Tractable Problems

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

- It is said to be \textbf{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: \( O(n^3) \)

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

- $\text{CLIQUE} \leq_P \text{INDEPENDENT SET}$.
- 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:
\[
\text{nd-choice}(\text{ch}_1, \text{ch}_2, \ldots, \text{ch}_j)
\]
returns one of the choices \(\text{ch}_1, \text{ch}_2, \ldots\) arbitrarily.

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

Nondeterministic CLIQUE Algorithm

```plaintext
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, $P \subset \mathcal{NP}$.

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

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

$$E = (x_5 + x_7 + x_3 + x_10) \cdot (x_2 + x_3) \cdot (x_1 + x_3 + x_6).$$

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.

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

A is not permitted to be harder than $\mathcal{NP}$. For example, Tower of Hanoi is not in $\mathcal{NP}$. It requires exponential time to verify a set of moves.

Is there a truth assignment for the variables that makes $E$ true? Cook won a Turing award for this work.
Proof Sketch

SAT ∈ \text{\textsc{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 \text{\textsc{np}}-hard:  
- Start with an arbitrary problem \( B \in \text{\textsc{np}} \).
- 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 \text{\textsc{np}}-complete, we have not defined an empty concept.
2. If SAT \( \in \text{\textsc{p}} \), then \( \text{\textsc{p}} = \text{\textsc{np}} \).
3. If \( \text{\textsc{p}} = \text{\textsc{np}} \), then SAT \( \in \text{\textsc{p}} \).
4. If \( A \in \text{\textsc{np}} \) and \( B \) is \text{\textsc{np}}-complete, then \( B \leq \text{\textsc{p}} A \) implies \( A \) is \text{\textsc{np}}-complete.
   Proof:
   - Let \( C \in \text{\textsc{np}} \).
   - Then \( C \leq \text{\textsc{p}} B \) since \( B \) is \text{\textsc{np}}-complete.
   - Since \( B \leq \text{\textsc{p}} A \) and \( \leq \text{\textsc{p}} \) is transitive, \( C \leq \text{\textsc{p}} A \).
   - Therefore, \( A \) is \text{\textsc{np}}-hard and, finally, \text{\textsc{np}}-complete.

Implications (cont)

5. This gives a simple two-part strategy for showing a decision problem \( A \) is \text{\textsc{np}}-complete.
   (a) Show \( A \in \text{\textsc{np}} \).
   (b) Pick an \text{\textsc{np}}-complete problem \( B \) and show \( B \leq \text{\textsc{p}} A \).

\text{\textsc{np}}-completeness Proof Paradigm

To show that decision problem \( B \) is \text{\textsc{np}}-complete:

- \( B \in \text{\textsc{np}} \)
  - 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 \text{\textsc{np}}-hard.
  - Choose a known \text{\textsc{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 \text{\textsc{np}}.

Implications (cont)

Proving \( A \in \text{\textsc{np}} \) is usually easy.

Don’t get the reduction backwards!

\( B \in \text{\textsc{np}} \) is usually the easy part.

The first two steps of the \text{\textsc{np}}-hard proof are usually the hardest.
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 $E$?

A special case of SAT.

One might hope that 3SAT is easier than SAT.

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

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

procedure nd-3SAT($E$) {
    for (i = 1 to $n$)
        $x[i] = \text{nd-choice(TRUE, FALSE)}$;  
    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

Choose SAT to be the known $\mathcal{NP}$-complete problem.

- We need to show that SAT $\leq_p$ 3SAT.
- Let $E = C_1 \cdot C_2 \cdots C_b$ 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 $y_1, \ldots, y_j$ are literals.

(a) $j = 1$

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

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

(b) $j = 2$

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

(c) $j > 3$

- Replace $(y_1 + y_2 + \cdots + y_j)$ with
  $$
  (y_1 + y_2 + z_1) \cdot (y_3 + \overline{z_1} + z_2) \cdot (y_4 + \overline{z_2} + z_3) \cdots 
  (y_{j-1} + \overline{z_{j-2}} + z_{j-1}) \cdot (y_j + \overline{z_{j-1}} + \overline{z_{j-3}})
  $$

  where $z_1, z_2, \ldots, z_{j-3}$ 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.
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.

By restriction, we have truth assignment for $E'$.

(a) $y_1$ is necessarily true.
(b) $y_1 + y_2$ is necessarily true.
(c) Proof by contradiction:
   - If $y_1, y_2, \ldots, y_t$ are all false, then $z_1, z_2, \ldots, z_{t-3}$ are all true.
   - But then $(y_{t-1} + y_{t-2} + z_{t-3})$ is false, a contradiction.

We conclude $\text{SAT} \leq 3\text{SAT}$ and $3\text{SAT}$ is $\mathcal{NP}$-complete.

Tree of Reductions

Reductions go down the tree.

Proofs that each problem $\in \mathcal{NP}$ are straightforward.

Perspective

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 $P$-time algorithm.

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

More on this observation later.
SAT ≤ₚ CLIQUE

(1) Easy to show CLIQUE in NP.
(2) An instance of SAT is a Boolean expression
   \[ B = C_1 \cdot C_2 \cdots C_m, \]
   where
   \[ C_i = y[i, 1] + y[i, 2] + \cdots + y[i, k_i]. \]
   Transform this to an instance of CLIQUE \( G = (V, E) \) and \( K \).
   \[ V = \{ v[i,j] | 1 \leq i \leq m, 1 \leq j \leq k_i \} \]
   Two vertices \( v[i,j] \) and \( v[i',j'] \) are adjacent in \( G \) if \( i \neq i' \)
   AND EITHER \( y[i,j] \) and \( y[i',j'] \) are the same literal
   OR \( y[i,j] \) and \( y[i',j'] \) have different underlying variables.
\( K = m. \)

We conclude that CLIQUE is NP-hard, therefore \( \text{NP} \)-complete.

SAT ≤ₚ CLIQUE (cont)

Example: \( B = (x_1 + x_2) \cdot (\overline{x_1} + x_2 + x_3). \)
\( K = 2. \)

(3) \( B \) is satisfiable iff \( G \) has clique of size \( \geq K. \)
   \begin{itemize}
   \item \( B \) is satisfiable implies there is a truth assignment such
     that \( y[i,j] \) is true for each \( i. \)
   \item But then \( v[i,j] \) must be in a clique of size \( K = m. \)
   \item If \( G \) has a clique of size \( \geq K \), then the clique must have
     size exactly \( K \) and there is one vertex \( v[i,j] \) in the clique
     for each \( i. \)
   \item There is a truth assignment making each \( y[i,j] \) true.
     That truth assignment satisfies \( B. \)
   \end{itemize}
We conclude that CLIQUE is NP-hard, therefore \( \text{NP} \)-complete.

Co-NP

\begin{itemize}
\item Note the asymmetry in the definition of NP.
\begin{itemize}
\item The non-determinism can identify a clique, and you can
   verify it.
\item But what if the correct answer is “NO”? How do you
   verify that?
\end{itemize}
\item Co-NP: The complements of problems in NP.
\begin{itemize}
\item Is a boolean expression always false?
\item Is there no clique of size \( k \)?
\end{itemize}
\item It seems unlikely that \( \text{NP} = \text{co-NP}. \)
\end{itemize}

Is NP-complete = NP?

\begin{itemize}
\item It has been proved that if \( P \neq \text{NP}, \) then \( \text{NP-complete} \neq \text{NP}. \)
\item The following problems are not known to be in \( P \) or \( \text{NP}, \)
   but seem to be of a type that makes them unlikely to be in \( \text{NP}. \)
\begin{itemize}
\item GRAPH ISOMORPHISM: Are two graphs isomorphic?
\item COMPOSITE NUMBERS: For positive integer \( K, \) are
   there integers \( m, n > 1 \) such that \( K = mn? \)
\item LINEAR PROGRAMMING
\end{itemize}
\end{itemize}
PARTITION \leq_{p} KNAPSACK

PARTITION is a special case of KNAPSACK in which

\[ K = \frac{1}{2} \sum_{a \in A} s(a) \]

assuming \( \sum s(a) \) is even.

Assuming PARTITION is \( \mathcal{NP} \)-complete, KNAPSACK is \( \mathcal{NP} \)-complete.

---

“Practical” Exponential Problems

- What about our \( O(KN) \) dynamic prog algorithm?
- Input size for KNAPSACK is \( O(N \log K) \)
  - Thus \( O(KN) \) is exponential in \( N \log K \).
- The dynamic programming algorithm counts through numbers \( 1, \ldots, K \). Takes exponential time when measured by number of bits to represent \( K \).
- If \( K \) is “small” \( (K = O(p(N))) \), then algorithm has complexity polynomial in \( N \) and is truly polynomial in input size.
- An algorithm that is polynomial-time if the numbers in the input are “small” (as opposed to number OF inputs) is called a pseudo-polynomial time algorithm.

---

“Practical” Problems (cont)

- Lesson: While KNAPSACK is \( \mathcal{NP} \)-complete, it is often not that hard.
- Many \( \mathcal{NP} \)-complete problems have no pseudo-polynomial time algorithm unless \( P = \mathcal{NP} \).

---

Coping with \( \mathcal{NP} \)-completeness

1. Find subproblems of the original problem that have polynomial-time algorithms.
2. Approximation algorithms.
3. Randomized Algorithms.
4. Backtracking; Branch and Bound.
5. Heuristics.
   - Greedy.
   - Simulated Annealing.
   - Genetic Algorithms.

---

The assumption about PARTITION is true, though we do not prove it.

The “transformation” is simply to pass the input of PARTITION to KNAPSACK.

This is an important point, about the input size. It has to do with the “size” of a number (a value). We represent the value \( n \) with \( \log n \) bits, or more precisely, \( \log N \) bits where \( N \) is the maximum value. In the case of KNAPSACK, \( K \) (the knapsack size) is effectively the maximum number. We will use this observation frequently when we analyze numeric algorithms.

The issue is what size input is practical. The problems we want to solve for Traveling Salesman are not practical.

The subproblems need to be “significant” special cases.

Approximation works for optimization problems (and there are a LOT of those).

Randomized Algorithms typically work well for problems with a lot of solutions.

(4) gives ways to (relatively efficiently) implement nd-choice.
Subproblems

Restrict attention to special classes of inputs.
Examples:
- VERTEX COVER, INDEPENDENT SET, and CLIQUE, when restricted to bipartite graphs, all have polynomial-time algorithms (for VERTEX COVER, by reduction to NETWORK FLOW).
- 2-SATISFIABILITY, 2-DIMENSIONAL MATCHING and EXACT COVER BY 2-SETS all have polynomial time algorithms.
- PARTITION and KNAPSACK have polynomial time algorithms if the numbers in an instance are all $O(p(n))$.
- However, HAMILTONIAN CIRCUIT and 3-COLORABILITY remain $NP$-complete even for a planar graph.

Backtracking

We may view a nondeterministic algorithm executing on a particular instance as a tree:
- Each edge represents a particular nondeterministic choice.
- The checking occurs at the leaves.

Example:

Each leaf represents a different set $S$. Checking that $S$ is a clique of size $\geq K$ can be done in polynomial time.

Backtracking (cont)

Backtracking can be viewed as an in-order traversal of this tree with two criteria for stopping.
- A leaf that accepts is found.
- A partial solution that could not possibly lead to acceptance is reached.

Example:

There cannot possibly be a set $S$ of cardinality $\geq 2$ under this node, so backtrack.

Since $(1, 2) \notin E$, no $S$ under this node can be a clique, so backtrack.

Branch and Bound

- For optimization problems, more sophisticated kind of backtracking.
- Use the best solution found so far as a bound that controls backtracking.
- Example Problem: Given a graph $G$, find a minimum vertex cover of $G$.
- Computation tree for nondeterministic algorithm is similar to CLIQUE.
  - Every leaf represents a different subset $S$ of the vertices.
  - Whenever a leaf is reached and it contains a vertex cover of size $B$, $B$ is an upper bound on the size of the minimum vertex cover.
  - Use $B$ to prune any future tree nodes having size $\geq B$. Whenever a smaller vertex cover is found, update $B$. When the corresponding decision problem is $NP$-complete.
Branch and Bound (cont)

- Improvement:
  - Use a fast, greedy algorithm to get a minimal (not
    minimum) vertex cover.
  - Use this as the initial bound \( B \).
- While Branch and Bound is better than a brute-force
  exhaustive search, it is usually exponential time, hence
  impractical for all but the smallest instances.
  - ... if we insist on an optimal solution.
- Branch and Bound often practical as an approximation
  algorithm where the search terminates when a “good
  enough” solution is obtained.

Approximation Algorithms

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

VERTEX COVER: Given a graph \( G = (V, E) \), find a vertex
cover of minimum size.

Let \( M \) be a maximal (not necessarily maximum) matching in
\( G \) and let \( V' \) be the set of matched vertices.
If \( OPT \) is the size of a minimum vertex cover, then
\[
|V'| \leq 2OPT
\]
because at least one endpoint of every matched edge must
be in any vertex cover.

Bin Packing

We have numbers \( x_1, x_2, \ldots, x_n \) between 0 and 1 as well as
an unlimited supply of bins of size 1.

Problem: Put the numbers into as few bins as possible so
that the sum of the numbers in any one bin does not exceed
1.

Example: Numbers 3/4, 1/3, 1/2, 1/8, 2/3, 1/2, 1/4.

Optimal solution: \([3/4, 1/8], [1/2, 1/3], [1/2, 1/4], [2/3]\).

First Fit Algorithm

Place \( x_i \) into the first bin.

For each \( i, 2 \leq i \leq n \), place \( x_i \) in the first bin that will contain
it.

No more than 1 bin can be left less than half full.
The number of bins used is no more than twice the sum of
the numbers.

The sum of the numbers is a lower bound on the number of
bins in the optimal solution.

Therefore, first fit is no more than twice the optimal number
of bins.
First Fit Does Poorly

Let $\epsilon$ be very small, e.g., $\epsilon = .00001$.

Numbers (in this order):
- 6 of $(1/7 + \epsilon)$.
- 6 of $(1/3 + \epsilon)$.
- 6 of $(1/2 + \epsilon)$.

First fit returns:
- 1 bin of $[6$ of $1/7 + \epsilon]$
- 3 bins of $[2$ of $1/3 + \epsilon]$
- 6 bins of $[1/2 + \epsilon]$

Optimal solution is 6 bins of $[1/7 + \epsilon, 1/3 + \epsilon, 1/2 + \epsilon]$.

First fit is $5/3$ larger than optimal.

Decreasing First Fit

It can be proved that the worst-case performance of first-fit is $17/10$ times optimal.

Use the following heuristic:
- Sort the numbers in decreasing order.
- Apply first fit.
- This is called decreasing first fit.

The worst case performance of decreasing first fit is close to $11/9$ times optimal.

Summary

- The theory of $NP$-completeness gives us a technique for separating tractable from (probably) intractable problems.
- When faced with a new problem requiring algorithmic solution, our thought process might resemble this scheme:

  
<table>
<thead>
<tr>
<th>Is it (NP)-complete?</th>
<th>Is it in (P)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alternately think about each question. Lack of progress on either question might give insights into the answer to the other question.</td>
<td>Once an affirmative answer is obtained to one of these questions, one of two strategies is followed.</td>
</tr>
</tbody>
</table>

Strategies

1. The problem is in $P$.
   - This means there are polynomial-time algorithms for the problem, and presumably we know at least one.
   - So, apply the techniques learned in this course to analyze the algorithms and improve them to find the lowest time complexity we can.

2. The problem is $NP$-complete.
   - Apply the strategies for coping with $NP$-completeness.
   - Especially, find subproblems that are in $P$, or find approximation algorithms.

That is the only way we could have proved it is in $P$. 