A Brief Introduction to Algorithms

Paul Cull

Department of Computer Science
Oregon State University

May 13, 2011

Contents

Chapter 1

HARD PROBLEMS

1.1 Classification of Algorithms and Problems

We have encountered various algorithms, particularly of the divide-and-conquer type, which have running times like $\Theta(n)$, $\Theta(nlogn)$, and $\Theta(n^2)$, where n is some measure of the size of the input. We have also encountered algorithms, particularly of the exhaustive search type, which have running times like $\Theta(2^n)$ and $\Theta(n!)$. For algorithms of the first type, doubling the size of the input increases the running time by a constant factor, while for algorithms of the second type, doubling n increases the running time by a factor proportional to the running time. If we double the speed of the computer we are using, then the largest input size which our computer can solve in a given time will increase by a constant factor if we have an algorithm of the first type; for an algorithm of the second type, the input size our computer can solve will only increase by an additive constant, at best. These considerations led Edmonds to propose that algorithms of the first type are computationally reasonable, while algorithms of the second type are computationally unreasonable. More specifically, he suggested the definitions:

- *reasonable algorithm:* an algorithm whose running time is bounded by a polynomial in the size of the input.
- *unreasonable algorithm:* an algorithm whose running time cannot be bounded by any polynomial in the size of the input.

This suggests that we should try to replace unreasonable algorithms by reasonable algorithms. Unfortunately, this goal is not always attainable. For the Towers of Hanoi problem, any algorithm must have running time at least $\Theta(2^n)$. Since some problems do not have reasonable algorithms, we should classify problems as well as algorithms. Corresponding to Edmonds definition for algorithms, Cook and Karp suggested the following definition for problems:

- *easy problem:* a problem which has a polynomial time bounded algorithm.

- *hard problem:* a problem for which there is no polynomial time bounded algorithm.

An easy problem may have unreasonable algorithms. For example, we have seen an $\Theta(n!)$ exhaustive search algorithm for sorting, but sorting is an easy problem because we also have an $\Theta(n \log n)$ sorting algorithm. A hard problem, on the other hand, can never have a reasonable algorithm.

Some problems like Towers of Hanoi are hard for the trivial reason that their output is too large. To avoid such output-bound problems, Cook suggested considering only *yes/no* problems; that is, problems whose output is limited to be either *yes* or *no*. One such yes/no problem is the variant of the Towers of Hanoi problem in which the input is a configuration and the question is: is this configuration used in moving the disks from tower A to tower C? Furthermore, this variant is an easy problem. Usual easy problems can be transformed into easy yes/no problems by giving as the instance of the yes/no problem both the input and the output of the usual problem and asking if the output is correct. For example, sorting can be converted into a yes/no problem when we give both the input and the output and ask if the output is the input in sorted order. As another example, matrix multiplication can be converted into a yes/no problem in which we give three matrices A, B, and C and ask if $C = AB$. There are also other ways to transform usual problems into yes/no problems. Examples are the above variant of the Towers of Hanoi, and the variant of sorting in which we ask if the input is in sorted order.

To avoid problems which are hard only because of the length of their output, Cook defined:

- P = the class of yes/no problems which have polynomial time algorithms.

(Some authors call this class $PTIME$.) For many problems, it is easy to show that they are in P. One gives an algorithm for the problem and demonstrates a polynomial upper bound on its running time. It may be quite difficult to show that a problem is not in P, but it is clear that there are problems which are not in P. For example, the halting problem, which asks if an algorithm ever terminates when given a particular input, is not in P because the halting problem has no algorithm, and therefore certainly no polynomial time algorithm.

Classification of problems would not be very useful if we only could say that some problems have polynomial time algorithms and some problems have no algorithms. We would like a finer classification, particularly one that helps classify problems which arise in practice. In the next section we introduce some machinery which is needed for a finer classification.

1.2 Nondeterministic Algorithms

Up to this point we have discussed only deterministic algorithms. In a deterministic algorithm there are no choices; the result of an instruction determines which instruction will be executed next. In a nondeterministic algorithm choices

are allowed; any one of a set of instructions may be executed next. Nondeterministic algorithms can be viewed as "magic" : if there is a correct choice, the magic forces the nondeterministic algorithm to make this correct choice. A less magic view is that if there are several possibilities the nondeterministic algorithm does all of them in parallel. Since the number of possibilities multiply at each choice-point, there may be arbitrarily many possibilities being executed at once. Therefore, a nondeterministic algorithm gives us unbounded parallelism.

This view of nondeterminism as unbounded parallelism makes clear that nondeterminism does not take us out of the realm of things which can be computed deterministically, because we could build a deterministic algorithm which simulates the nondeterministic algorithm by keeping track of all the possibilities. However, a nondeterministic algorithm may be faster than any deterministic algorithm for the same problem. We define the time taken by a nondeterministic algorithm as the fewest instructions the nondeterministic algorithm needs to execute to reach an answer. (This definition is not precise since what an instruction means is undefined. This could be made precise by choosing a model of computation like the Turing machine in which an instruction has a well-defined meaning, but this imprecise definition should suffice for our purposes.) With this definition of time, a nondeterministic algorithm could sort in $\Theta(n)$ time because it always makes the right choice, whereas a deterministic algorithm would take at least $\Theta(n \log n)$ time in some cases. In some sense we are comparing the best case of the nondeterministic algorithm with the worst case of the deterministic algorithm, so it is not surprising that the nondeterministic algorithm is faster.

For yes/no problems we give nondeterministic algorithms even more of an edge. We divide the inputs into yes-instances and no-instances. The yes-instances eventually lead to yes answers. The no-instances always lead to no answers. We assume that our nondeterministic algorithms cannot lead to yes as a result of some choices, and to a no as a result of some other choices. For a yes-instance, the running time of a nondeterministic algorithm is the fewest instructions the algorithm needs to execute to reach a yes answer. The running time of the nondeterministic algorithm is the maximum over all yes-instances of the running time of the algorithm for the yes-instances. We ignore what the nondeterministic algorithm does in no-instances.

As an example of nondeterministic time, consider the yes/no problem: given a set of n numbers each containing $\log n$ bits, are there two identical numbers in the set? In a yes-instance, a nondeterministic algorithm could guess which two numbers were identical and then check the bits of the two numbers, so the running time for this nondeterministic algorithm is $\Theta(\log n)$. On the other hand, even in a yes-instance, a deterministic algorithm would have to look at almost all the bits in worst case, so the running time for any deterministic algorithm is at least $\Theta(n \log n)$.

The definition of nondeterministic time may seem strange, but it does measure an interesting quantity. If we consider a yes/no problem, the yes-instances are all those objects which have a particular property. The nondeterministic time is the length of a proof that an object has a certain property. We ignore

no-instances because we are not interested in the lengths of proofs that an object does not have the property.

1.3 NP and Reducibility

Now that we have a definition of the time used by a nondeterministic algorithm, we can, in analogy with the class P, define

- NP = the class of yes/no problems which have polynomial time nondeterministic algorithms.

It is immediate from this definition that

$$P \subseteq NP$$

because every problem in P has a deterministic polynomial time algorithm and we can consider a deterministic algorithm as a nondeterministic algorithm which has exactly one choice at each step. It is not clear, however, whether P is properly contained in NP or whether P is equal to NP.

Are there problems in NP which may not be in P? Consider the yes/no version of satisfiability: given a Boolean expression, is there an assignment of true and false to the variables which makes the expression true? This problem is in NP because if there is a satisfying assignment we could guess the assignment, and in time proportional to the length of the expression we could evaluate the expression and show that the expression is true. We discussed exhaustive algorithms for satisfiability because these seem to be the fastest deterministic algorithms for satisfiability. No polynomial time deterministic algorithm for satisfiability is known. Similarly, the problem: given a graph does it contain a *Hamiltonian path*?, seems to be in NP but not in P. Hamiltonian path is in NP because we can guess the Hamiltonian path and quickly (i.e., in polynomial time) check to see if the guessed path really is a Hamiltonian path. Hamiltonian path does not seem to be in P, because exhaustive search algorithms seem to be the fastest deterministic algorithms for this problem and these search algorithms have worst case running times which are at least $\Theta(2^n)$, and hence their running times cannot be bounded by any polynomial.

Another problem which is in NP but may not be in P is *composite* number: given a positive integer n, is n the product of two positive integers which are both greater than 1? Clearly this is in NP because we could guess the two factors, multiply them, and show that their product is n. Why isn't this problem clearly in P? Everyone knows the algorithm which has running time at most $\Theta(n^2)$ and either finds the factors or reports that there are no factors. This well-known algorithm simply tries to divide n by 2 and by each odd number from 3 to $n-1$. While this algorithm is correct, its running time is not bounded by a polynomial in the size of the input. Since n can be represented in binary, or in some other base, the size of the input is only $\log n$ bits, and n cannot be bounded by any polynomial in $\log n$. This example points out that we have been

too loose about the meaning of size of input. The official definition says that the size of the input is the number of bits used to represent the input. According to the official definition, the size of the input for this problem is $\log n$ if n is represented in binary. But if n were represented in unary then the size of the input would be n. So the representation of the input can affect the classification of the problem.

There are a great variety of problems which are known to be in NP, but are not known to be in P. Some of these problems may not seem to be in NP because they are optimization problems rather than yes/no problems. An example of this kind of optimization problem is the *traveling salesman problem*: given a set of cities and distances between them, what is the length of the shortest circuit which visits each city exactly once and returns to the starting city? This optimization problem can be changed into a yes/no problem by giving an integer B as part of the input. The yes/no question becomes: is there a circuit which visits each city exactly once and returns to the starting city and has length at most B? At first glance the optimization problem seems harder than the yes/no problem, because we can solve the yes/no problem by solving the optimization problem, and solving the yes/no problem does not give a solution to the optimization problem. But we can use the yes/no problem to solve the optimization problem.

- Set B to n times the largest distance; then the answer to the yes/no problem is yes.
- Set $STEP$ to $B/2$.
- Now set B to $B - STEP$, and $STEP$ to $STEP/2$.
- Now do the yes/no problem with this new B and this new $STEP$.
- If the answer is yes, set B to $B - STEP$ and $STEP$ to $STEP/2$.
- If the answer is no, set B to $B + STEP$ and $STEP$ to $STEP/2$.
- Continue this process until $STEP = 0$.

The last value of B will be the solution to the optimization problem. How long will this take? Since $STEP$ is halved at each call to the yes/no procedure, the number of calls will be the log of the initial value of $STEP$. But $STEP$ is initially n times the largest distance, so the number of calls is $\log n + \log$(largest distance) which is less than the size of the input. Thus the optimization problem can be solved by solving the yes/no problem a number of times which is less than the length of the input.

This example suggests the idea that two problems are equally hard if both of them can be solved in polynomial time if either one of them can be solved in polynomial time. This equivalence relation on problems also suggests a partial ordering of problems. A problem A is no harder than problem B if a polynomial time deterministic algorithm for B can be used to construct a polynomial time deterministic algorithm for A. We symbolize this relation by $A \leq B$. If A is the

yes/no traveling salesman problem, and B is the traveling salesman optimization problem, then we have both $A \leq B$ and $B \leq A$. If A and B are any two problems in P then we have both $A \leq B$ and $B \leq A$, because we could take the polynomial time deterministic algorithm for one problem and make it a subroutine of the polynomial deterministic algorithm for the other problem and never call the subroutine. While in these examples, the relation $\leq$ works both ways, there are cases in which $\leq$ only works one way. For example, let A be any problem in P. and let $HALT$ be the halting problem; then $A \leq HALT$, but $HALT \not\leq A$, because we don't need an algorithm for $HALT$ to construct a polynomial time algorithm for A, and the polynomial time algorithm for A cannot help in constructing any algorithm for $HALT$, let alone a polynomial time algorithm for $HALT$.

This relation $A \leq B$ which we are calling *A is no harder than B*, is usually called polynomial time reducibility, and is read A is polynomial time reducible to B. There are many other definitions of reducibility in the literature. We refer the interested reader to Garey and Johnson[1979] and Hartley Rogers[1967].

The notion of a partial ordering on problems should aid us in our task of classifying problems. In particular, it may aid us in saying that two problems in NP are equally hard. Further, it suggests the question: is there a hardest problem in NP? We consider this question in the next section.

1.4 NP-Complete Problems

In this section, we will consider the relation $\leq$ defined in the last section, and answer the question: is there a hardest problem in NP? Since we have a partial order $\leq$, we might think of two very standard instances of partial orders: the partial (and total) ordering of the integers, which has no maximal element; and the partial ordering of subsets, which has a maximal element. From these examples, we see that our question cannot be answered on the basis that we have a partial order. If we consider $\leq$ applied to problems, is there a hardest problem? The answer is no, because the Cantor diagonal proof always allows us to create harder problems. On the other hand, one may recall that the halting problem is the hardest recursively enumerable (RE) problem. So on the analogy with RE, there may be a hardest problem in NP. Cook [1971] proved that there is a hardest problem in NP. A problem which is the hardest problem in NP is called an NP-complete problem. Cook proved the more specific result:

Theorem 1 (Cook's Theorem). *Satisfiability is NP-complete.*

The proof of this theorem requires an exact definition of nondeterministic algorithm, and since we have avoided exact definitions, we will only be able to give a sketch of the proof. We refer the interested reader to Garey and Johnson[1979] for the details. The basic idea of the proof is to take any instance of a problem in NP which consists of a nondeterministic algorithm, a polynomial that gives the bound on the nondeterministic running time, and an input for the algorithm, and to show how to construct a Boolean expression which is

satisfiable iff the nondeterministic algorithm reaches a *yes* answer within the number of steps specified by the polynomial applied to the size of the input. The construction proceeds by creating clauses which can be interpreted to mean that at step 0 the algorithm is in its proper initial state. Then for each step, a set of clauses are constructed which can be interpreted to mean that the state of the algorithm and the contents of the memory are well-defined at this step. Further, for each step a set of clauses are constructed which can be interpreted to mean that the state of the algorithm and the contents of memory at this step follow from the state and contents at the previous step by an allowed instruction of the algorithm. Finally, some clauses are constructed which can be interpreted to mean that the algorithm has reached a *yes* answer. The polynomial time bound is used to show that the length of this Boolean expression is bounded by a polynomial in the length of the input.

An interesting consequence of the proof is that satisfiability of Boolean expressions in clause form is NP-complete. This result can be refined to show that satisfiability in clause form with exactly 3 literals per each clause is also NP-complete.

After Cook's result, Karp[1972] quickly showed that a few dozen other standard problems are NP-complete. Garey and Johnson's book contains several hundred NP-complete problems. Johnson also writes a column for the Journal of Algorithms which contains even more information on NP-complete problems.

Why is this business of NP-complete problems so interesting? The NP-complete problems are the hardest problems in NP in the sense that for any problem A in NP, $A \leq NP-complete$. So if there were a polynomial time deterministic algorithm for any NP-complete problem, there would be a polynomial time deterministic problem for any problem in NP; that is, P and NP would be the same class. Conversely, if $P \neq NP$, then there is no point to looking for a polynomial time deterministic algorithm for an NP-complete problem. Simply knowing that a problem is in NP without knowing that it is NP-complete leaves open the question of whether or not the problem has a polynomial time bounded algorithm even on the supposition that $P \neq NP$. The fact that a number of NP-complete problems have been well known problems for several hundred years and no one has managed to find a reasonable algorithm for any one of them suggests to most people that $P \neq NP$ and that the NP-complete problems really are hard.

One of the virtues of Cook's theorem is that *to show that an NP problem A is NP-complete you only have to show that $satisfiability \leq A$.*

As a catalog of NP-complete problems is built, the task of showing that an NP problem A is NP-complete gets easier because you only have to pick some NP-complete problem B and show that $B \leq A$.

We have already mentioned the traveling salesman problem (TSP). This problem is NP-complete. Let us show that if Hamiltonian circuit is NP-complete then TSP is NP-complete. The Hamiltonian circuit problem (HC) is: *given a graph, is there a circuit which contains each vertex exactly once and uses only edges in the graph?* Given an instance of HC, we create an instance of TSP by letting each vertex from HC become a city for TSP, and defining the

distance between cities by $d(i,j) = 1$ if there is an edge in HC between vertices i and j, and $d(i,j) = 2$ if there is no such edge. Now if there were n vertices in HC, we use $B = n$ as our bound for TSP. If the answer to TSP is *yes*, then there is a circuit of length n, but this means that the circuit can only contain edges of distance 1 and hence this circuit is also a Hamiltonian circuit of the original graph. Conversely, if there is a Hamiltonian circuit, then there is a TSP circuit of length n. To complete our proof. We must make sure that this transformation from HC to TSP can be accomplished in polynomial time in the size of the instance of HC. Since HC has n vertices and TSP can be specified by giving the $n(n-1)/2$ distances between the n cities, we only have to check for each of the $n(n-1)/2$ distances whether or not it corresponds to an edge in the original graph. Even with a very simple algorithm this can be done in at worst $\Theta(n^4)$, which is bounded by a polynomial in the size of the HC instance.

This is a very simple example of proving that one problem in NP is NP-complete by reducing a known NP-complete problem to the problem. We refer the reader to Garey and Johnson[1979] for more complicated examples.

1.5 Dealing with Hard Problems

In the theoretical world there are hard problems. Some of these hard problems are NP-complete problems and there are other problems which are harder than NP-complete problems. How can these hard problems be handled in the real world?

The simplest way to handle hard problems is to ignore them. Many practical programmers do not know what hard problems are. Their programming involves tasks like billing and payroll which are theoretically trivial, but practically quite important. Ignoring hard problems may be a reasonable strategy for these programmers.

Another way to handle hard problems is to avoid them. To avoid hard problems, you have to know what they are. One of the major virtues of lists of NP-complete problems is that they help the programmer to identify hard problems and to point out that no reasonable algorithms for these problems are known. It is often unfortunately the case that a programmer is approached with a request for a program and the requester has tried to remove all the specific information about the problem and generalize the problem as much as possible. Over-generalization can make a problem very hard. If the programmer can get the specific information, she may be able to design a reasonable algorithm for the real problem and avoid the hard generalization.

Sometimes real problems are really hard but not too big. For example, many real scheduling problems turn out to be traveling salesman problems with 30 to 50 cities. For these situations an exhaustive algorithm may still solve the problem in reasonable time. The programmer should still try to tune the algorithm to take advantage of any special structure in the problem, and to take advantage of the instructions of the actual computer which will be used. Exhaustive search is a way to handle some NP-complete problems when the

size of the input is not too large.

Heuristics are another way to deal with hard problems. We have already mentioned heuristics in discussing backtrack algorithms. A heuristic is a method to solve a problem which doesn't always work. To be useful a heuristic should work quickly when it does work. The use of heuristics is based on the not unreasonable belief that the real world is usually not as complicated as the worst case in the theoretical world. This belief is supported by the observation that creatures which seem to have less computing power than computers can make a reasonable living in the real world. Artificial intelligence has been using heuristics for years to solve problems like satisfiability. These heuristics seem to be very effective on the instances of satisfiability which arise in artificial intelligence contexts. Heuristics are also widely used in the design of operating systems. Occasional failures in these heuristics lead to software crashes. Since we usually see only a couple of such crashes per year these heuristics seem to be very effective.

Sometimes the behavior of heuristics can be quantified so that we can talk about the probability of the heuristic being correct. For example, a heuristic to find the largest element in an n element array is to find the largest element among the first $n-1$ elements. If the elements of the array are in random order, then this heuristic fails with probability $1/n$, and as $n \to \infty$ the probability that this heuristic gives the correct answer goes to 1. Such probabilistic algorithms are now being used for a wide variety of hard problems. For example, large primes are needed for cryptographic purposes. While it seems to be hard to discover large primes, there are tests which are used so that if a number passes all the tests, then the number is probably a prime.

Many hard problems can be stated as optimization problems: find the smallest or largest something which has a particular property. While actually finding the optimum may be difficult, it may be much easier to find something which is close to optimum. For example, in designing a computer circuit one would like the circuit with the fewest gates which carries out a particular computation. This optimization problem is hard. But from a practical point of view, no great disaster would occur if you designed a circuit with 10% more gates than the optimum circuit. For various hard problems, approximation algorithms have been produced which produce answers close to the optimum answer. We will consider an example of approximation in the next section.

1.6 Approximation Algorithms

Let us consider the traveling salesman optimization problem: *given a set of cities and distances between them, find the shortest circuit which contains each city exactly once.* This problem arises in many real scheduling situations. We will try to approximate the shortest circuit.

To make an approximation possible, we will assume that the distances behave like real distances, that is, the distances obey the triangle inequality $d(i,j) \leq d(i,k) + d(k,j)$, so that the distance from city i to city j is no longer than the

distance from city i to city k plus the distance from city k to city j.

A simpler task than finding the minimum circuit is finding the minimum spanning tree. *The minimum spanning tree is a set of links which connects all the cities and has the smallest sum of distances.* In the minimum spanning tree, the cities are not all directly connected; several links may have to be traversed to get from city i to city j. There is a reasonable algorithm for the minimum spanning tree because the shortest link is always in this tree. So one can proceed to find this tree by putting in the shortest link and continuing to add the shortest link which does not complete a cycle.

A circuit can be constructed from the minimum spanning tree by starting at some city and traversing the links of the tree to visit every other city and returning to the starting city. This circuit is twice as long as the sum of the distances of the links in the minimum spanning tree. But this circuit may visit some cities more than once. To "clean up" this circuit, we use this circuit while no city is repeated and, if city j is the first repeated city and if city i is the city before city j and if city k is the next city after city j which has not yet been visited, we connect city i to city k. We continue to use this procedure to produce a circuit in which each city is visited exactly once. From the triangle inequality we have that the length of this new circuit is at most as long as the circuit with cities repeated, and hence that this new circuit is no longer than twice the length of the minimum spanning tree.

The optimum circuit must be at least as long as the minimum spanning tree because the optimum circuit connects every city. Thus we have

$$OPT \leq ALG \leq 2OPT$$

where OPT is the length of the optimum circuit and ALG is the length of the circuit produced by the approximation algorithm.

It would be pleasant if all hard optimization problems had approximation algorithms. Unfortunately this is not the case. We really needed the triangle inequality to produce an approximation for the traveling salesman problem. Consider an instance of Hamiltonian circuit with n vertices. We can convert this to an instance of traveling salesman without triangle inequality by assigning distance 1 to all the edges which are in the original graph, and assigning distance $n + 2$ to all the edges which were not in the original graph. Now if there were a Hamiltonian circuit in the original graph, then there would be a traveling salesman circuit of length n. If we could approximate this traveling salesman problem within a factor of 2, the traveling salesman circuit would have length at most $2n$ exactly when the original graph had a Hamiltonian circuit, because if the traveling salesman circuit used even one of the edges not in the original graph, it would have length at least $2n + 1$. So *approximating the traveling salesman problem without triangle inequality is as hard as Hamiltonian circuit.*

This example can be generalized to show that no approximation within a factor of $f(n)$ is possible by assigning each edge not in the graph a distance greater than $n\,f(n) + 1$. If the original graph has a Hamiltonian circuit then the approximating algorithm must report a value of at most $n\,f(n)$. On the other

hand, if there was no Hamiltonian circuit then the approximating algorithm must report a value of at least $n(f(n)+1)$. So, by simply looking at the reported approximating value, one can determine whether or not the original graph has a Hamiltonian circuit.

To make sure the above transformation can be carried out in polynomial time in the size of the instance of Hamiltonian circuit, $f(n)$ must be bounded by $2^{p(n)}$ where $p(n)$ is a polynomial. Thus no reasonable approximation to the traveling salesman problem is possible unless the Hamiltonian circuit problem can be solved quickly; that is, unless $P = NP$.

1.7 The World of NP

The fact that various NP-complete problems have resisted attempts to find reasonable algorithms for them suggests to many people that $P \neq NP$. Even if we accept this belief, there are still other open questions about classes of problems associated with NP. The class NP is defined in terms of the *yes*-instances of its problems. A class could also be defined in terms of *no*-instances. In correspondence with NP, we define the class *coNP* as problems whose complements are in NP; that is, for each problem in *coNP* there is a nondeterministic algorithm which has polynomial bounded running time for the *no* instances of the problem.

The notion of $\leq$ we have used before will lump NP and *coNP* together. To tell them apart we will need a finer notion called *polynomial time many-one reducibility*, and as is traditional in computer science, we will "overload" the symbol $\leq$ to stand for this new partial ordering. This new ordering will be defined on sets, but we will also use the ordering for problems where we ambiguously use the same name for the problem and for the set of *yes*-instances of the problem. For example, we will use SAT to mean the satisfiability problem, and we will also use SAT to mean the set of Boolean formulas which can be satisfied.

With these preliminaries, we define:

$$A \leq B$$

iff there exists a polynomial time computable function $f(x)$*, so that for every input instance* x*,*

$$x \in A \qquad \text{iff} \qquad f(x) \in B.$$

With this ordering it is still true that all (nontrivial) sets in P are mutually equivalent, i.e.

$$\forall A, B \in P \qquad A \leq B \qquad \text{and} \qquad B \leq A.$$

On the other hand, sets in NP and *coNP* do not have to be related. For example, let SAT mean all the Boolean expressions that are *true* for *at least* one setting of the variables, and let *coSAT* mean all the Boolean expressions that are *false* for *all* settings of the variables. We have already seen that $SAT \in NP$.

Now, notice that $coSAT \in coNP$ because if the answer to the $coSAT$ question is *no*, then there is a setting which makes the expression true (that is, not false) and a nondeterministic algorithm could quickly guess the setting and verify that the setting makes the expression true. While SAT and $coSAT$ seem to be very closely related, this notion of polynomial time many-one reducibility ($\leq$) does *not* seem to see them as related. Notice that for an expression E, we have

$$E \in coSAT \qquad \text{iff} \qquad E \notin SAT.$$

but our notion of $\leq$ does *not* allow $\not\leq$, it requires that

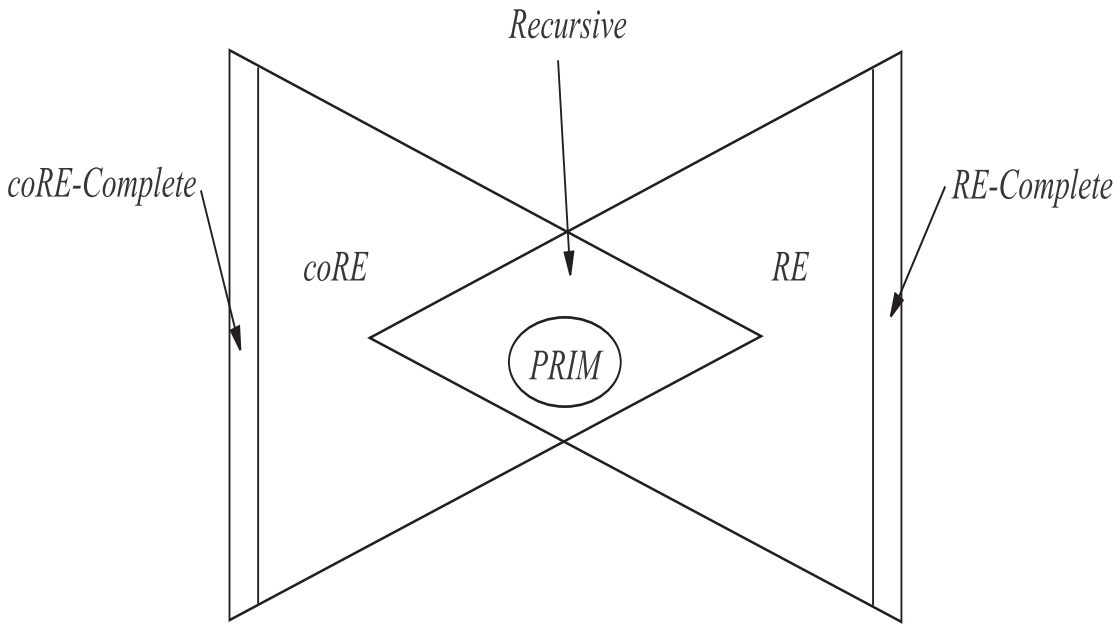
$$E \in coSAT \qquad \text{iff} \qquad f(E) \in SAT.$$

While it is possible to construct this $f(E)$, the only ways I know would give, at least for some E's, an $f(E)$ whose length is exponentially larger than the length of E. So, we expect that this notion of $\leq$ will be able to distinguish NP from $coNP$.

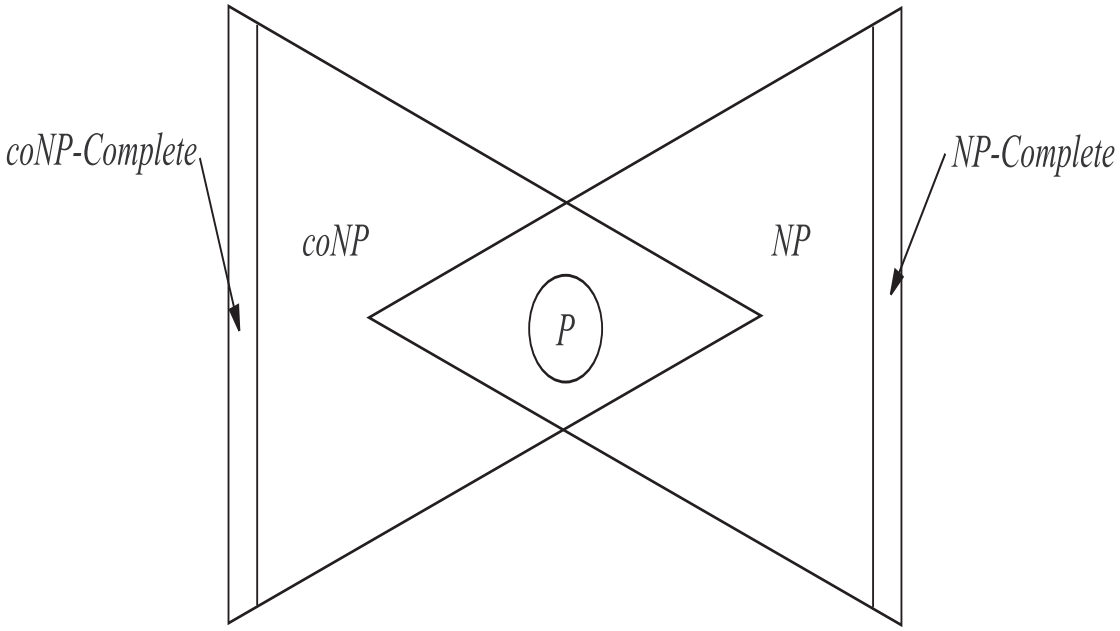
The above definition of $coNP$ suggests the questions:

- Does $NP = coNP$?
- Does $P = NP \cap coNP$?

Unfortunately, these questions are unsolved. While the above questions are unsolved, many people believe that the answer to each of these questions is no. The basis for this belief is the analogy between NP and RE. For RE the following diagram can be shown to be valid:



If the analogy between NP and RE is valid, the world of NP should look like:



A minor difference in the two diagrams is that $RE \cap coRE$ has the name $RECURSIVE$, but $NP \cap coNP$ has not been assigned a name. If this diagram is correct then there are several types of hard problems which are not NP-complete. In particular, there may be problems which are in $NP \cap coNP$ but are not in P.

Until recently Composite-Number was a candidate problem for this status. We know that Composite-Number is in NP because we can show that a number is composite by guessing a factor and then proving that it is a factor by dividing which takes polynomial time in the number of digits. The complement of Composite-Number is Primes. While it is not immediately obvious, for every prime number there is a proof that the number is prime and the length of the proof is bounded by a polynomial in the log of the number. So the complement of composite is also in NP, and composite is in $NP \cap coNP$. But everyone (including the National Security Agency) assumes that composite and prime do not have reasonable algorithms. Unfortunately, proving that composite/prime is not in P is probably very difficult since this would imply $P \neq NP$.

The status of Primes has recently been clarified by Agrawal *et al*[2002]. (Interestingly, this result was discovered by one professor working with two undergraduates.) They showed that there is a polynomial time algorithm which determines whether or not a number is a prime. At the moment, their algorithm is *not* considered fast enough to be a practical algorithm.

How does $PRIMES \in P$ effect cryptographic schemes? As you may know several cryptographic schemes, like RSA, depend on the difficulty of factoring for their security. The new algorithm does not (at present) effect RSA's security because the algorithm can show that a number is composite without giving a factor. Even if we know that q has a factor, we don't know how to find the factor quickly. Hence, RSA is still secure. (But many people expect that a fast composite algorithm which produces a factor will be found soon. The paranoids assume that NSA has such a factoring algorithm but wants to keep it secret so only they can break RSA coded messages.)

We are now without a reasonable candidate for a problem in $NP \cap coNP$ but not in P. Each of the problems that were known to be in $NP \cap coNP$ have each been shown to be in P. For example, Linear-Inequalities was known to be in $NP \cap coNP$ and it has now been shown to be in P.

LINEAR INEQUALITIES:
INPUT: A set of inequalities of the form

$$a_{1i}\, x_1 \; + a_{2i}\, x_2 \; + \cdots + \; a_{ni}\, x_n \; \leq \; b_i$$

for $i = 1$ through m.
QUESTION: Is there a set of rational numbers $x_1, x_2, \ldots, x_n$ which simultaneously satisfy all m inequalities?

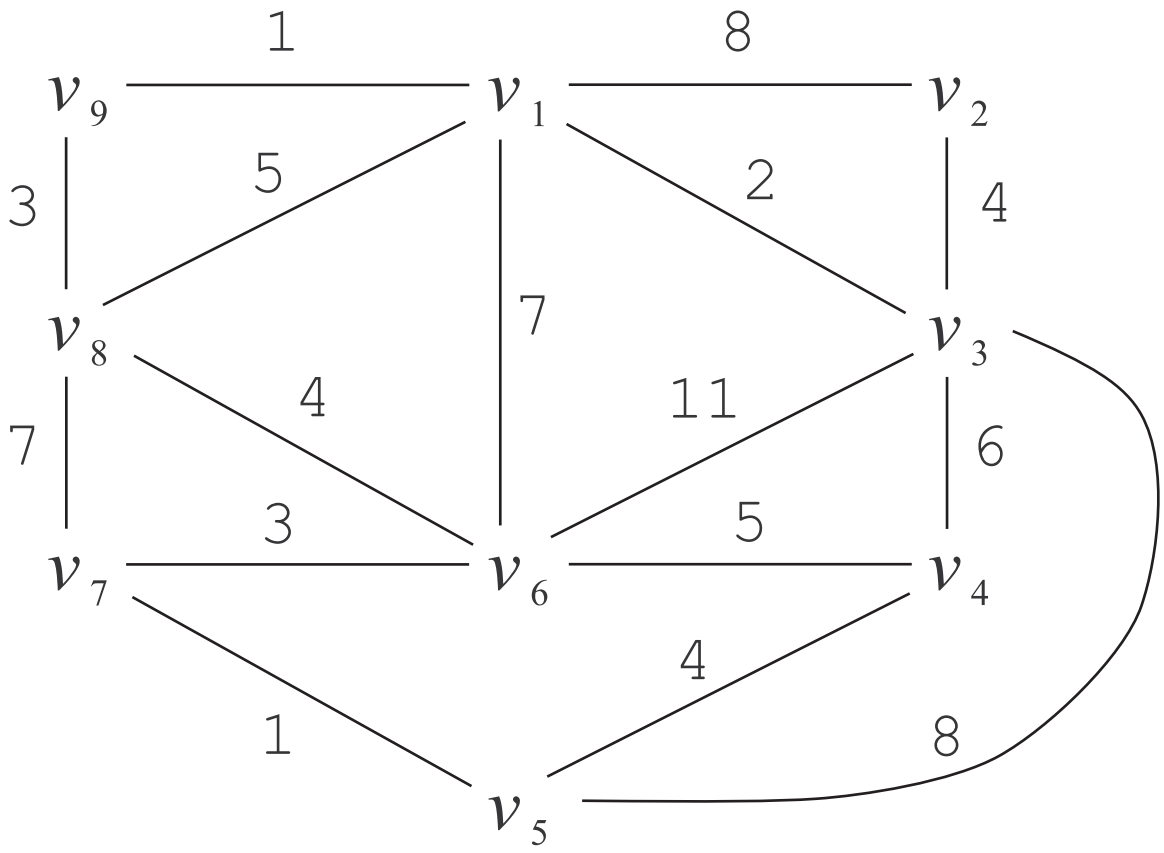
We conclude by mentioning that our diagram of the world of NP may be incorrect, but there is some reasonable circumstantial evidence to support it.

1.8 Exercises

Ex 1.1. Show that if Hamiltonian circuit is NP-complete then Hamiltonian path is NP-complete.

Ex 1.2. Show that if Hamiltonian path is NP-complete then Hamiltonian circuit is NP-complete.

Ex 1.3. For the following graph use the minimum spanning tree method to construct a short traveling salesman circuit. Assume that any missing edges have the longest distance consistent with the triangle inequality. How close to the minimum circuit is your constructed circuit?



Ex 1.4. Show that the following variant of the Towers of Hanoi is in the class P. INPUT: A configuration, CON, of the Towers of Hanoi puzzle with n disks. QUESTION: Does CON occur in the sequence of configurations used to move n disks from tower A to tower C using the minimal number of moves?

Ex 1.5. Show that the following *Graph Isomorphism* problem is in the class NP. INPUT: Two graphs G_1 and G_2. QUESTION:Can the vertices of G_1 be relabeled so that the relabeled G_1 is identical to G_2.