IE 607 Heuristic Optimization

Introduction to Optimization
Part II

Algorithm Design & Analysis

Steps:

- Design the algorithm
- Encode the algorithm
    - Steps of algorithm
    - Data structures
- Apply the algorithm

Algorithm Efficiency

- Run-time
- Memory requirements
- Number of elementary computer operations to solve the problem in the worst case

Study increased in the 70's

Complexity Analysis

- Seeks to classify problems in terms of the mathematical order of the computational resources required to solve the problems via computer algorithms
- Judge a problem whether we can find a polynomial-time algorithm to solve it
- Decide the “right” approach to solve a problem

- **Problem** is a collection of instances that share a mathematical form but differ in size and in the values of numerical constants in the problem form (i.e. *generic model*).

  Example: shortest path.
- **Instance** is a special case of problem with specified data and parameters.
- An algorithm *solves* a problem if the algorithm is guaranteed to find the optimal solution for any instance.

Complexity Measures

- **Empirical Analysis**
  - see how algorithms perform in practice
  - write program, test on classes of problem instances
- **Average Case Analysis (Expected Case Analysis)**
  - determine expected number of steps
  - choose probability distribution for problem instances, use statistical analysis to derive asymptotic expected run times

Complexity Measures (cont.)

- **Worst Case Analysis**
  - provides upper bound (UB) on the number of steps an algorithm can take on *any* instance
  - count largest possible number of steps
  - provides a “guarantee” on number of steps the algorithm will need

Complexity Measures (cont.)

- **CONs of Empirical Analysis**
  - algorithm performance depends on computer language, compiler, hardware, programmer's skills
  - costly and time consuming to do
  - algorithm comparison can be inconclusive

Complexity Measures (cont.)

- **CONs of Average Case Analysis**
  - depends heavily on choice of probability distribution
  - hard to pick the probability distribution of realistic problems
  - analysis often very complex
  - assumes analyst solving multiple problem instances

Complexity Measures (cont.)

- **Worst Case Analysis**

  PROs

  - independent of computing environment

  - relatively easy

  - guarantee on steps (time)

  - definitive

  CONs

  - simplex method exception

  - algorithm comparisons can be inconclusive

Big "O" Notation

- A theoretical measure of the execution of an algorithm given the problem size n.
- An algorithm is said to run in O(g(n)) time if f(n) = O(g(n)) (of order g(n)) and there are positive constants c and k, such that $0 \le f(n) \le cg(n)$ for all $n \ge k$ (i.e. the time taken by the algorithm is at most cg(n) for all $n \ge k$).

"f of n is big o of g of n"

Big "O" Notation (cont.)

- Usually interested in performance on large instances
- Only consider the dominant term

Example: $100n + 1000n^2 + 0.01n^3$

→ $O(n^3)$

Polynomial vs Exponential-Time Algorithms

- What is a “good” algorithm?
- It is commonly accepted that worst case performance bounded by a polynomial function of the problem parameters is “good”. We call this a *Polynomial-Time Algorithm*.

  Example: $O(n^3)$
- Strongly preferred because it can handle arbitrarily large data

Polynomial vs Exponential-Time Algorithms (cont.)

- In Exponential-Time Algorithms, worst case run time grows as a function that cannot be polynomially bounded by the input parameters.

  Example: $O(2^n)$ $O(n!)$
- Why is a polynomial-time algorithm better than an exponential-time one?

  →Exponential time algorithms have an explosive growth rate.

Polynomial vs Exponential-Time Algorithms (cont.)

	n=5	n=10	n=100	n=1000
n	5	10	10^2	10^3
n^2	25	100	10^4	10^6
n^3	125	1000	10^6	10^9
nlogn		10	2×10^{2}	3×10^{3}
2^n	32	1024	1.27×10^{30}	1.07×10^{301}
n!	120	3.6×10^6	9.33×10^{157}	4.02×10^{2567}

Optimization vs Decision Problems

- **Optimization Problem**

  A computational problem in which the object is to find the best of all possible solutions. (i.e. find a solution in the feasible region which has the minimum or maximum value of the objective function.)

- **Decision Problem**

  A problem with a “yes” or “no” answer.

- **Convert Optimization Problems into equivalent Decision Problems**

  What is the optimal value?

  →Is there a feasible solution to the problem with an objective function value equal to or superior to a specified threshold?

Class P

- The class of decision problems for which we can find a solution in *polynomial* time.

  i.e. **P** includes all decision problems for which there is an algorithm that halts with the correct yes/no answer in a number of steps bounded by a polynomial in the problem size n.
- The **Class P** in general is thought of as being composed of relatively “easy” problems for which efficient algorithms exist.

Examples of Class P Problems

- Shortest path
- Minimum spanning tree
- Network flow
- Transportation, assignment and transshipment
- Some single machine scheduling problems

Class NP

- **NP** = Nondeterministic Polynomial
- **NP** is the class of decision problems for which we can *check* solutions in polynomial time.

  i.e. *easy to verify but not necessarily easy to solve*

  Example: easy to verify the correctness of a mathematical proof but difficult to generate a mathematical proof

Class NP (cont.)

- Formally, it is the set of decision problems such that if *x* is a “yes” instance then this could be *verified* in *polynomial* time if a **clue** or **certificate** whose size is polynomial in the size of *x* is appended to the problem input.
- **NP** includes all those decision problems that could be polynomial-time solved if the right (polynomial-length) clue is appended to the problem input.

Extra information so the correctness of an answer to a decision problem can be quickly checked.

Class NP (cont.)

- Given a hypothetical solution to a decision problem, if one can efficiently check that all constraints are met (i.e., feasible) & compute the objective function to compare with the bound, then the problem is in NP.

  Example: composite number problem

Class P vs Class NP

- **Class P** contains all those that have been conquered with well-bounded, constructive algorithms.
- **Class NP** includes the decision problem versions of virtually all the widely studied combinatorial optimization problems.
- **P** is a subset of **NP**.

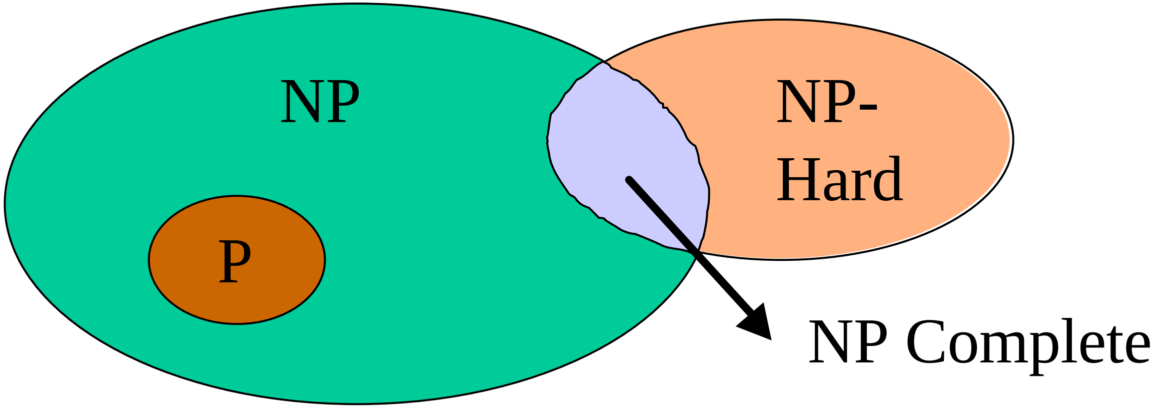
NP Hard vs NP Complete

- When a decision version of a combinatorial optimization problem is proven to belong to the class of **NP-Complete** problems, an optimization version is **NP-Hard.**

-NIST Dictionary of Algorithms & Data Structure

NP Hard vs NP Complete (cont.)

- A problem is said to be ***NP-Hard*** if all members of **NP** *polynomially reduce* to this problem.→ ***NP-Hard*** problems are at least as hard as or harder than any problem in ***NP***.
- A problem is said to be ***NP-Complete*** if (a) it ? **NP**, and (b) it is ***NP-Hard.***→ ***NP-Complete*** problems are the hardest problems in ***NP***.



- **Cook's Theorem**: If there is an efficient (i.e. polynomial) algorithm for some **NP-Complete** problem, then there is a polynomial algorithm existing for all problems in **NP**.→ **P = NP**

- Examples of NP-Hard problems:

  TSP, graph coloring, set covering and partitioning, knapsack, precedence-constrained scheduling, etc.

Reference

- NIST Dictionary of Algorithms & Data Structure

  http://www.nist.gov/dads/
- Comp. Theory FAQ

  http://db.uwaterloo.ca/~alopez-o/comp-faq/faq.html

Polynomial(-time) Reduction

- A transformation of one problem into another which is computable in polynomial time.
- Problem **P** reduces in polynomial-time to another problem **P´**, if and only if,

  - there is an algorithm for problem **P** which uses problem **P´** as a subroutine,

  - each call to the subroutine of problem **P´** counts as a single step,

  - this algorithm for problem **P´** runs in polynomial-time.

Polynomial(-time) Reduction (cont.)

- If problem **P** *polynomially reduces* to problem **P´** and there is a polynomial-time algorithm for problem **P´**, then there is a polynomial-time algorithm for problem **P**.

  →Problem **P´** is at least as hard as problem **P**!

  i.e., If **P´** can be used to solve instances of **P**,

  then **P´** is at least as hard as or harder than **P**.

P		**P´**
easy	←	easy
hard	→	hard