

Mathematical Programming Glossary Supplement: Introduction to Computational Complexity

Harvey J. Greenberg
hjgreenberg@gmail.com

August 8, 2008
(previous versions: December 17, 2000; May 4, 2001)

This note is a quick introduction to the central concept of a measure of how hard it is to solve some particular optimization problem. A deeper understanding requires texts like the seminal work of Garey and Johnson [2] or the classical book by Wilf [8].

An *algorithm* is a finite sequence of steps defined over a problem domain (input) that terminates after a finite number of steps have been executed with a member of its solution domain (output). The complexity of an algorithm is how many steps it requires to solve a problem as a function of its *size*. (This is *time complexity*, and there is a notion of *space complexity*, which we do not consider here.) If the problem has size n (e.g., n items to be sorted), and an algorithm can solve it in $a + bn$ steps, where a and b are constants, the algorithm has *linear time complexity*, which we denote by $O(n)$. Quadratic complexity is denoted $O(n^2)$, and *polynomial complexity* is denoted $O(n^p)$, where p is a constant.

The “big O” notation is defined as follows. Consider a function that maps non-negative integers to real values: $f : \mathbb{Z}^+ \mapsto \mathbb{R}$. Then, f is $O(K(n))$ if there exists a constant, c , and an integer, N , such that $f(n) \leq cK(n)$ for all $n \geq N$. For example, if an algorithm requires $5n^3 + 2n + 10$ fundamental operations on a problem of size n , its time complexity is $O(n^3)$. The simplex algorithm for linear programming has an exponential worst case time complexity, which we denote by $O(2^n)$, established by Klee and Minty [6] (also see *Glossary* entry).

These are *worst case* values, so the actual time for a particular instance could be less. For example, Gaussian elimination applied to a linear system of equations, $Ax = b$, has $O(n^3)$ worst-case time complexity, but it is much faster if $A = I$ (the identity matrix).

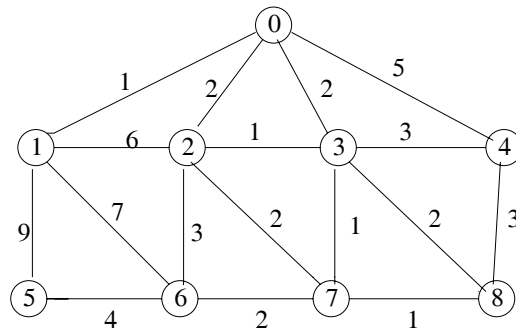
The size of a problem is not always simply the number of variables. In the system of equations, size must also account for the number of equations and the number of digits used to represent the numerical values. We let L be the *length* of the input, which is a scalar that accounts for all parameters pertaining to what we naturally think of as those that determine problem size. (Abstractly, it is the length of an alphabet for the Turing Machine that underlies the algorithm.)

Now that we have the concept of an algorithm's complexity, we shall apply it to define problem classes. How hard is linear programming? Although the simplex method has an exponential worst-case time complexity, some interior methods are polynomial. We thus consider linear programming to be in the class of problems for which a polynomial time algorithm is known to exist, and we denote all such problems as P . Formally, let $\pi(L)$ denote a problem of length L in some problem class \mathcal{C} . Let A denote an algorithm that can solve any problem in \mathcal{C} . Then,

$$P \stackrel{\text{def}}{=} \{ \mathcal{C} : \forall \pi(L) \in \mathcal{C}, \exists A \text{ with complexity } O(L^p) \text{ for some } p \}.$$

Those problem classes that are not in P are either known to have no polynomial time algorithm, or we simply do not know whether such an algorithm exists.

To illustrate, consider how we establish the (worst case) time complexity of Dijkstra's algorithm to find a shortest path in a network. Let $G = [V, E]$ be a graph with vertex set V and edge set E . The following example has $V = \{0, 1, 2, 3, 4, 5, 6, 7, 8\}$ and $E = \{(0, 1), (0, 2), \dots, (7, 8)\}$.



Dijkstra's algorithm finds a shortest path from vertex 0 to each of the other vertices.

```

for  $v \in V$  do
   $L(v) := \infty$ 
end for
 $L(s) := 0; S := \emptyset$ 
while  $T \not\subseteq S$  do
   $u \in \operatorname{argmin}\{L(v) : v \notin S\}$ 
   $S \leftarrow S \cup \{u\}$ 
  for  $v \notin S$  do
    if  $L(u) + w(u, v) < L(v)$  then
       $L(v) := L(u) + w(u, v)$ 
    end if
  end for
end while

```

Let $n = |V|$ (= number of vertices). The initialization takes $O(n)$ time. The first time through the **while** statement, $|S| = n - 1$, so we execute $n - 1$ path length updates. The

second time, we execute $n-2$ updates, and this continues until, as a worst case, we end with 1 update. (We would terminate sooner if we could permanently label the destination node before labelling every other node.) There are thus at most $n-1 + n-2 + n-3 + \dots + 1$ steps executed, which is $O(n^2)$.

Because we found an $O(n^2)$ algorithm that computes the shortest path from one node to a specified destination, we know that the shortest path problem is a member of P .

A particular superset of P is the set of problems for which a trial solution can be verified in polynomial time, and we call this *nondeterministic* polynomial time problems, denoted NP . For example, suppose the problem class is a system of diophantine equations, $Ax = b$ for $x \in \mathbb{Z}^n$. We do not know if there exists a polynomial algorithm to solve this, but for any given solution, x , we can certify that it is a solution in polynomial time (just check that $x \in \mathbb{Z}^n$, then multiply by A to see if $Ax = b$).

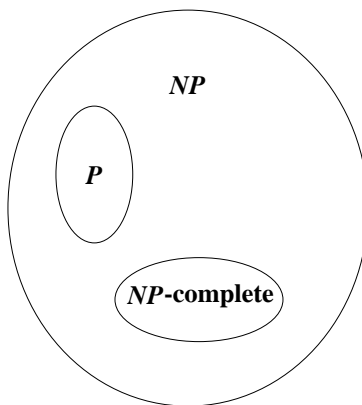
A subset of NP , called *NP-complete*, is defined to be an NP problem for which there is a *polynomial reduction* from any NP problem to it. Formally, let $\pi^1 \propto \pi^2$ denote the property that problem π^1 can be represented as problem π^2 with an algorithmic transformation that has polynomial time complexity. (Note this has nothing to do with whether one problem can be solved in polynomial time.) For example, suppose π^1 is a problem to solve a system of the form $Ax = b$, $x \in \mathbb{Z}^n$, $0 \leq x \leq U$, where $U_j = 2^{d_j} - 1$, $d_j \in \mathbb{Z}^+$ for all j . (In words, each upper bound is one less than some power of 2.) Consider π^2 to be the special case when $U_j = 1$ ($d_j = 1$) for all j . We can transform the more general case of integer solutions to the binary case as follows. Given A, b, U , which define an instance of π^1 , define the corresponding instance of π^2 as follows. Let y denote the binary vector, so π^2 has the form $A'y = b'$ and $y \in \{0, 1\}^n$. Our task is to show we can obtain A' and b' from A, b, U with an algorithm that runs in polynomial time.

Let $x_j = \sum_{k=0}^{d_j} 2^k y_{kj}$. In words, we are using y to represent x with a sequence of bits. Here are some examples, where $U_j = 15$ ($= 2^4 - 1$):

x_j	$y_{\bullet j} = (y_{0j}, y_{1j}, y_{2j}, y_{3j})$	bit sequence				
0	(0, 0, 0, 0)	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	0	0	0
0	0	0	0			
1	(1, 0, 0, 0)	<table border="1"><tr><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	0	0	0	1
0	0	0	1			
4	(0, 0, 1, 0)	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>	0	1	0	0
0	1	0	0			
6	(0, 1, 1, 0)	<table border="1"><tr><td>0</td><td>1</td><td>1</td><td>0</td></tr></table>	0	1	1	0
0	1	1	0			
13	(0, 1, 1, 1)	<table border="1"><tr><td>0</td><td>1</td><td>1</td><td>1</td></tr></table>	0	1	1	1
0	1	1	1			
15	(1, 1, 1, 1)	<table border="1"><tr><td>1</td><td>1</td><td>1</td><td>1</td></tr></table>	1	1	1	1
1	1	1	1			

Thus, $Ax = b \Leftrightarrow \sum_k A_{\bullet,j} 2^k y_{kj} = b$. The transformation is to define the system $A'y = b'$ with $b' = b$ and $A' = [A_{\bullet,j} 2^k]$. This takes polynomial time since the number of columns in A' is a polynomial function of the number of columns of A , so the general bounded system of diophantine equations is \propto the binary system of diophantine equations.

Here is a picture of the sets of problem classes of interest to us.



The picture shows P disjoint from NP -complete, but this is not known. A famous open problem is whether $P = NP$. No one knows, but many suspect that $P \neq NP$.

The concept of reducing one problem class to another is fundamental to the theory of NP -completeness. One property is that the problems in NP -complete are either all polynomial or none are. To see this, consider an algorithm A with complexity $O(L^p)$ that solves a member, $\pi(L)$. Let $\pi'(L')$ be a member of some other class in NP -complete. Apply the polynomial algorithm to transform, $\pi' \propto \pi$, then apply A and obtain the solution to π , say x . Transform x to a solution of π' and ask for certification, which can be done in polynomial time. The composite algorithm has polynomial complexity because each of its parts does.

Finally, an NP -hard problem includes optimization problems that can be solved with a polynomial number of NP -complete problem solutions. For example, the travelling salesman problem (TSP) is NP -hard because it can be solved by getting a polynomial number of solutions to the question, *Does there exist a tour whose value does not exceed c ?*, where c is a constant. (Obtain initial bounds on the TSP, then apply bisection.)

Although complexity theory has roots in computability, notably by Rabin in 1959, this all got started in the early 1960s (see Hartmanis and Stearns^[3]) A landmark followed in 1971 by the work of a logician, Stephen Cook, who considered the *Satisfiability Problem* (SAT): *Given a set of boolean variables and a collection of clauses, is there an assignment of values to the variables such that all clauses are true?*

Here are short definitions of the basic problem sets:

Problem set	What it contains
P	For each problem class, there is a known algorithm to solve it in polynomial time.
NP	For each problem class, there is not a known algorithm to solve in polynomial time, but there is a polynomial algorithm to certify a candidate solution.
NP -complete	NP problems $\{\pi\}$, such that $\pi' \propto \pi$ for all $\pi' \in NP$. Key property: if any π is in P , they are all in P .

Cook's Theorem^[1]: SAT is NP-complete.

The next milestone was by Richard Karp^[5], who added many optimization problems to the NP-complete set. Since then, the list of NP-complete problems has become enormous, and the central concept of complexity has been extended to consider parallel computation (P -completeness) and average case complexity.

The structure of a proof that a problem class, $\mathcal{C} = \{\pi\}$, is NP-complete is composed of two parts:

1. Show $\pi \in NP$.
2. Show $\pi' \propto \pi$ for some $\pi' \in NP$ -complete.

Transitivity gives the desired result: for any $\pi'' \in NP$, we have $\pi'' \propto \pi'$, so $\pi' \propto \pi \Rightarrow \pi'' \propto \pi$.

Here is an example of a proof. Let \mathcal{C} be the system of binary equations and inequalities: $Ax = 1$, $Bx \geq 1$ $x \in \{0, 1\}^n$, where the data are binary — that is, $A_{ij}, B_{kj} \in \{0, 1\}$ for all i, k, j (and all right-hand sides are 1). This is in NP since any solution, x , can be certified with a polynomial algorithm that first checks if $x \in \{0, 1\}^n$, then multiplies x by A and B and tests if $Ax = 1$ and $Bx \geq 1$. We shall show that SAT $\propto \pi$.

Suppose we are given propositions P_1, \dots, P_n and clauses C_1, \dots, C_m . Define $2n$ variables, $x_{-1}, \dots, x_{-n}, x_1, \dots, x_n$, where $x_{-j} + x_j = 1$ for all j . Let $A = [I \ I]$, so the system of equations, $Ax = 1$, corresponds to defining the logical complements. (Think of $x_j = 1 \Leftrightarrow P_j = \text{TRUE}$, and $x_{-j} = 1 \Leftrightarrow \neg P_j = \text{TRUE}$.) Now define

$$B_{ij} = \begin{cases} 1 & \text{if } P_j \in C_i; \\ 0 & \text{otherwise.} \end{cases} \quad B_{i,-j} = \begin{cases} 1 & \text{if } \neg P_j \in C_i; \\ 0 & \text{otherwise.} \end{cases}$$

Any solution to this system corresponds to the SAT solution by letting $P_j = \text{TRUE}$ iff $x_j = 1$.

Example: Given SAT: $(P_1 \vee P_3) \wedge (P_2 \vee \neg P_3) \wedge (\neg P_1 \vee \neg P_2 \vee P_3)$, the transformation creates

the following system of equations and inequalities:

$$\begin{array}{rcccccc}
 x_{-1} & & & + & x_1 & & = & 1 \\
 & x_{-2} & & & & + & x_2 & = & 1 \\
 & & x_{-3} & & & & + & x_3 & = & 1 \\
 & & & & x_1 & & + & x_3 & \geq & 1 \\
 & & & & x_{-3} & & + & x_2 & \geq & 1 \\
 x_{-1} & + & x_{-2} & + & & & + & x_3 & \geq & 1
 \end{array}$$

The first three equations define the logical complementary relation, $x_{-j} = 1 - x_j$ for $j=1,2,3$. The inequalities ensure each clause has at least one literal that is true. In this example, a binary solution to the above system is $x = (0, 0, 0, 1, 1, 1)$; the corresponding truth assignment for which the SAT is true is $P_j = \text{TRUE}$ for all $j = 1, 2, 3$.

The complete theory of computational complexity includes other problem classes, but the fundamental property of polynomiality remains a key concept, notably the reducibility from one problem class to another. Critics of complexity theory argue that it is not practical to consider only the worst-case complexity, and the performance of the simplex method in practice appears to be a fairly low order polynomial complexity, once pathologies like the Klee-Minty polytope are removed. Advocates of complexity theory argue that it does provide a practical guide to algorithm design, and this is especially true of *approximation algorithms*. These comprise an approach to “solving” hard problems, where we give up accuracy in order to have tractability. To be an approximation algorithm, it must have polynomial complexity, and it must give a guaranteed bound on the quality of a solution.

For more tutorial-quality introductions see Tovey^[7] and David Johnson’s columns^[4].

References

- [1] S. A. Cook, The Complexity of Theorem-Proving Procedures, *Proceedings of the 3rd Annual IBM Symposium on Theory of Computing Machinery*, ACM, New York, NY, 1971, 151–158.
- [2] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W.H. Freeman and Co., New York, NY, 1979.
- [3] J. Hartmanis and R. E. Stearns, On the Computational Complexity of Algorithms, *Transactions of the American Mathematical Society* 117 (1965), 285–306.
- [4] D. S. Johnson, NP-Completeness Columns, <http://www.research.att.com/~dsj/columns/>
- [5] R.M. Karp, Reducibility among Combinatorial Problems, in *Complexity of Computer Computations*, K.E. Miller and J.W. Thatcher (editors), Plenum Press, New York, NY, 1972, 85–103.
- [6] V. Klee and G. J. Minty. How Good is the Simplex Algorithm?, in *Inequalities*, III, O. Shisha (editor), Academic Press, New York, NY, 1972, 159–175.
- [7] C. Tovey, Tutorial on Computational Complexity, *Interfaces* 32:3 (2002), 30–61.
- [8] H. S. Wilf, *Algorithms and Complexity*, Internet Edition, <http://www.cis.upenn.edu/~wilf/AlgComp.html>, 1994.