered with dominos?

These puzzles have the quality that, while their answers may be hard to FIND, they are easy to CHECK.

3. Is 667 composite or prime?

Complexity and Computability

Complexity:

- How cheaply can this be computed?
- How hard is this to compute?

Computability:

- When can this be computed?
- Can this be computed at all?

Types of "hard" problems:

- Hard to understand (or specify) the problem
  - Software Engineering
- Hard to design a solution
  - Artificial Intelligence
- Hard to compute in reasonable time
  - Complexity Theory
- Hard (impossible) to do at all
  - Computability Theory

Hard Problems

We say that a problem is computationally "hard" if the running time of the best known algorithm is exponential on the size of its input.

Support:

- Polynomials are closed under composition and addition.
  - Doing polynomial time operations in series is polynomial.
- All computers today are polynomially related.
  - If it takes polynomial time on one computer, it will take polynomial time on any other computer.
- Polynomial time is (generally) feasible, while exponential time is (generally) infeasible.
  - An empirical observation: For most polynomial-time algorithms, the polynomial is of low degree.

Hard Problems (Cont.)

Note that for a faster machine, the size of problem that can be run in a fixed amount of time

- grows by a multiplicative factor for a polynomial-time algorithm.
- grows by an additive factor for an exponential-time algorithm.

Nondeterminism

Imagine a computer that works by guessing the correct solution from among all possible solutions to a problem.

Alternative: Super parallel machine that tests all possible solutions simultaneously.

It might solve some problems more quickly than a regular computer.

Consider a problem which, when given a proposed solution, we can check in polynomial time if the solution is correct.

Even if the number of guesses is exponential, checking (in this case) is polynomial.

Conversely: if you can't guess an answer and check in polynomial time, there can be no polynomial time algorithm!

Nondeterministic Algorithm

An algorithm is **nondeterministic** if it works by guessing the right answer from among a finite number of choices.

Alternatively, imagine a tree of choices, polynomial levels deep.

- A super parallel machine follows all branches of the tree in parallel.
- If any single branch reaches a solution, the problem is solved.

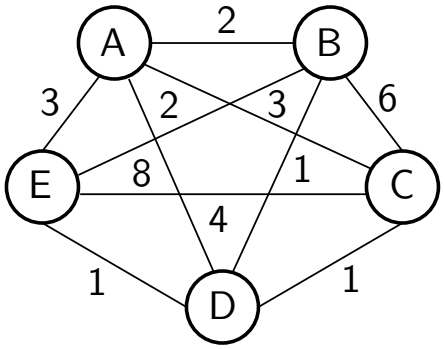
A problem that can be solved in polynomial time by a nondeterministic machine is said to be "in $\mathcal{NP}$."

Is Towers of Hanoi in $\mathcal{NP}$?

Traveling Salesman Problem

TRAVELING SALESMAN (1):

- Input: A complete, directed graph G with distances assigned to each edge in the graph.
- Output: The shortest simple cycle that includes every vertex.



Problem: How to tell if a proposed solution is *shortest*?

Traveling Salesman (Cont.)

Decision problem: A problem with a YES or NO answer.

TRAVELING SALESMAN (2):

- Input: A complete, directed graph G with distances assigned to each edge in the graph, and an integer K.
- Output: YES if there is a simple cycle with total distance $\leq K$ containing every vertex in G, and NO otherwise.

In $\mathcal{NP}$: We can guess a cycle, and quickly check if it meets the requirements.

$\mathcal{NP}$-complete Problems

Many problems are like traveling salesman:

- They are in $\mathcal{NP}$.
- Nobody knows a polynomial time algorithm.

Is there any relationship between them?

A problem X is said to be $\mathcal{NP}$-hard if ANY problem in $\mathcal{NP}$ can be reduced to X in polynomial time.

- X is AS HARD AS any problem in $\mathcal{NP}$.

A problem X is said to be $\mathcal{NP}$-complete if

1. It is in $\mathcal{NP}$.
2. It is $\mathcal{NP}$-hard.

To start the process we need to prove just one problem H is $\mathcal{NP}$-complete.

- To show that X is $\mathcal{NP}$-hard, just reduce H to X.
- DON'T GET IT BACKWARDS!

Why Care about $\mathcal{NP}$-Completeness?

Your boss asks you to write a fast program for TRAVELING SALESMAN.

- Its obviously an easy problem to understand.
- She can easily see some algorithm to solve the problem.
- It must be easy to speed up!

If you can't do the job, what do you tell her?

- I can't do it.
- I can't find evidence that anyone can do it.
- Nobody has been able to do it, despite the fact that many people have tried. Furthermore, if anyone solved any of this long list of problems, then they would be able to do this problem too.

Satisfiability

Let E be a Boolean expression over variables $x_1, x_2, ..., x_n$ in Conjunctive Normal form:

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

SATISFIABILITY (SAT):

- INPUT: A Boolean expression E over variables $x_1, x_2, ...$ in Conjunctive Normal Form.
- OUTPUT: YES if there is an assignment to the variables that makes E true, NO otherwise.

This is the "grand-daddy" $\mathcal{NP}$-complete problem.

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

$\mathcal{NP}$-completeness Proof Model

Implication: If a polynomial time algorithm can be found for ANY problem that is $\mathcal{NP}$-complete, then by a chain of polynomial time reductions, ALL $\mathcal{NP}$-complete problems can be solved in polynomial time.

To show that a decision problem X is $\mathcal{NP}$-complete:

1. Show that X is in $\mathcal{NP}$.
   - Give a polynomial-time, nondeterministic algorithm.
2. Show that X is $\mathcal{NP}$-hard.
   - Choose a known $\mathcal{NP}$-complete problem, A.
   - Describe a polynomial-time transformation that takes an ARBITRARY instance $\mathbf{I}$ of A to an instance $\mathbf{I}'$ of X.
   - Describe a polynomial-time transformation from $\mathbf{S}'$ to $\mathbf{S}$ such that $\mathbf{S}$ is the solution for $\mathbf{I}$.

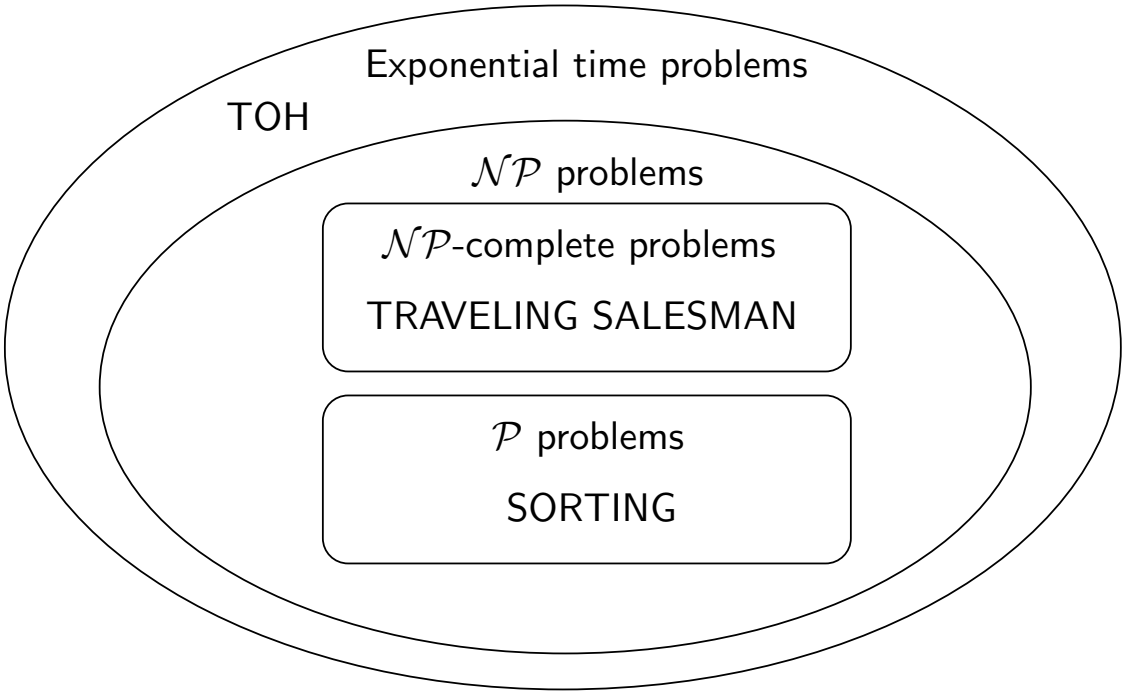
Cook's Proof Outline

1. Any decision problem can be recast as a language acceptance problem:

$$F(I) = \text{YES} \Leftrightarrow L(I') = \text{ACCEPT}.$$

2. Turing machines are a simple model of computation for writing programs that are language acceptors.
3. There is a "universal" Turing machine that can take as input a description for a Turing machine, and an input string, and return the execution of that machine on that string.
4. This in turn can be cast as a boolean expression such that the expression is satisfiable if and only if the Turing machine yields ACCEPT for that string.
5. Thus, *any* decision problem that is performable by the Turing machine is transformable to SAT: This is $\mathcal{NP}$-hard.

The World of Exponential-time(?) Problems



Question: Does $\mathcal{P} = \mathcal{NP}$?

3-SATISFIABILITY (3 SAT)

Input: Boolean expression E in CNF such that each clause contains exactly 3 literals.

Output: YES if the expression can be satisfied, NO otherwise.

A special case of SAT.

- Is 3 SAT easier than SAT?

Theorem: 3 SAT is $\mathcal{NP}$-complete.

Proof:

- 3 SAT is in $\mathcal{NP}$.
  - Guess (nondeterministically) values for the variables.
  - The correctness of the guess can be verified in polynomial time.
- 3 SAT is $\mathcal{NP}$-hard, by a reduction from SAT to 3 SAT.

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

Find a polynomial time reduction from SAT to 3 SAT.

Let $E = C_1 \cdot C_2 \cdot ... \cdot C_k$ by 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 $C_i = x_1 + x_2 + ... + x_j$ where $x_1, ..., x_j$ are literals.

Replacement

1. $j = 1$, so $C_i = x_1$. Replace C_i with

   $(x_1 + v + w) \cdot (x_1 + \overline{v} + w) \cdot (x_1 + v + \overline{w}) \cdot (x_1 + \overline{v} + \overline{w})$

   where v and w are new variables.
2. $j = 2$, so $C_i = (x_1 + x_2)$. Replace C_i with

   $$(x_1 + x_2 + z) \cdot (x_1 + x_2 + \overline{z})$$

   where z is a new variable.
3. $j > 3$. Replace C_i with

   $(x_1 + x_2 + z_1) \cdot (x_3 + \overline{z_1} + z_2) \cdot (x_4 + \overline{z_2} + z_3) \cdot ...$

   $\cdot (x_{j-2} + \overline{z_{j-4}} + z_{j-3}) \cdot (x_{j-1} + x_j + \overline{z_{j-3}})$

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

After appropriate replacements have been made for each C_i, a Boolean expression results that is an instance of 3 SAT.

Each replacement is satisfiable if and only if the original clause is satisfiable.

The reduction is clearly polynomial time.

Third Case

If E is satisfiable, then E' is satisfiable:

- Assume x_m is assigned true.
- Then assign $z_t, t \leq m-2$ as true and $z_k, t \geq m-1$ as false.
- Then all clauses in Case (3) are satisfied.

If E' is satisfiable, then E is satisfiable:

- Proof by contradiction.
- If $x_1, x_2, ..., x_j$ are all false, then $z_1, z_2, ..., z_{j-3}$ are all true.
- But then $(x_{j-1} + x_{j-2} + \overline{z_{j-3}})$ is false, a contradiction.

(Not necessary for proof, but may help insight.)
Conversely, if E is not satisfiable, then E' is not satisfiable.

- E not satisfiable means all x_i are false.
- This leaves E' as

  $$(z_1) \cdot (\overline{z_1} + z_2) \cdot ... \cdot (\overline{z_{j-4}} + z_{j-3}) \cdot (\overline{z_{j-3}})$$

  which is NOT satisfiable.

Two Problems

VERTEX COVER:

Input: An undirected graph G and an integer k.
Output: YES if there is a subset of vertices in G of size k or less such that every edge in the graph has at least one of its ends in the subset; NO otherwise.

K-CLIQUE:

Input: An undirected graph G and an integer k.
Output: YES if there is a subset of the vertices of size k or greater that is a complete graph (a clique).

We can reduce either problem to the other by switching G to its inverse G'.

- If edge (i, j) is in G, it is NOT in G'.
- If edge (i, j) is NOT in G, it IS in G'.

K CLIQUE is $\mathcal{NP}$-Complete

Easy to show that K CLIQUE is in $\mathcal{NP}$.

Reduce SAT to K CLIQUE.

An instance of SAT is a Boolean expression

$$B = C_1 \cdot C_2 \cdot ... \cdot C_m$$

where

$$C_i = y[i, 1] + y[i, 2] + ... + y[i, k_i].$$

Transform this to an instance of K CLIQUE as follows.

$$V = \{v[i, j] | 1 \leq i \leq m, 1 \leq j \leq k_i\}.$$

All vertices $v[i_1, j_1]$ and $v[i_2, j_2]$ have an edge between them UNLESS they are two literals within the same clause ($i_1 = i_2$) OR they are opposite values for the same variable.

Set $k = m$.

Example

$$B = (y_1 + y_2) \cdot (\overline{y_1} + y_2 + y_3).$$

B is satisfiable if and only if G has a clique of size $\geq k$.

- B satisfiable implies there is a truth assignment such that $y[i, j_i]$ is true for each i.
- But then, $v[i, j_i]$ must be in a clique of size $k = m$.
- If G has a clique of size $\geq k$, then the clique must have size exactly k and there is one vertex $v[i, j_i]$ in the clique for each i.
- There is a truth assignment making each $y[i, j_i]$ true. That truth assignment satisfies B.

We conclude that K CLIQUE is $\mathcal{NP}$-hard, therefore $\mathcal{NP}$-complete.

Coping with $\mathcal{NP}$-Completeness

1. Organize to reduce costs.
   - Dynamic programming.
   - Backtracking.
   - Branch and Bounds.
2. Find subproblems of the original problem that have polynomial-time solutions.
   - Significant special cases that are useful to answer.
3. Approximation algorithms.
4. Randomized algorithms.
5. Use heuristics.
   - Greedy algorithms.
   - Simulated Annealing.
   - Genetic Algorithms.

Knapsack Analysis Revisited

Fact: Knapsack is $\mathcal{NP}$-complete.

But we have a $\Theta(nK)$ algorithm!!

Question: How big is K?

- Input size is typically $O(n \log K)$ since the item sizes are smaller than K.
- Thus, $\Theta(nK)$ is exponential on input size.

This algorithm is tractable if the numbers are "reasonable."

- nK can be thousands.
- This is different from TRAVELING SALESMAN which cannot handle $n = 100$.

Such an algorithm is called a **pseudo-polynomial** time algorithm.

Subproblems and Special Cases

Some restricted cases of $\mathcal{NP}$-complete problems are useful, and not $\mathcal{NP}$-complete.

- VERTEX COVER and K CLIQUE have polynomial time algorithms for bipartite graphs.
- 2-SATISFIABILITY has a polynomial time solution.
- Several geometric problems are polynomial-time in two dimensions, but not in three or more.
- KNAPSACK is polynomial if the numbers are "small."

Approximation Algorithms

Seek algorithms for optimization problems with a guaranteed bound on quality of the solution.

For VERTEX COVER:

- Let M be a maximal (not necessarily maximum) **matching** in G.
  - A matching pairs vertices (with connecting edges) so that no vertex is paired with more than one match.
  - Maximal means pick as many pairs as possible.
- If OPT is the size of a minimum vertex cover, then

  $$|M| \leq 2 \cdot \text{OPT}$$

  because at least one endpoint of every matched edge must be in ANY vertex cover.

BIN PACKING

INPUT: Numbers $x_1, x_2, ..., x_n$ between 0 and 1, and an unlimited supply of bins of size 1.

OUTPUT: An assignment of numbers to bins that requires the fewest possible number of bins (no bin can hold numbers whose sum exceeds 1).

This problem is $\mathcal{NP}$-complete.

Heuristic: First fit

- Place a number in the first bin that fits.
- The number of bins used is no more than twice the sum of the numbers.
- First fit can be much worse than optimal.
- Consider 6 of $1/7 + \epsilon$, 6 of $1/3 + \epsilon$, 6 of $1/2 + \epsilon$.

Better Heuristic: Decreasing first fit

- Sort the items, then use first fit.
- This can be proven to yield no more than 11/9 the optimal number of bins.

Summary

The theory of $\mathcal{NP}$-completeness gives a technique for separating tractable from (probably) untractable problems.

When faced with a new problem, we might alternate between:

- Check if it is tractable (find a fast solution).
- Check if it is intractable (prove the problem is $\mathcal{NP}$-complete).

If the problem is in $\mathcal{NP}$-complete, then use one of the "coping" strategies.