IS703: Decision Support and Optimization

Week 1: Introduction, Asymptotics, Complexity

Lau Hoong Chuin
School of Information Systems
Objectives

• **Introduction to DSS**
  - Understanding DSS (source: Chapter 10, Introduction to Information Technology by Turban, Rainer and Potter)

• **Course Outline**

• **Computational Complexity**
  - Asymptotics
  - Complexity
Information Systems

• An information system converts data into information and is designed to collect, transform, and disseminate information.

• Information technology represents various types of hardware and software used in an information system.
Decision Support Systems (DSS)

Intelligent information system that combines models and data in an attempt to support decision-making with extensive user involvement.

Applications:
• University information systems
• Credit card companies
• Telephone companies
• Airlines, hotels, and hospitals
• Manufacturing and Logistics companies
• Investment companies
Components of DSS

- Data
- Analysis
- Modeling and Optimization
- Simulation
- Presentation of Results
- Graphical User Interface

Information System

Decision Support System
Characteristics and Capabilities of DSSs

- **Goal-seeking analysis.** Study that attempts to find the value of the inputs necessary to achieve a desired level of output.

- **Sensitivity analysis.** The study of the impact that changes in one (or more) parts of a model have on other parts.

- **What-if analysis.** The study of the impact of a change in the assumptions (input data) on the proposed solution.
Schematic of Decision Support System
Decision Support Framework

<table>
<thead>
<tr>
<th>Type of Decision</th>
<th>Operational Control</th>
<th>Management Control</th>
<th>Strategic Planning</th>
<th>Support Needed</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structured</td>
<td>Accounts receivable, order entry</td>
<td>Budget analysis, short-term forecasting, personnel reports, make-or-buy analysis</td>
<td>Financial management (investment), warehouse location, distribution systems</td>
<td>MIS, management science models, financial and statistical models</td>
</tr>
<tr>
<td>Semistructured</td>
<td>Production scheduling, inventory control</td>
<td>Credit evaluation, budget preparation, plant layout, project scheduling, reward systems design</td>
<td>Building new plant, mergers and acquisitions, new product planning, compensation planning, quality assurance planning</td>
<td>DSS</td>
</tr>
<tr>
<td>Unstructured</td>
<td>Selecting a cover for a magazine, buying software, approving loans</td>
<td>Negotiating, recruiting an executive, buying hardware, lobbying</td>
<td>R &amp; D planning, new technology development, social responsibility planning</td>
<td>DSS ES neural networks</td>
</tr>
<tr>
<td>Support Needed</td>
<td>MIS, management science</td>
<td>Management science, DSS, EIS, ES</td>
<td>EIS, ES, neural networks</td>
<td></td>
</tr>
</tbody>
</table>
Emerging Types of DSS

- **Frontline decision making.** The process by which companies automate decision process and push them down into the organization and sometimes out to partners.

- **Real-Time decision support.** The systems that supports business decisions that must be made at the right time and frequently under time pressure.

- **Group (Collaborative) decision support system.** An interactive computer-based system that supports the process of finding solutions by a group of decision makers.
Search and Optimization Technologies in DSS

- Underpins the development of DSS
- Multi- (Inter-) Disciplinary:
  - Computer Science
  - Mathematics
  - Engineering
  - Business (Decision Science)
  - Economics
- Interface of Operations Research and AI
- “Tool box” of methodologies

Course Outline
Problem Formulation

• What is the **objective**?
  – Maximize profit,
  – Minimize inventory, ...

• What are the **decision variables**?
  – Capacity, routing, production and stock levels

• What are the **constraints**?
  – Capacity is limited by capital
  – Production is limited by capacity
Asymptotics

Reference:
• CLRS Chapters 1-3, Appendix A

Objectives:
• To understand the basic concepts of algorithm design and analysis
• To learn asymptotic analysis
Background

• “Algorithm”
  – derived from Mohammed Al-Khowarizmi, 9th Century Persian mathematician

• A computational **problem** is defined by an input/output relationship, e.g.

  **Sorting Problem:**
  **Input:** A sequence of n numbers \([x_1, \ldots, x_n]\)
  **Output:** A permutation \([x_1', \ldots, x_n']\) of the input s.t. \(x_1' \leq x_2' \leq \ldots \leq x_n'\)

• An (input) **instance**

• An algorithm for a computational problem is **correct** if, for every instance, it halts with the correct output.
Design of Algorithms

- Eight paradigms for designing a good algorithm:
  - Reduce to a known problem (e.g. graph problem)
  - Divide-and-Conquer
  - Greedy Algorithm
  - Dynamic Programming
  - Branch and Bound Search
  - Local Search
  - Metaheuristics: tabu search, genetic algorithms, simulated annealing, etc.
  - Randomized/Probabilistic Algorithm
Analysis of Algorithms

• Analysis is performed with respect to a computational model

• We use a generic uni-processor random-access machine (RAM)
  – All memory equally expensive to access
  – No concurrent operations
  – All reasonable instructions take unit time
    • Except, of course, function calls
  – Constant word size
Analysis of Algorithms

• How does an algorithm behave as the problem size gets very large?
  – Running time / number of operations (time complexity)
  – Memory requirements (space complexity)
  – Bandwidth/power requirements/logic gates/etc.

• Complexity of an algorithm is expressed as a function of the input size, e.g.
  – Sorting: number of input items
  – Multiplication: total number of bits
  – Graph algorithms: number of nodes & edges

• Asymptotic Analysis (Order of Growth)
  – Ignore coefficients of terms. Why?
  – Consider only the most dominant term
Types of Analysis

Let $t(X)$ denote the time the algo takes on instance $X$.

1. **Worst case**
   - Provides an upper bound on running time
   - $f(n) = \max_{|X|=n} \{t(X)\}$
   - An absolute guarantee

2. **Average case**
   - Provides the expected running time
   - $f(n) = \sum_{|X|=n} \{t(X) \cdot Pr(X)\}$
   - Very useful, but treat with care: what is “average”?  
     - Random (equally likely) inputs? Real-life inputs?

3. **Amortized analysis**
   - Amortized over $k$ operations: $f(n) = (1/k) \max_{X_1, \ldots, X_k, |X_j|=n} \{ \sum_{j=1..k} t(X_j)\}$
   - Not within the scope of this course
Role of Data Structures in Algorithm Design

- Support efficient implementation of algorithms

<table>
<thead>
<tr>
<th>Choice</th>
<th>Insert()</th>
<th>deleteMin()</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unsorted array</td>
<td>O(1)</td>
<td>O(n)</td>
</tr>
<tr>
<td>Sorted linked list</td>
<td>O(n)</td>
<td>O(1)</td>
</tr>
<tr>
<td>Binary search tree</td>
<td>O(n) worst, O(log n) avg.</td>
<td>O(n) worst, O(log n) avg</td>
</tr>
<tr>
<td>Binary heaps</td>
<td>O(log n)</td>
<td>O(log n)</td>
</tr>
<tr>
<td>Fibonacci heaps</td>
<td>O(1) (amortized)</td>
<td>O(log n) (amortized)</td>
</tr>
</tbody>
</table>

e.g. Dijkstra’s shortest path algorithm: $O(n^2) \rightarrow O(n \log n + m)$ with F-heaps
Why are faster algorithms a concern?
Why are faster algorithms a concern?

Largest size of problem that can be solved within:

<table>
<thead>
<tr>
<th>f(n)</th>
<th>1 sec</th>
<th>1 day</th>
<th>1 century</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt(n)</td>
<td>$10^{12}$</td>
<td>$7.5 \times 10^{21}$</td>
<td>$10^{31}$</td>
</tr>
<tr>
<td>n</td>
<td>$10^6$</td>
<td>$8.6 \times 10^{10}$</td>
<td>$3.1 \times 10^{15}$</td>
</tr>
<tr>
<td>n lg n</td>
<td>60,000</td>
<td>$2.8 \times 10^9$</td>
<td>$6.9 \times 10^{13}$</td>
</tr>
<tr>
<td>n^2</td>
<td>1000</td>
<td>$2.9 \times 10^5$</td>
<td>$5.6 \times 10^7$</td>
</tr>
<tr>
<td>n^3</td>
<td>100</td>
<td>4400</td>
<td>$1.5 \times 10^5$</td>
</tr>
<tr>
<td>2^n</td>
<td>20</td>
<td>36</td>
<td>51</td>
</tr>
<tr>
<td>n!</td>
<td>9.5</td>
<td>14</td>
<td>17.8</td>
</tr>
</tbody>
</table>

Question: How fast does log n grow?
Upper Bound (Big Oh) Notation

- Time complexity of Insertion Sort = $O(n^2)$
- $f(n) = O(g(n))$: 
  - $g(n)$ is an asymptotic upper bound of $f(n)$
  - $\exists$ positive constants $c$ and $n_0$ s.t. $f(n) \leq c \cdot g(n)$ $\forall$ $n \geq n_0$
- Strictly speaking,
  - $O(g(n)) = \{ f(n) : \exists$ positive constants $c$ and $n_0$ s.t. $f(n) \leq c \cdot g(n)$ $\forall$ $n \geq n_0 \}$
Lower Bound (Omega) Notation

- \( f(n) = \Omega(g(n)) \):
  - \( g(n) \) is an asymptotic lower bound of \( f(n) \)
  - \( \exists \) positive constants \( c \) and \( n_0 \) s.t. \( f(n) \geq c \cdot g(n) \) \( \forall n \geq n_0 \)
Tight Asymptotic (Theta) Bound

- \( f(n) = \Theta(g(n)) \):
  - \( g(n) \) is an asymptotic **tight** bound of \( f(n) \)
  - If \( \exists \) positive constants \( c_1, c_2, \) and \( n_0 \) s.t.
    \[
    c_1 g(n) \leq f(n) \leq c_2 g(n) \quad \forall \ n \geq n_0
    \]
- \( f(n) \) is \( \Theta(g(n)) \) iff \( f(n) \) is both \( O(g(n)) \) and \( \Omega(g(n)) \)
Algorithm Performance Categories

An algorithm with running time $\Theta(1)$ is said to run in constant time.
An algorithm with running time $\Theta(\log n)$ is said to run in logarithmic time.
An algorithm with running time $\Theta(n)$ is said to run in linear time.
An algorithm with running time $\Theta(n^2)$ is said to run in quadratic time.
An algorithm with running time $O(n^k)$, for some constant $k$ is said to run in polynomial time.

What about exponential time?
Loose Asymptotic Upper Bound

- $f(n) = o(g(n))$:
  - $g(n)$ is a loose upper bound of $f(n)$
  - If for any positive constant $c$, $\exists n_0$ s.t. $f(n) < c \cdot g(n) \forall n \geq n_0$

i.e. $f(n)$ becomes insignificant relative to $g(n)$ as $n$ approaches infinity:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Example: $2n = o(n^2)$, but $2n^2 \neq o(n^2)$
Loose Asymptotic Lower Bound

- $f(n) = \omega(g(n))$: 
  - $g(n)$ is a loose lower bound of $f(n)$
  - if for any positive constant $c \exists n_0$ s.t. $c \cdot g(n) < f(n) \quad \forall \ n \geq n_0$

\[ \lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty \]

Example: $n^2/2 = \omega(n)$, but $n^2/2 \neq \omega(n^2)$
Check

Have you understood the meaning of Tight and Loose Bounds?
Asymptotic Comparisons of Functions

• **Intuitively,**
  - \( o() \) is like \(< \)
  - \( \omega() \) is like \(>\)
  - \( \Theta() \) is like \(=\)
  - \( O() \) is like \(\leq\)
  - \( \Omega() \) is like \(\geq\)

• **Transitivity**
  - \( f(n) = \Omega(g(n)) \) and \( g(n) = \Omega(h(n)) \) imply \( f(n) = \Omega(h(n)) \), etc

• **Reflexivity**
  - \( f(n) = O(f(n)) \), \( f(n) = \Omega(f(n)) \), \( f(n) = \Theta(f(n)) \)

• **Symmetry**
  - \( f(n) = \Theta(g(n)) \) iff \( g(n) = \Theta(f(n)) \)
  - \( f(n) = O(g(n)) \) iff \( g(n) = \Omega(f(n)) \)
  - \( f(n) = o(g(n)) \) iff \( g(n) = \omega(f(n)) \)
Limits and Asymptotic Analysis

Suppose \( \lim_{n \to \infty} \frac{f(n)}{g(n)} = L \)

1. \( L=0 : f = o(g) \)
2. \( L=\infty : f = \omega(g) \)
3. \( L\neq0 : f = \Theta(g) \)
4. Not defined : No asymptotic relationship between \( f \) and \( g \).

Example:
Let \( f(n)=n^2+n, \ g(n)=2n^2 \). Then

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{n^2 + n}{2n^2} = \lim_{n \to \infty} \frac{1 + 1/n}{2} = \frac{1}{2}
\]

So \( f = \Theta(g) \)
L’Hopital’s Rule

When the limits of \( f(n)/g(n) \) end up as \( 0/0 \) or \( \infty/\infty \), then we need to apply the rule by taking the limits of their derivatives.

Example:

Let \( f(n)=\ln n \) and \( g(n)=n \). Then

\[
\lim_{n \to \infty} \frac{f(n)}{g(n)} = \lim_{n \to \infty} \frac{\ln n}{n} = \infty/\infty
\]

\[
= \lim_{n \to \infty} \frac{(\ln n)'}{(n)'} = \lim_{n \to \infty} \frac{1/n}{1} = 0
\]

So \( f = o(g) \)
Asymptotic notation involving 2 parameters

\[ f(n,m) = O(g(n,m)) \] if

\[ \exists \text{ positive constants } c \text{ and } n_0 \text{ and } m_0 \text{ s.t.} \]

\[ f(n,m) \leq c \ g(n,m) \text{ for all } n \geq n_0 \text{ and } m \geq m_0 \]

e.g. \[ f(n,m) = n^2 + m^2 \text{ and } g(n,m) = n^3 + m^3, \]
\[ f(n,m) = O(g(n,m)) \text{ by choosing } c = n_0 = m_0 = 1 \]
i.e., \[ n^2 + m^2 \leq n^3 + m^3, \text{ for all } n, m \geq 1 \]

Similarly, we can define lower bound and tight bounds involving 2 or more parameters.
Another example

\[ f(n,m) = nm^2 \text{ and } g(n,m) = n^2m \]

for any given values of \( c, n_0 \), and \( m_0 \), it is impossible to have \( nm^2 \leq c n^2m \) for all \( n \geq n_0 \) and \( m \geq m_0 \).

Why? Divide \( nm \) on both sides:

\[ m \leq cn, \text{ for } n \geq n_0 \text{ and } m \geq m_0, \text{ which is impossible.} \]

\( f(n,m) \leq cg(n,m) \) is always violated for certain value of \( n \geq n_0 \) and \( m \geq m_0 \).

Hence \( f(n,m) \neq \mathcal{O}(g(n,m)) \) and \( g(n,m) \neq \mathcal{O}(f(n,m)) \)
Bounding by Integrals

[CLRS Appendix A]

Let \( f(k) \) be monotonically increasing for nonnegative \( k \). Then

\[
\int_{m-1}^{n} f(x) \, dx \leq \sum_{k=m}^{n} f(k) \leq \int_{m}^{n+1} f(x) \, dx
\]
Example:

Let $g(j) = j^2$. Then

$$
\int_0^n g(t) \, dt \leq \sum_{j=1}^n g(j) \leq \int_1^{n+1} g(t) \, dt
$$

$$
\int_0^n t^2 \, dt \leq \sum_{j=1}^n j^2 \leq \int_1^{n+1} t^2 \, dt
$$

$$
n^3/3 \leq \sum_{j=1}^n j^2 \leq ((n + 1)^3 - 1) / 3
$$

Hence

$$
\sum_{j=1}^n j^2 = \Theta(n^3)
$$
People who analyze algorithms have double happiness.

First of all they experience the sheer beauty of elegant mathematical patterns that surround elegant computational procedures.

Then they receive a practical payoff when their theories make it possible to get other jobs done more quickly and more economically.

D.E. Knuth
Introduction to Complexity

Reference:
• CLRS Chapters 34, 35
• GJ 1979, “Computers and Intractability”
  (or Online Annotated List of selected NP-complete Problems
   http://www.csc.liv.ac.uk/~ped/teachadmin/COMP202/annotated_np.html)

Objectives:
• To introduce the theory of NP-Completeness and the technique of reduction in proving hardness
• To surface classical NP-complete problems and their embedding in Supply Chain problems
• To understand how to cope with NP-Complete problems
Introduction

• Some problems are *intractable* (hard)
  – as they grow large, we are unable to solve them in “reasonable” time

• What constitutes reasonable time? In this course, it means *polynomial time*
  – On an input of size $n$, the worst-case running time is $O(n^k)$ for some constant $k$
  – Polynomial time: $O(n^2)$, $O(n^3)$, $O(1)$, $O(n \log n)$, etc
  – Not in polynomial time: $O(2^n)$, $O(n^{\log n})$, $O(n!)$, $O(nL)$, etc
Introduction

• We define P to be the class of problems solvable in polynomial time

• Are all problems solvable in polynomial time?
  – No: Turing’s “Halting Problem” is not solvable by any computer, no matter how much time is given
  – Such problems are clearly intractable, not in P

• *NP-Complete* problems are an interesting class of problems whose status is unknown
  – No polynomial-time algorithm has been discovered for an NP-Complete problem
  – No superpolynomial lower bound has been proved for any NP-Complete problem, either
Introduction

• We call this the $P = NP$ question
  – The biggest open problem in CS
• Difference between P and NP-Complete problems appear to be slight
  – 2SAT vs 3SAT, 2-COLOR vs 3-COLOR, 2FS vs 3FS, Matching vs 3DMatching
  – Euler tour vs Hamiltonian cycle
  – Shortest vs Longest path problem
Decision vs Optimization Problems

- **Optimization problems**
  - Each feasible solution has an objective value
  - Goal: to find the feasible solution with the optimal (min/max) value

- **Decision problems** are “yes/no” problem

- **Example. PATH vs Shortest Path**
  - **PATH**: Given G, u, v and integer k, is there a path from u to v with distance at most k?
  - **Shortest Path**: Given G, u, v, find a path from u to v with minimum distance.

- Strictly speaking, NP-Completeness applies to decision problems and *not* directly to optimization problems

- Decisions problems are “easier” (“no harder”) than the corresponding optimization problems. Why?
Complexity Classes P and NP

- **P** is class of problems that can be solved in polynomial time
- **NP** (nondeterministic polynomial time) is the class of problems that can be solved in polynomial time by a nondeterministic computer
- For Algorithms people, **NP** refers to the class of problems that can be verified by a polynomial-time algorithm.
Verification

If you tell me that this graph is 3-colorable,

it is very difficult for me to check whether you are right.
But if you tell me that this graph is 3-colorable and give me a solution, it is very easy for me to verify whether you are right.
Hamiltonian Cycle

• A *hamiltonian cycle* of an undirected graph is a simple cycle that *contains every vertex*

• Hamiltonian-cycle problem (**HAM**): Given a graph G, does it have a hamiltonian cycle?

• How might *a naïve algorithm* solves **HAM**?
Verification

- Your friend tells you that a given $G$ is hamiltonian and gives you a “proof” (the vertices along a hamiltonian cycle)
- This proof is also called a certificate
- How to verify the “proof” in polynomial time?
- Verification algorithm $A$(instance $x$, certificate $y$):
  - For any instance $x$,
    a. if $x$ yields a “yes” answer, $A(x,y)$ returns 1
    b. if $x$ yields a “no” answer, then $A(x,y)$ returns 0 for all $y$.
- A problem is in class $\text{NP}$ iff there exists a polynomial-time verification algorithm.
P and NP

• Summary so far:
  – **P** = problems that can be *solved* in polynomial time
  – **NP** = problems which can be *verified* in polynomial time
  – Is **P** a subset of **NP**?
  – Unknown whether **P** = **NP** (most suspect not)

• **HAM** is in **NP**:
  – Cannot *solve* in polynomial time
  – Can *verify* “solution” in polynomial time
Want to identify the “hardest” problems in a class such as NP.

If these problems have deterministic polynomial-time solutions then all problems in that class do.

Reduces the $P = NP$ question to that of whether one of these “hard” problems is in $P$.

**Definitions:** Let $C$ be a complexity class. L is hard for $C$ if every problem in $C$ is polynomially transformable to L. L is complete for $C$ if it is (1) in $C$ and (2) hard for $C$. 
NP-Complete Problems

• NP-Complete problems are the “hardest” problems in NP:
  – If any *one* NP-Complete problem can be solved in polynomial time…
  – …then *every* NP-Complete problem can be solved in polynomial time…
  – …and in fact *every* problem in \( \text{NP} \) can be solved in polynomial time (which would show \( \text{P} = \text{NP} \))
Reduction

• The crux of NP-Completeness is *reducibility*
• Informally, a problem X can be reduced to another problem Q if:
  – *any* instance of X can be “easily modelled” as an instance of Q,
  – the solution to the latter provides a solution to the former and vice versa
• Intuitively: If X reduces to Q, X is “no harder to solve” than Q
Reduction

• Let \( L_1 \) and \( L_2 \) be 2 decision problems.

• \( L_1 \) is polynomial time reducible to \( L_2 \), written \( L_1 \leq_p L_2 \) when there is a function \( f \) that maps \( x \), an instance of the problem \( L_1 \) into \( f(x) \) in \( L_2 \) in polynomial time s.t. \( x \in L_1 \) iff \( f(x) \in L_2 \)

• \( f \) is called a reduction algorithm.

• If \( L_2 \) can be solved in polynomial time then \( L_1 \) can be solved in polynomial time
Reduction

1. Composition: If $L_1 \leq_p L_2$ and $L_2 \leq_p L_3$ then $L_1 \leq_p L_3$

2. $L_1 \leq_p L_2$ means that $L_2$ is harder (or equally hard) to solve.

3. Problems solvable in polynomial time are considered easy.

4. All problems in $\mathbf{P}$ are easy, so hard problems can only be found in $\mathbf{NP}$ or beyond $\mathbf{NP}$. 
Formal Definition of NP-Completeness

- A problem (language) $L$ is **NP-hard** if $L' \leq_p L$, for all $L' \in \text{NP}$

- $L$ is **NP-complete (NPC)** if
  1. $L$ is NP-hard
  2. $L \in \text{NP}$

- So NP complete problems are the **hardest problems in NP**.
- If $L' \leq_p L$ and $L'$ is NP-Complete, $L$ is also NP-Complete
Theorem 34.4 (CLRS)
1. If any NPC problem $\in$ P, then $P=NP$
2. If any NP problem is not in P, then all NPC problems are not in P.


Most people believe that $P \not= NP$, i.e., $P \nsubseteq \not\in NP$
Proving NP-Completeness

To prove that $X$ is NP-complete:

1. Prove that $X \in \text{NP}$;
2. Select a known NP-complete problem $Q$;
3. Reduce $Q$ to $X$:
   a. Describe a reduction $f$ that maps instances of $Q$ to instances of $X$, s.t. “yes” for $X = “yes”$ for $Q$ (so if we had a poly-time solver for $X$, then we could use it to solve $Q$ in polynomial time)
   a. Prove that the $f$ runs in polynomial time.
Example: Traveling Salesman Problem \textbf{TSP}

- **Optimization problem**: Given a weighted graph $G$, find a hamiltonian cycle with the minimum weight.
- **Decision problem**: Given $G$ and integer $k$, does $G$ have a hamiltonian cycle with cost $k$?
Example: Traveling Salesman Problem $\text{TSP}$

To prove $\text{TSP}$ is NP-Complete:

1. Prove that $\text{TSP} \in \text{NP}$
2. Pick $\text{HAM}$
3. Reduction $\text{HAM} \leq_p \text{TSP}$
   a. maps an instance of $\text{HAM}$ to an instance of $\text{TSP}$
      s.t. “yes” for $\text{TSP}$ = “yes” for $\text{HAM}$
   b. Can we do this mapping in polynomial time?
Family of NP-Complete Problems

• Given one NP-Complete problem, we can prove many interesting problems NP-Complete using reduction:
  – 3-COLORING: can a given graph be colored with 3 colors such that no adjacent vertices are the same color?
  – SUBSET SUM: given a set of integers, does there exist a subset that adds up to some target \( T \)?
    – Vertex Cover
    – Clique
    – Set Cover
    – 0/1 Knapsack problem
    – Traveling salesman
    – Job scheduling, etc, etc
The **SAT** Problem

- One of the first problems to be proved NP-Complete is *Satisfiability (SAT)*:

  Input: a Boolean expression on \( n \) variables
  Question: is there an assignment such that the expression is TRUE?

  Example: \(((x_1 \rightarrow x_2) \lor \neg((\neg x_1 \leftrightarrow x_3) \lor x_4)) \land \neg x_2\)

- **Cook’s Theorem:** **SAT** is NP-Complete
  - Note: Argue from first principles, not reduction
  - Proof: Very difficult!
Conjunctive Normal Form

• Even if the form of the Boolean expression is simplified, the problem is still NP-Complete:
  – *Literal*: an occurrence of a Boolean or its negation
  – A Boolean formula is in *conjunctive normal form*, or *CNF*, if it is an AND of clauses, each of which is an OR of literals
    • Ex: \((x_1 \lor \neg x_2) \land (\neg x_1 \lor x_3 \lor x_4) \land (\neg x_5)\)
  – *3-CNF*: each clause has exactly 3 distinct literals
    • Example: \((x_1 \lor \neg x_2 \lor \neg x_3) \land (\neg x_1 \lor x_3 \lor x_4) \land (\neg x_5 \lor x_3 \lor x_4)\)
    • Notice: TRUE iff at least one literal in each clause is true
  – *3SAT*: Given a 3-CNF expression, is it satisfiable?
The $3\text{SAT}$ Problem

- Thm 34.10 (CLRS): $3\text{SAT}$ is NP-Complete
  - Proof: Too complicated!!
- The reason we care about $3\text{SAT}$ is that it is relatively easy to reduce to others
- Thus by proving $3\text{SAT}$ is NP-Complete we can prove many seemingly unrelated problems NP-Complete
**Clique**

- A clique of a graph G: a subset of vertices fully connected to each other, i.e. a complete subgraph of G
- **Clique**: Given a graph G and integer k, is there a clique of size k?

- Thm 34.11 (CLRS): **Clique** is NP-Complete
  1. Is **Clique** in NP?
  2. Reduction **3SAT** $\leq_p$ **Clique**
     - Transform a 3-CNF formula to a graph, for which a $k$-clique will exist (for some $k$) iff the 3-CNF formula is satisfiable
3SAT $\leq_p$ Clique

• The reduction:
  – Let $B = C_1 \land C_2 \land \ldots \land C_k$ be a 3-CNF formula with $k$ clauses, each of which has 3 distinct literals
  – For each clause put a triple of vertices in the graph, one for each literal
  – Put an edge between two vertices if they are in different triples and their literals are consistent, meaning not each other’s negation
Example

\[ B = (x_1 \lor x_2 \lor x_3) \land (\overline{x}_1 \lor \overline{x}_2 \lor \overline{x}_3) \land (\overline{x}_1 \lor x_2 \lor x_3) \]
$3SAT \leq_p \text{Clique}$

2. given a k-clique, a satisfying assignment can be formed

B = 3-CNF expression

1. given a satisfying assignment, a k-clique can be formed
3SAT \leq_p \text{Clique}

Proof:

1. If B has a satisfying assignment, then each clause has at least one literal (vertex) that evaluates to 1. Picking one such “true” literal from each clause gives a set \( V' \) of \( k \) vertices. \( V' \) is a clique (Why?)

2. If G has a clique \( V' \) of size \( k \), it must contain one vertex in each triple (Why?)
   We can assign 1 to each literal corresponding with a vertex in \( V' \), without fear of contradiction
Expanding the Catalogue

CIRCUIT SAT

SAT

3SAT

CLIQUE

3SAT

VERTEX COVER

HAM

TSP

SET COVER

NP
General Comments

• Literally hundreds of problems have been shown to be NP-Complete
• Handbook: [Garey and Johnson 78]
• Some reductions are profound, some are comparatively easy, many are easy once the key insight is given
Coping with NP-Completeness

We have 3 options:

1. Give up on “polynomial time”
   Algorithms that give optimal answers to all instances, but may take exponential time in the worst case.

- Exhaustive search procedures, e.g. math programming (e.g. branch-&-bound), dynamic programming, constraint programming.
- The best of these do well very often and only run for too long occasionally.
Coping with NP-Completeness

2. Give up on “all instances”
   Algorithm runs in poly-time and gives the optimal answer, but not for *any* input.

Two categories:
- **Special case algorithms:** Some hard problems are easy on special types of graphs, for example.
- **Randomized algorithms:** Some problems have poly-time algs with coin flips. None known to be NP-hard, but some lack known poly-time algorithms with no coin flips.
Coping with NP-Completeness

3. Give up on “optimal”
   Poly-time algorithms for all instances producing solutions which are “near-optimal”

Two categories:
• **Approximation algorithms** (proven performance)
• **Heuristics, Meta-heuristics** (usually no proofs)
Performance Measurements

What do we expect from a “good” heuristic:

- It should do so quickly
- It should find a solution of good quality

Tools to measure heuristic performances:

1. Run time analysis
   - number of steps the algorithm takes with respect to input size
2. Analysis on solution quality (performance guarantee)
   - we’ll briefly discuss in this lecture - to find out more, read CLRS Chapter 35
Performance Guarantee

• Consider an NP-hard minimization problem P and a (polynomial-time) algorithm A to solve P

• For an instance I of P:
  – A(I) = cost of solution returned by A on instance I
  – OPT(I) = cost of optimal solution of instance I

• Define $R_A(I) = A(I)/OPT(I)$

• Worst-case approximation ratio $R_A$ is defined by
  $$R_A = \max_I \{R_A(I)\}$$

• What’s the range of $R_A$?

• An algorithm A for P is an approximation algorithm if, for all instances, A returns a solution whose value is most a given factor away from the value of the optimal solution
Performance Guarantee

• This value provides us with an upper bound on the performance of algorithm $A$.

• That is, the solutions returned by $A$ are at most $R_A$ times larger than the optimal solution.

• For maximization problem: we take min() instead of max()

• What’s now the range of $R_A$?

• Note: CLRS uses $R_A(I) = OPT(I)/A(I)$
Approximation Algorithms for TSP

- Given a graph $G$ with positive weights, find a shortest tour which visits all vertices.

- **Nearest Neighbor** heuristic worst-case ratio:
  \[ R_{NN} \leq \frac{1}{2} \left( 1 + \log_2 n \right) \]

- The ratio grows with $n$

  - e.g. for $n = 10$: $R_{NN} \leq 2.16$
  - $n = 1000$: $R_{NN} \leq 5.48$
2-approximation TSP

- **Constant** approximation ratio
- **Assume** triangle inequality
  \[ w(a,b) + w(b,c) \geq w(a,c) \]  // when is this realized?
- **TSP is NP-complete, even if the cost function satisfy triangle inequality**
- **MST algorithm:**
  1. find a MST of G rooted at arbitrary r, T
  2. let L be list of vertices visited in pre-order walk of T
  3. return tour that visits the vertices in the order L

- Example: CLRS Figure 35.2
Approximation Algorithm –
Bin Packing Problem

Given: \( n \) items \( a_1, a_2, ..., a_n \), \( 0 < \text{size}(a_i) \leq 1 \) \( \forall \ i = 1,..,n \)
unlimited number of bins of size 1
Goal: Pack all items into a minimum number of bins

Optimal Packing

\[ N_0 = 4 \]
Approximation Algorithm – Bin Packing Problem

Optimal Packing

\[ N_0 = 4 \]

\[ \frac{N}{N_0} \leq 2 \]

Next Fit Packing Algorithm

\[ N = 6 \]
Approximation Algorithm – Bin Packing Problem

Next Fit Packing Algorithm

First Fit Packing Algorithm

\[ \frac{N}{N_0} \leq 1.7 \]
Approximation Algorithm –
Bin Packing Problem

Let $a_1 + a_2 + \ldots > \Sigma$

$2 \Sigma \geq N - 1$

$N_0 \geq \Sigma \geq \frac{N - 1}{2} \geq \frac{N}{2}$

$\frac{N}{N_0} \leq 2$
Next Week

Self-Study (Assessment) Week:

Read CLRS Chapters 6-9 and do Assignment 1