Computational Complexity

[1] *Integer and Combinatorial Optimization*, G.L. Nemhauser& L. A. Wolsey, Wiley, Chapter 15 Computational Complexity.

[2] *Introduction to Algorithms*, T.H. Cormen, C.E. Leiserson, and R.L. Rivest, Mc-Graw Hill, which is good source for learning the analysis of running time of an algorithm, Chapter 36 NP-completeness. It is good source for learning the analysis of running time of an algorithm.

[3] *Combinatorial Optimization*, W.J. Cook, W.H. Cunningham, W. R. Pulleyblank, and A. Schrijver, Wiley, Chapter 9 NP and NP-completeness.

[4] *Scheduling: Theory, Algorithms, and Systems,* Michael L. Pinedo, Springer, Appendix D Complexity Theory.

Size of a problem instance is the amount of information required to represent the instance.

1. Complexity of algorithms

Notation such as big *O* (asymptotically upper bound),Θ ("sandwitch" bounds, asymptotically tight bounds), Ω(asymptotically lower bounds), little *o* (asymptotically non-tight upper bound) [2]

- **Big *O***: Rather than attempting to get a precise expression for the function of running time (worst case), it will suffice to approximate it from above. Function $f(n) = O(g(n))$ as $n \to \infty$ whenever there exists a positive constant c and a positive integer n' such that $|f(n)| \leq c|g(n)|$ for all integers $n \geq n'$. This means that only the asymptotic behavior of the function as *n approaches infinite* (*asymptotically*) is being considered.
- Little *o*: For every positive constant ε there exists a constant n' such that $|f(n)| \leq \varepsilon|g(n)|$ for all integers $n \geq n'$.The big *O* has to be true for *at least one* constant *c*, the little o must hold for *every* positive constant *ε*, however small value (for *ε*).
- Big Theta Θ: $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for some positive c_1 and c_2.
- Big Omega Ω: $f(n) \geq cg(n)$ for some positive *c*.

The algorithm time is evaluated in terms of the size of the input is the number of bits required to write out that input. A 32-bit register (0-1 coding) can store 2^{32} different values.

Polynomial-time algorithms have running time bounds of $O(n^k)$ for small values of *k*, where *n* denotes the number of bits of input given to the algorithm. They have the nice property that their running times do not increase too rapidly as the problem sizes increase. It could be evaluated in the number of arithmetic operations (+, -, multiplications, comparisons) (or called elementary operations) or the number of bit operations (NOT, AND, OR, XOR).

An algorithm runs in **pseudo-polynomial time**, if its running time is a polynomial function of the length of the data encoded in unary (a one-symbol alphabet or numeric value), but is exponential in the length of the input – the number of bits required to represent it.

Example: The knapsack problem can be solved in a pseudo-polynomial time $O(nW)$, where *n* is the number of distinct items and *W* is the weight limitation. Here, *W* is not polynomial in the length of the input though, which is what makes it pseudo-polynomial. Let *s* be the number of bits required to represent *W*. That is, size of input $= \log(W) \to 2^S = W$. Now, running time of knapsack $= O(nW) = O(n\, 2^S)$ which is not polynomial. In many cases pseudo-polynomial time algorithms are perfectly fine because the size of the numbers won't be too large. If we artificially constrain *W* so that *W* is not too large (say, if let *W* be 2), then the runtime is $O(n)$, which actually is polynomial time.

The function *f* is **exponential** if for some constants $c_1, c_2 > 0$ and $d_1, d_2 > 0$ and a positive integer n' we have $c_1 d_1^n \leq f(n) \leq c_2 d_2^n$ for all integers $n \geq n'$.

In fact, some polynomial time algorithms are inefficient and that some algorithms known to be exponential in the worst case are very reliable algorithm for solving practical problems. For example, the simplex algorithm runs in exponential worst-case complexity. Of course, polynomial time algorithms that run in linear time are fast.

2. Complexity of problems

A main theme of computational complexity is the inherent difference between problems known to be in *P* and others for which no polynomial time algorithm is known. It is about the problem classification.

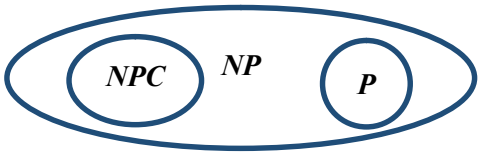
The decision problems, recognition problems, feasibility problems, or yes-no problems are the same: those having a yes/no solution. In the optimization problems, some values must be minimized or maximized. There are standard techniques for transforming optimization problems into decision problems, and vice versa, that do not significantly change the computational difficulty of these problems. For this reason, computability theory and complexity theory have typically focused on decision problems. In order to apply the theory of NP-completeness to optimization problems, we must recast them as feasibility problems. Strictly speaking, the theory of NP-completeness restricts attention to feasibility problems. Typically, an optimization problem can be recast by imposing a bound on the value to be optimized (e.g., k). For example, the feasibility problem form of the knapsack problem is: Can a value of at least V be achieved without exceeding the weight W?

If there exists a polynomial algorithm that solves the feasibility problem, then one can find the maximum value for the optimization problem in polynomial time by applying this algorithm iteratively while increasing the value of k. On the other hand, if an algorithm finds the optimal value of optimization problem in polynomial time, then the feasibility problem can be solved in polynomial time by comparing the value of the solution output by this algorithm with the value of k. That is, if there exists a polynomial time algorithm for the optimization problem, then there exists a polynomial time algorithm for the feasibility problem and vice versa. Thus, both versions of the problem are of similar difficulty.

Let $\boldsymbol{NP}$ be the class of feasibility problems for which any given solution (certificate) can be verified in polynomial time, but there may not be known efficient way to locate a solution (i.e., solve the problem). "NP" stand for "nondeterministic polynomial time". With a certificate (information), the answer "yes" can be checked in polynomial time. Here it is not required that we are able to FIND the certificate in polynomial time. The only requirement is that there exists a certificate with the required properties. Most combinatorial optimization problems are in NP.

Let $\boldsymbol{P}$ be the class of feasibility problems that can be solved in polynomial time.

Let $\boldsymbol{NPC}$ be the class of hardest feasibility problems called NP-complete problems. A problem X in NP is **NP-complete** if every problem in NP can be reduced to X in polynomial time. $NPC \subset NP$; that is, if there exists $X \in NPC \cap P$, then every problem in NP is in P, that is, $P = NP$.



If X_2 is polynomial time solvable, and X_1 is polynomially reducible to X_2, then X_1 is polynomial time solvable. If X_1 is NP-complete and X_1 is polynomially reducible to $X_2 \in NP$, then X_2 is NP-complete. If a problem is easier than a polynomial-solvable problem, it is also polynomial-solvable.

The main consequence of "A reduces to B" is "if we can solve B then we can solve A", in a sense that "A is at most as hard as B is". Polynomial time reduction is a special type of reduction in which the reduction step could be carried out in polynomial time (within some model of computation.) A consequence of the reduction step being carried out polynomial time (i.e., "A reduces to B in polynomial time") is that if we can solve B in polynomial time then one can also solve A in polynomial time. Hence, A is at most as hard as B is when polynomial time computation is concerned. That is, B is at least as hard as A. If we already know that A is at least as hard as any problem in NP, then B is at least as hard as any problem in NP, i.e., it is NP-complete.

We call a problem (a feasibility problem or an optimization problem) **NP-hard** if there is an NP-complete problem that can be polynomially reduced to it. While the feasibility problem is NP-complete, the optimization problem is NP-hard, and its resolution is at least as difficult as the feasibility problem. If problem X_1 is NP-hard, and if X_1 is a special case of X_2, then X_2 is NP-hard. If a problem is harder than an NP-hard problem, it is NP-hard.

Actually, not all problems within the NP-hard class are equally difficult. Some problems are more difficult than others. An NP-hard problem with known pseudo-polynomial time algorithms is called **weakly**

NP-hard or **NP-hard in the ordinary sense** or **simply NP-hard**. An NP-hard problem is called **strongly NP-hard** if it is proven that it cannot be solved by a pseudo-polynomial time algorithm unless $P = NP$. The strong/weak kinds of NP-completeness are defined analogously. TSP is an NP-hard problem.

3. Comparisons among algorithms

[1] Nolinear Programming: Theory and Algorithms, Wiley, M.S. Bazaraa, H.D. Sherali, and C.M. Shetty, Chapter 7.4].

[2] Numerical Optimization, 2nd ed. Jorge Nocedal & Stephen J. Wright, Springer, pp. 8-9.

A solution procedure, or an algorithm, for solving an optimization problem can be viewed as an iterative process that generates a sequence of points according to a prescribed set of instructions, together with a termination. **The algorithms are iterative**. They begin with an initial guess of the variable and generate a sequence of improved estimates until they terminate, hopefully at a solution. The strategy used to move from one solution to the next distinguishes one algorithm from another.

The following goals may conflict.

1. Generality
   Generality of an algorithm refers to the variety of problems (i.e., the number of classical problems) that the algorithm can handle and also to the restrictiveness of the assumptions required by the algorithm.
2. Reliability (Robustness)
   Reliability, or **robustness**, means the ability of the procedure to solve most of the problems in the class for all reasonable values of the starting point for which it is designed with reasonable accuracy. It is preferred that the success of algorithm to deal with a certain size of problems is not sensitive to the starting (feasible) solution as well as the algorithm parameters (such as the step size, the acceleration factor, etc.).
3. Precision (Accuracy)
   An algorithm should be able to identify a solution with precision, without being overly sensitive to errors in the data or the arithmetic rounding errors that occur when the algorithm is implemented on a computer.
4. Preparational and computational effort (Efficiency)
   The preparational effort includes preparing the data (such as the first- or second-order derivatives). The **computational effort** could be assessed by the computer time, the number of iterations, or the number of functional evaluations.
5. Convergence
   Let the sequence $\{x_k\}$ of real numbers converge to $\bar{x}$, and assume that $x_k \neq \bar{x}$ for all k. The **order of convergence** of the sequence is the supremum of the nonnegative numbers $p$satisfying
   $$\overline{\lim_{k\to\infty}} \frac{|x_{k+1}-\bar{x}|}{|x_k-\bar{x}|} = \beta < \infty.$$
   If $p = 1$ and the convergence ratio $\beta \in (0,1)$, the sequence is said to have a **linear convergence rate**. If $p > 1$, or if $p = 1$ and $\beta = 0$, the sequence is said to have **superlinear convergence**. In particular, if $p = 2$ and $\beta < \infty$, the sequence is said to have a **second-order, or quadratic, rate of convergence**. The foregoing rates of convergence are sometimes referred to, respectively, as q-linear, q-superlinear, q-quadratic, and so on. The prefix q stands for the **quotient** taken in the above definition and is used to differ from another weaker type of r-order (root) convergence rate, in which the errors $\|x_k - \bar{x}\|$ are bounded above only by elements of some r-order sequence converging to zero.
6. Storage requirement
   For a polynomial time algorithm, the storage requirement is a polynomial function of the length of the input. However, the converse is false.
7. Theoretical analysis such as algorithm complexity and basic operation count