# NP-Completeness

## P: Polynomial-Time Solvable Problems

Algorithms: Design and Analysis, Part II

# Ubiquitous Intractability

Focus of this course (+ Part I): Practical algorithms + supporting theory for fundamental computational problems.

Sad fact: Many important problems seem impossible to solve efficiently.

Next: How to formalize computational intractability using NP-completeness.

Later: Algorithmic approaches to NP-complete problems.

# Polynomial-Time Solvability

Question: How to formalize (in)tractability?

Definition: A problem is polynomial-time solvable if there is an algorithm that correctly solves it in $O(n^k)$ time, for some constant $k$.

[Where $n$ = input length = # of key strokes needed to describe input]

[Yes, even $k = 10,000$ is sufficient for this definition]

Comment: Will focus on deterministic algorithms, but to first order doesn't matter.

# The Class P

Definition: P = the set of poly-time solvable problems.

Examples: Everything we've seen in this course except:

- Cycle-free shortest paths in graphs with negative cycles

- Knapsack [running time of our algorithm was $\Theta(nW)$, but input length proportional to $\log W$]

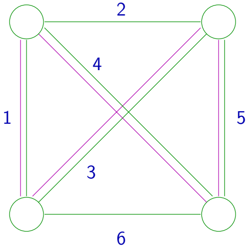
Both problems are NP-complete

Interpretation: Rough litmus test for "computational tractability".

# Traveling Salesman Problem

Input: Complete undirected graph with nonnegative edge costs.

Output: A min-cost tour [i.e., a cycle that visits every vertex exactly once].




$OPT = 13$

Conjecture: [Edmonds '65] There is no polynomial-time algorithm for TSP.

[As we'll see, equivalent to P$\neq$NP]

# NP-Completeness

Reductions and Completeness

Algorithms: Design and Analysis, Part II

# Reductions

Conjecture: [Edmonds '65] There is no polynomial-time algorithm that solves the TSP. [Equivalent to P$\neq$NP]

Really good idea: Amass evidence of intractability via relative difficulty - TSP "as hard as" lots of other problems.

Definition: [A little informal] Problem $\Pi_1$ reduces to problem $\Pi_2$ if: given a polynomial-time subroutine for $\Pi_2$, can use it to solve $\Pi_1$ in polynomial time.

# Quiz

Which of the following statements are true?

A) Computing the median reduces to sorting

B) Detecting a cycle reduces to depth-first search

C) All pairs shortest paths reduces to single-source shortest paths

D) All of the above

# Completeness

Suppose $\Pi_1$ reduces to $\Pi_2$.

Contrapositive: If $\Pi_1$ is not in P, then neither is $\Pi_2$.

That is: $\Pi_2$ is at least as hard as $\Pi_1$.

Definition: Let $\mathcal{C}$ = a set of problems.

The problem $\Pi$ is $\mathcal{C}$-complete if:

(1) $\Pi \in \mathcal{C}$ and (2) everything in $\mathcal{C}$ reduces to $\Pi$.

That is: $\Pi$ is the hardest problem in all of $\mathcal{C}$.

# Choice of the Class $\mathcal{C}$

Idea: Show TSP is $\mathcal{C}$-complete for a REALLY BIG set $\mathcal{C}$.

How about: Show this where $\mathcal{C}$ = ALL problems.

Halting Problem: Given a program and an input for it, will it eventually halt?

Fact: [Turing '36] No algorithm, however slow, solves the Halting Problem.

Contrast: TSP definitely solvable in finite time (via brute-force search).

Refined idea: TSP as hard as all brute-force-solvable problems.

# NP-Completeness

## Definition and Interpretation

Algorithms: Design and Analysis, Part II

# The Class NP

Refined idea: Prove that TSP is as hard as all brute-force-solvable problems.

Definition: A problem is in NP if:

(1) Solutions always have length polynomial in the input size

(2) Purported solutions can be verified in polynomial time.

Examples: - Is there a TSP tour with length $\leq 1000$?

- Constraint satisfaction problems (e.g., 3SAT)

# Interpretation of NP-Completeness

Note: Every problem in NP can be solved by brute-force search in exponential time. [Just check every candidate solution.]

Fact: Vast majority of natural computational problems are in NP [$\approx$ Can recognize a solution]

By definition of completeness: A polynomial-time algorithm for one NP-complete problem solves every problem in NP efficiently [i.e., implies that P=NP]

Upshot: NP-completeness is strong evidence of intractability!

# A Little History

Interpretation: An NP-complete problem encodes simultaneously all problems for which a solution can be efficiently recognized (a "universal problem").

Question: Can such problems really exist?

Amazing fact #1: [Cook '71, Levin '73] NP-complete problems exist.

Amazing fact #2: [started by Karp '72] 1000s of natural and important problems are NP-complete (including TSP).

# NP-Completeness User's Guide

Essential tool in the programmer's toolbox: The following recipe for proving that a problem $\Pi$ is NP-complete.

(1) Find a known NP-complete problem $\Pi'$ (see e.g. Garey + Johnson, Computers + Intractability)

(2) Prove that $\Pi'$ reduces to $\Pi$

$\Rightarrow$ implies that $\Pi$ at least as hard as $\Pi'$

$\Rightarrow$ $\Pi$ is NP-complete as well (assuming $\Pi$ is an NP problem)


# NP-Completeness

## The P vs. NP Question

Algorithms: Design and Analysis, Part II

# The P vs. NP Question

Question: Is P = NP ?

polynomial time solvable

can verify correctness of a solution in polynomial time

Widely conjectured: P≠NP. [Though see Gödel '56]

But: Has not been proved. [Worth $1 million from Clay Institute]

Reasons to believe:

(1) (psychological) if P=NP, someone would have proved it by now

(2) (philosophical) if P=NP, then finding a proof always as easy as verifying one

(3) (mathematical) ??

# What's In A Name

FAQ: What does "NP" stand for?

~~"not polynomial"~~

Answer: "Nondeterministic polynomial"

[Modern, mathematically equivalent definition via efficient verification of purported solutions]

Historical reference: Knuth, "A Terminological Proposal", 1974.

Passed over:

PET → "possibly exponential time"
PET → "provably exponential time"
PET → "previously exponential time"

# NP-Completeness: The Beginning, Not the End

Question: So your problem is NP-complete. Now what?

Important: NP-completeness not a death sentence.

$\Rightarrow$ but, need appropriate expectations/strategy

Three useful strategies:

(1) Focus on computationally tractable special cases

Examples: - WIS in path graphs (and trees, bounded tree width) (NP-c in general graphs)

- Knapsack with polynomial size capacity (e.g., $W = O(n)$)

- 2SAT (P) instead of 3SAT (NP-c)

- Vertex cover when OPT is small

# Three Useful Strategies (con'd)

(2) Heuristics - fast algorithms that are not always correct

Examples (forthcoming): Greedy and dynamic programming-based heuristics for knapsack.

(3) Solve in exponential time but faster than brute-force search.

- Knapsack ($O(n)$ instead of $2^n$)
- TSP ($\approx 2^n$ instead of $\approx n!$) (forthcoming)
- Vertex cover ($\approx 2^{\mathrm{OPT}} n$ instead of $n^{\mathrm{OPT}}$) (forthcoming)