COS 226 Lecture 23: NP-completeness

For a large class of important problems
    no algorithms are known
    that are guaranteed to be fast

EFFICIENT, TRACTABLE
    running time bounded by polynomial
    on input size, no matter what the input

INEFFICIENT, INTRACTABLE
    running time exponential in input size
    on some input

Can't guarantee solution to intractable problems
    even with a supercomputer!!
* tough to explain to the boss...
* source of frustration for early programmers
* theory provides some consolation

Includes a very large class of natural problems
Fundamental issue in the theory of computation

Some Important (But Hard) Problems

* TRAVELING SALESMAN

  A salesman needs to visit a set of N cities. Find a route that minimizes travel distance.

* SCHEDULING

  A set of jobs of varying length need to be done on two identical machines before a certain deadline. Can the jobs be arranged so that the deadline is met?

* SEQUENCING

  A set of four-character fragments have been obtained by breaking up a long string into overlapping pieces. Can the fragments be reconstituted into the long string?

* TRIPARTITE MATCHING

  Given three sets of N individuals, and a set of triples with one from each set, is there a subset of the triples containing each individual just once?

* SATISFIABILITY

  Is there a way to assign truth values to a given logical formula that makes it true?

Polynomial Time Algorithms

P:

The set of all problems solvable by deterministic algorithms in polynomial time

Covers virtually all programs running on actual computers

Polynomial-time algorithms are closed under composition

* can use one polynomial-time algorithm as subroutine for another
* can simulate one computer with polynomial-time algorithm on another

Polynomial-time characteristic independent of particular computer and implementation

Includes Turing machines, parallel supercomputers, and everything in between

Nondeterministic algorithms

NP:

The set of all problems solvable by non-deterministic algorithms in polynomial time.

If a machine can guess (and is lucky), it can solve a problem in NP quickly (guess the right answer).

Alternate view: for a problem in NP, we can CHECK that a putative solution is actually the solution in polynomial time on a conventional computer.

Real computers can simulate Lucky Guessing, in exponential time, by checking every possibility.

The main question

What is the relationship between P and NP?

P is contained in NP
    don’t have to use nondeterminism
    to solve in polynomial time

NP – P: intractable problems
    can’t be solved in polynomial time
    except by using nondeterminism

Convenient way to classify problems

For many problems, no solution better
than trying every possibility is known
    BUT no one has been able to PROVE that any
    problem is in NP – P (!!)

IS P=NP?

Nondeterminism seems powerful
    BUT no one has been able to PROVE
    that it helps for any particular problem

NP-completeness

NP-COMPLETE

A problem in NP is said to be NP-complete
if the existence of a polynomial-time solution
for it implies that P=NP.

If an NP-complete problem can be solved in
polynomial time, then so can *all* problems in NP.

POLYNOMIAL-TIME REDUCTION

Reduce problem A to problem B by

* converting instance of A to instance of B
* solving instance of B
* convert solution of B to solution of A

THM: If A is NP-complete, A reduces to B
(and B is in NP) then B is NP-complete

Proof: if B can be solved efficiently,
then so can A which implies that P=NP.

Thousands of problems have been shown to be
NP-complete (including traveling salesperson).

If any one of these important problems can be
solved efficiently, they all can.
(Moreover, so can any problem in NP).

Still need direct proof that some particular
problem is NP-complete

Cook’s Theorem

SATISFIABILITY is NP-complete.

Outline of proof:

Non-deterministic Turing machines can
solve problems in NP.

Logical formulas can describe Turing machines
including non-deterministic ones

Given any problem in NP
establish a correspondence
with some instance of SATISFIABILITY

SATISFIABILITY solution
gives simulation of Turing machine
solving the corresponding problem

If SATISFIABILITY can be solved quickly,
then so can any problem in NP.

Any problem reducible to SATISFIABILITY
is NP-complete, etc.

Theoretical approaches to the TSP

THM: TSP on graphs is NP-complete

THM: No polynomial-time approximation algorithm can guarantee to find a tour within a constant factor of the optimum unless P = NP

Triangle inequality:
A to C is no longer than A to B to C

THM: Tour within a factor of 2 of optimum can be found in N lg N steps, if the triangle inequality is satisfied.
Proof: Compute MST, traverse, skip vertices already visited.

THM: Tour within a factor of 3/2 of optimum can be found in polynomial time, if the triangle inequality is satisfied

Euclidean TSP:
points in plane, Euclidean metric distances

THM: Euclidean TSP is NP-complete

THM: Tour within a factor of 1/x of optimum can be found in N^x steps, for Euclidean TSP

Practical approaches to the TSP

Brute force: N!
* try all possibilities

Dynamic programming: N 2^N
* build tables of tours for subsets of nodes

Branch-and-bound
* keep track of best tour found so far
* stop looking if current path can't lead
  to a better tour

Local optimization
* random tour, eliminate crossing
* random tour, improve all sets of three edges

...

Best tours for some problems with thousands of nodes have been found.

TSP WEB page
* standard sample problems
* code with tuned heuristics

Implications of NP-completeness

EITHER:

Real machines can do as well as machines endowed with nondeterminism, but we don't know how to make them do so. (P=NP)

OR:

Nondeterminism DOES help, but is a fiction on real machines, and no NP-complete problem can be solved in polynomial time. (P!=NP)

Virtually no one believes that P=NP

...but it's possible.

Proof that a problem is NP-complete is usually taken as a signal to abandon hope of finding an efficient "solution".

Guaranteed polynomial-time performance may be too stringent a requirement

May not need "solution" in practical applications

* Slow algorithm may be fast on real problems (cf. simplex!)
* Running time may be between polynomial and exponential
* Approximate solution may suffice
* Randomization may help
* Known fast algorithms may help