NP-Completeness

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

Objectives:
• To understand the theory of NP-Completeness
• To learn the technique of reduction in proving NP-Completeness
• To appreciate why NP-Completeness is an important consideration in Algorithm Design and Analysis
• A philosophical conjecture: Creation is much harder than Verification.

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 \( P \).
- \( NP\text{-Complete} \) problems are an interesting class of problems whose status is unknown.
  - No polynomial-time algorithm has been discovered for an \( NP\text{-Complete} \) problem.
  - No superpolynomial lower bound has been proved for any \( NP\text{-Complete} \) problem, either.

Introduction

- We call this the \( P = NP \) question.
  - The biggest open problem in CS.
- Difference between \( P \) and \( NP\text{-Complete} \) 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.
Shortest Path vs Longest Path

- **Optimal Sub-structure**
  - Optimal sub-structure exists for the Shortest Path problem:
    - consider shortest path between A and D passing through B
    - Paths from A to B and B to D are both optimal. Why?
  - Optimal sub-structure does **not** exist for Longest Path problem:

![Graph showing Shortest Path vs Longest Path](image)

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.
Verification

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?
- Examples: CLRS Figure 34.2
- 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 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

---

**Hard and Complete Problems**

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 NP can be solved in polynomial time (which would show P = NP)

Reduction

• The crux of NP-Completeness is *reducibility*
• Informally, a problem P can be reduced to another problem Q if :
  – *any* instance of P 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 P reduces to Q, P 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. $f$ is called a reduction algorithm.

- If $L_2$ can be solved in polynomial time then $L_1$ can be solved in polynomial time

\[ \begin{array}{c}
\text{Polynomial-time reduction alg. } f \\
\text{Polynomial-time alg. to solve } L_2 \\
\text{Polynomial-time algorithm to solve } L_1
\end{array} \]

\[ \begin{array}{c}
\text{yes} \\
\text{yes} \\
\text{no} \\
\text{no}
\end{array} \]

Reduction

1. 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 $P$ are easy, so hard problems can only be found in $NP$ or beyond $NP$.

5. Polynomial complexities are used frequently because they are closed under function composition, sum, difference and product.
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 \( \text{NP} \).
• If \( L' \leq_p L \) and \( L' \) is NP-Complete, \( L \) is also NP-Complete

P = NP?

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 \( P \neq NP \), i.e., \( P \subset \not\subset \text{NP} \)
Why Bother to Prove NP-Completeness?

• Though nobody has proven that $P \neq NP$, if you prove a problem NP-Complete, most people accept that it is probably intractable.
• Once proven that a problem is NP-Complete:
  – Stop thinking about developing a polynomial-time algorithm to solve the problem exactly.
  – Can instead work on approximation algorithms.
  – Or work on exact problems with clever branching or pruning techniques (e.g. branch and bound, A*).
• Solve HAM in $O(n^{100})$ time, you’ve proved that $P = NP$.
  – Turing Award (http://www.acm.org/awards/taward.html), Von Neumann Price (http://www.informs.org/Prizes/vonNeumannDetails.html) etc awaits you!

Proving NP-Completeness

To prove that $P$ is NP-complete:

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

To prove TSP is NP-Complete:
1. Prove that TSP $\in$ NP
2. Pick HAM
3. Reduction HAM $\leq_p$ TSP
   a. maps an instance of HAM to an instance of TSP s.t.
   "yes" for TSP = "yes" for 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 3SAT Problem

• Thm 34.10 (CLRS): 3SAT is NP-Complete
  – Proof: Too complicated!!
• The reason we care about 3SAT is that it is relatively easy to reduce to others
• Thus by proving 3SAT 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
\[ B = (x_1 \lor x_2 \lor x_3) \land (\overline{x}_1 \lor \overline{x}_2 \lor \overline{x}_3) \land (x_1 \lor x_2 \lor x_3) \]

3SAT \leq_p 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.

Clique \leq_p \text{Vertex Cover}

- A vertex cover for a graph \( G \) is a subset of vertices incident to every edge in \( G \)
- Vertex Cover: Given graph \( G \) and integer \( k \), does \( G \) have a vertex cover of size \( k \)?

\[
\begin{align*}
\{1,4,5\} & \text{ is a VC} \\
\{2,3\} & \text{ is a min VC}
\end{align*}
\]

- Thm 34.12 (CLRS): Vertex Cover is NP-Complete
  1. Show Vertex Cover is in \( \text{NP} \)
  2. Reduction: Clique \leq_p \text{Vertex Cover}
     - The complement \( G_c \) of a graph \( G \) contains exactly those edges not in \( G \)
     - Can compute \( G_c \) in polynomial time
     - Prove \( G \) has a clique of size \( k \) iff \( G_c \) has a vertex cover of size \( |V| - k \)
Clique \leq_p Vertex Cover

- Claim: If G has a clique of size k, G_C has a vertex cover of size |V|-k.
  (Let V' be the k-clique. Then V-V' is a vertex cover in G_C)

- Proof:
  - Let (u,v) be any edge in G_C
  - Then u and v cannot both be in V' \((Why?)\)
  - Thus at least one of u or v is in V-V' \((Why?)\), so edge (u, v) is covered by V-V'
  - Since true for any edge in G_C, V-V' is a vertex cover for G_C

Clique \leq_p Vertex Cover

- Claim: If G_C has a vertex cover V' \subseteq V, with |V'| = |V|-k, then G has a clique of size k

- Proof:
  - For all u, v \in V,
    if (u,v) \in G_C then u \in V' or v \in V' or both \((Why?)\)
  - Contrapositive: if u \notin V' and v \notin V', then (u,v) \in G
  - In other words, all pairs of vertices in V-V' are connected by an edge in G, thus V-V' is a clique in G
  - Since |V|-|V'| = k, the size of the clique is k
Expanding the Catalogue

CIRCUIT SAT

SAT

3SAT

CLIQUE

HAM

VERTEX COVER

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

- Your boss asks you to implement an algorithm for a problem that you know is NP-complete. What do you do?

- Tell him that the whole world of computer scientists have not found an elegant algorithm
- Buy a faster machine, and learn to be much more patient!
- Try to find a polynomial time algorithm to solve it (and hence prove that P=NP)!
- Look for approximations, heuristics, etc
- Give up, and look for a new job!