

CDM

Nondeterministic Computation

Klaus Sutner
Carnegie Mellon University
www.cs.cmu.edu/~sutner

Battleplan

- Beyond Tractability
- Guess and Verify: NP
- Some NP Problems
- A Definability Definition
- Nondeterministic Turing Machines
- Deterministic Simulation
- Beyond NP

Tractable vs. Nearly Tractable

A clear understanding of the difference between decidability and tractability first arose in the 1960's as a result of the increasing use of computers. Since then many algorithms have been developed that solve problems in polynomial time.

In some cases, finding the right algorithm may involve decades of research. The latest example is the polynomial time primality testing algorithm by Agrawal, Kayal and Saxena.

As it turns out, there is a rather large class of problems that are very nearly tractable but have resisted all efforts to find polynomial time algorithms. There are trivial exponential time algorithms, but the step down to polynomial time seems exceedingly difficult – though no one has been able to establish appropriate lower bounds.

These problems form the famous class NP, see below for a definition. Here is an informal definition and some examples.

NP, Informally

NP is a class of decision problems, so we are only dealing with Yes/No answers.

The idea is that a problem is in NP if we can "solve" it as follows. Given some input x :

- Guess a witness y , a bit-string of length polynomial in the length of x .
- Verify some property $P(x, y)$ – which property can be tested in polynomial time.

We insist that for all Yes-instances there is an appropriate witness, but for No-instances no such witness exists.

So if we only could clairvoyantly produce the right witness, the whole decision procedure would take polynomial time.

But a brute-force search over all potential witnesses takes exponential time.

Membership in NP

Hence, to establish membership of a decision problem in NP we have to do two things:

- Determine what the witness should be that we are going to "guess" nondeterministically.
- Make sure that all the operations needed in verification are indeed polynomial time.

In many cases, both steps are quite straightforward, even trivial at times.

Alas, there are some examples of problems in NP where either one of the two tasks is difficult and requires some effort.

► But first a few easy examples.

Vertex Cover

Here is a classical example of an NP problem in graph theory.

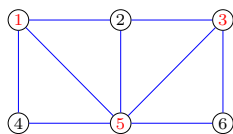
Definition 1. Let $G = \langle V, E \rangle$ be an undirected graph.

A **vertex cover** for G is a set $C \subseteq V$ such that $\forall \{v, u\} \in E (v \in C \vee u \in C)$.

Note that V is always a vertex cover, so the interesting problem is to find a small cover. There are several variants of the problem:

- Decision Version: given G and a bound k , is there a cover of size k ?
- Function Version: given G , compute the lexicographically first cover of minimal size.
- Search Version: given G , compute a cover of minimal size.

Cheap Example



Minimal cover: 1, 3, 5.

You can think of this as a problem of trying to eavesdrop on all conversations in a network: we would like to place the minimal number of listening devices on some of the nodes of the network.

Tackling Vertex Cover

Let's deal with the decision version, which is a candidate for membership in NP.

We can check all $\binom{n}{k}$ subsets of size k . Each check is $O(n^2)$. Unfortunately, $\binom{n}{k}$ is not polynomial in n for variable k , so this is not polynomial time.

Of course, there might be smarter ways. E.g., we could start with the highest degree vertices. Surprisingly, none of the methods to approach this problem seem to succeed to push it down into polynomial time.

But Vertex Cover is trivially in NP: we guess the vertex cover and verify in polynomial time that it is indeed a cover and has the right size.

Exercise 1. Explain why the highest degree method does not always produce a minimal vertex cover. Try to come up with a better version.

Travelling Salesman (TSP)

We are given a n by n matrix of distances $d(i, j) \in \mathbb{N}^+$.

Think of $[n]$ as a collection of cities, and $d(i, j)$ as the distance between city i and city j .

Formally, a tour is permutation π of $[n]$. For simplicity assume $\pi(1) = 1$. Thus we go from city 1 to $\pi(1)$, then $\pi(2)$ and so on till $\pi(n)$, then return to city 1.

The cost of this tour is

$$\text{cost}(\pi) = \sum_i d(\pi(i), \pi(i+1)) + d(\pi(n), 1)$$

We want to find a tour of minimal cost.

In the decision version there is a bound K and we ask whether there is a tour of cost at most K .

Variants

In the Triangle TSP the distances are required to be symmetric and conform to the triangle inequality:

$$d(i, j) \leq d(i, k) + d(k, j).$$

In the Geometric TSP we are given a finite set S of points in the plane with integer coordinates, and the distances are defined to be the standard Euclidean distance between these points.

Thus, Geometric TSP is a special case of Triangle TSP (we don't really have to compute square roots!)

All these problems are trivially in NP, and not known to be in P.

Hamiltonian Cycle

A **Hamiltonian cycle** in an undirected graph G is a cycle that contains every vertex of G exactly once.

Don't confuse Hamiltonian and **Eulerian cycles**: an Eulerian cycle is required to use every edge of the graph exactly once. While this may seem to be a minor difference, it most emphatically is not:

- One can test in linear time whether a graph has an Eulerian cycle: the graph only needs to be connected, and every node must have even degree.
- But there is no known polynomial time test for Hamiltonicity.

Exercise 2. Come up with a reasonable algorithm to test Eulericity (or is it Eulerian?). Your algorithm should construct an Eulerian cycle if one exists, and return "No" otherwise.

Evans-Minieka Algorithm

With a little bit of effort, one can even construct an Eulerian cycle in linear time.

Suppose we have a procedure `cycle(x)` that, given any vertex x , returns some arbitrary cycle anchored at vertex x . This is easy to do in time linear in the size of the cycle provided the graph is given in some reasonable representation such as adjacency lists.

Using `cycle(x)` as a subroutine we can construct an Eulerian cycle as follows.

- Pick an arbitrary vertex x and use `cycle(x)` to obtain a cycle C anchored at x .
- Remove all the edges on C from G .
- Traverse C until a vertex y is found that still has edges.
- Splice the output of `cycle(y)` into C .
- Repeat till all edges are used up.

Exercise 3. Fill in the details in the algorithm and prove correctness.

Polynomials mod 2

Suppose we have a system of polynomial equations over \mathbb{Z}_2 :

$$P_i(x_1, x_2, \dots, x_n) = 0 \quad i = 1, \dots, m.$$

Note that one can check in polynomial time if there exists a solution to a single polynomial equation over \mathbb{Z}_2 . But if we look for a solution for the whole system of equations the problem is not known to be in \mathbb{P} .

On the other hand, the problem is trivially in \mathbb{NP} : guess the solution and evaluate all the polynomials on it to verify correctness.

Exercise 4. Find a polynomial time test for a single polynomial equation over \mathbb{Z}_2 .

A Pebbling Game

Here is a slightly more complicated example.

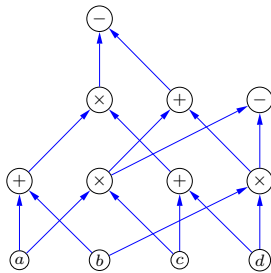
Consider an acyclic directed graph G . You want to place pebbles on all the nodes with out-degree 0 subject to the following rules:

- Initially all nodes are empty (unpebbled).
- Each node is pebbled at most once.
- A new pebble can be placed on node x only if there are pebbles on all nodes y such that $(y, x) \in E$.
- A pebble can be removed at any time.

Of course, there is a catch: you want to use the minimal number of pebbles (removed pebbles can be reused). So we have a decision problem: can G be pebbled with K pebbles?

The Real Prey

The real goal is to minimize temporary storage when executing a straight-line program. Recall the circuit corresponding to a SLP for complex multiplication:



Register Allocation

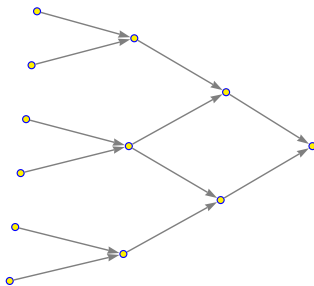
In general, the nodes of the digraph correspond to computational tasks, and execution of node/task x requires the results of all tasks y such that $(y, x) \in E$.

For simple arithmetic operations, the pebbles correspond to registers, so this is really about register allocation.

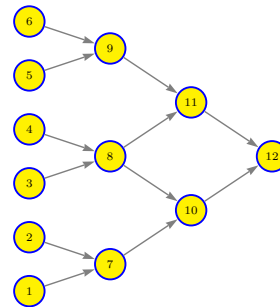
Recall that there is a simple linear time algorithm to "sequentialize" the tasks using just one processor: topological sorting.

But this does not answer our problem: it just says "execute the tasks in such and such order" and ignores resource issues entirely.

A Simpler Example



Bad Solution



How Many Pebbles?

Given a permutation $\pi = x_1, \dots, x_n$ of the vertices which describes a certain pebbling order, the corresponding minimal pebble sets are determined as follows:

$$P_0 = \emptyset$$

$$P_t = P_{t-1} + x_t - \text{all useless pebbles in } P_{t-1}$$

A pebble at y is useless if all vertices x such that $(y, x) \in E$ are already taken care of.

The size of P_t is considered to be the number of pebbles before the removal of the useless ones (that is, $|P_{t-1}| + 1$).

Exercise 5. Verify that given the permutation π we can compute in polynomial time the optimal pebbling strategy that pebbles the vertices in order given by π .

Different Versions

A minor variant of the problem allows one to shift one pebble from y to x given an edge $(y, x) \in E$.

Makes essentially no difference (just one pebble).

A much different version of the Pebbling game allows one to re-pebble a vertex, arbitrarily often.

This corresponds to recomputing a value, rather than storing it in a register. Needless to say, there is a trade-off: recomputation takes longer but may well require fewer pebbles than our version.

This is actually very important for simulations translating time into space, but we won't go there.

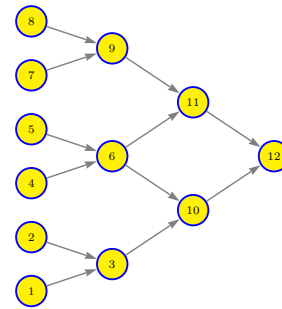
Bad Solution, Contd.

The pebble sets for the bad example. The top row indicates the stages of the construction.

0	1	2	3	4	5	6	7	8	9	10	11	12
-	1	1	1	1	1	1	3	5	7	8	10	12
		2	2	2	2	4	6	8	9	11		
			3	3	3	5	7	9	10			
				4	4	4	6	8				
					5	5	7					
						6						
0	1	2	3	4	5	6	7	6	5	4	4	3

The bottom row indicates the number of pebbles used at this stage, before removal of unneeded pebbles.

Optimal Solution



Requires only 5 pebbles.

Pebbling is in NP

As usual, we consider the decision version:

Problem: Pebbling (Decision Version)

Instance: A directed acyclic graph G and a bound K .

Question: Can G be pebbled with at most K pebbles?

To see membership in NP we guess the right pebbling strategy S and verify that it requires no more than K pebbles.

Ignoring details, this test can certainly be done in polynomial time.

In fact, we could slightly improve matters by guessing a permutation of the vertices and then use our previous result that allows us to compute the best possible strategy using the order given by the permutation.

Of course, there are still too many permutations . . .

Back To Inequivalence of LOOP(1)

Recall that two LOOP(1) programs are inequivalent if they produce different output on some input \vec{x} of limited size. The size of the distinguishing input is determined by two numerical parameters β and μ which in turn are polynomial in the size of the given programs.

But then Inequivalence is in NP:

We can guess the right \vec{x} and then verify in polynomial time that the output of the two programs on input \vec{x} is indeed different.

Of course, there are exponentially many choices for \vec{x} and brute-force search is exponential.

Note that it this is an example where membership in NP is far from obvious; it might well be the case that the first witness for inequivalence has exponential size (so we could not use a polynomial guessing mechanism to find it).

From NP to Intractability

Membership in NP is **not** a lower bound, every polynomial time problem is trivially in NP.

But the ones we have just seen seem to be outside of P: a great many people have tried very hard for at least three decades to find fast algorithms, and they have all failed.

While separating P from NP is currently still an open problem (and arguably the most important problem in theoretical computer science, some would even give it very high rank in mathematics in general), there is quite a bit of evidence that suggests these problem are indeed not tractable.

We need a bit more machinery to compare the difficulty of problems to explain this evidence, though.

Before we do this, though, here is a more formal definition of NP.

Nondeterministic Polynomial Time

We can make the idea of "guess and verify" precise as follows.

Definition 2. A set $A \subseteq \Sigma^*$ is in **nondeterministic polynomial time, NP**, if there is a polynomial time decidable relation R and a polynomial p such that

$$x \in A \iff \exists w (|w| \leq p(|x|) \wedge R(w, x))$$

A decision problem is in NP if its set of Yes-instances is.

As always, we assume that instances are coded in some natural way.

Example 1. *Vertex Cover, Hamiltonian Cycle, Travelling Salesman, Pebbling and LOOP(1) Inequivalence* are all in NP.

A Technical Detail

Note that the running time of the algorithm for R is polynomial in $|w| + |x|$ but the input for our decision problem is just x .

In order to avoid cheating by witnesses of the form

$$w = \underbrace{00 \dots 000}_\text{lots} W$$

where W is the real witness we have to require short witnesses: $|w| \leq p(|x|)$ for some polynomial p .

Exercise 6. *What would happen if we dropped the bound on witness length in the definition of NP? What class of problems would we describe?*

The Open Problem

It is clear that $P \subseteq NP$, and it seems very likely that $P \neq NP$.

But, despite 30 years of effort, no one has been able to prove that this is the case.

The $P = NP$ question is one of the 7 open problems for which you can win \$1,000,000 from the Clay Foundation.

And $P = NP$ is number 3 on Smale's list.

The Classical Counterpart

As we have seen, computably enumerable sets can be obtained defined in terms of a similar Σ_1 definition (or projection operation if you prefer the non-logical approach):

$$x \in W \iff \exists y R(x, y)$$

where R is a decidable relation.

The witness for the membership of x in W is y , and the verification is handled by a decidable predicate. There are no resource bounds, but otherwise this is exactly the same type of definition.

A semi-algorithm for W halts on all Yes-instances but fails to produce any output for No-instances.

Asymmetry

We have the same asymmetry in membership versus non-membership that tends to confuse people.

Classical computation theory:

Lemma 1. *The decidable sets are closed under intersection, union and complements. The semi-decidable sets are only closed under intersection and union.*

Translating this into resource-bound computation theory we would like:

Lemma 2. *The polynomial time decidable sets are closed under intersection, union and complements. The nondeterministic polynomial time decidable sets are only closed under intersection and union.*

There is a major glitch, though: non-closure under complementation would show that $P \neq NP$, so there is little hope to establish this result.

Closure Properties

Here is the easy part.

Lemma 3. *The class NP is closed under intersection and union.*

Proof.

One can simply guess the two witnesses, and then run the two verification procedures. In the end, perform a simple Boolean \vee or \wedge operation on their outputs.

The whole operation is still polynomial time since polynomials are closed under addition. \square

This argument is actually a bit easier than for semi-decidable sets since there is no problem with halting (union for semi-decidable sets requires to run the machines in parallel by interleaving computations, here we can go sequentially).

And Complement?

But complementation runs into difficulties:

$$x \notin A \iff \forall w, |w| \leq p(|x|) (\overline{R}(w, x))$$

Here \overline{R} is the negation of the original predicate R and thus still polynomial time decidable.

There is no obvious reason why it should be possible to replace the universal quantifier by an existential one.

The class of problems with this characterization is called co-NP. A typical example is testing whether a boolean formula is a tautology.

Of course, if $\text{P} = \text{NP}$ then NP is indeed closed under complementation and we have $\text{P} = \text{NP} = \text{co-NP}$.

Actual Computability is Hard

Note that the analogous problem in classical recursion theory is much easier:

The classes of decidable, semi-decidable and co-semi-decidable sets are all distinct: the Halting set K is semi-decidable but not decidable, and therefore its complement is co-semi-decidable but not semi-decidable.

But the additional constraint of performing the computations in polynomial time makes things significantly more complicated. In particular, the classical diagonalization proof for the undecidability of K simply does not carry over.

Exercise 7. *Try to prove $\text{P} \neq \text{NP}$ by diagonalization and discuss what goes wrong.*

Nondeterministic Machines

So, NP compared to P is very similar to semi-decidable compared to decidable.

One might wonder if the machinery of time and space complexity classes can be adjusted to capture collections such as NP.

In other words, is there a natural machine model for NP?

There is: one can introduce nondeterministic machines, but they are a bit more strange than their classical, deterministic counterparts.

Nondeterministic machines were first introduced by Rabin and Scott in the context of finite state machines. It has since become customary to describe complexity classes in terms of nondeterministic machines (we will talk about nondeterministic FSMs later).

The basic idea is simple: given a particular configuration, the machine may be able to perform several possible next moves.

Nondeterministic Turing Machines

We want to define a *nondeterministic polynomial time Turing machine acceptor*, a type of machine that can solve precisely all problems in NP.

Classical Turing machines have transition functions of the form

$$\delta : Q \times \Sigma \rightarrow Q \times \Sigma \times \{\pm 1, 0\}.$$

To introduce choice (nondeterminism) we switch to a more general transition relation

$$\delta \subseteq Q \times \Sigma \times Q \times \Sigma \times \{\pm 1, 0\},$$

a relation that is no longer required to be single-valued?

The idea being that $(p, a, q, b, d) \in \delta$ means that the machine could go to state q , write symbol b and move the head by d , but could possibly also do something else.

Acceptors Only

It is very difficult to make sense out of nondeterministic transducers: if the machine computes several possible function values, which one is right? The left-most one (think of the tree of all possible computations)? The most frequent answer? The one requiring the least number of steps? A mess.

But for acceptors there is a relatively smooth definition.

Definition 3. *A nondeterministic Turing machine M accepts input x if there is at least one computation ending in the accepting state q_Y .*

Note that because of nondeterminism the "good" computation leading to state q_Y need not be unique.

Popular Misconceptions

The definition most emphatically does not say that all computations are required to lead to acceptance; in general there will be computations that end in state q_N .

The definition does not require a substantial fraction of all computations to lead to acceptance; all we need is one good computation.

Insisting on many accepting computations leads to the very important notion of a probabilistic algorithm, but \mathbb{NP} per se has nothing to do with randomness.

When in doubt, revert to the original definition – don't assign mysterious properties to nondeterministic Turing machines.

Computation Trees

A computation of a deterministic TM is simply a linear sequence of IDs:

$$C_x = C_0, C_1, \dots, C_{t-1}, C_t = C'$$

But for a nondeterministic machine we are dealing with a tree:

- the root is C_x
- node C has as children all C' such that $C \vdash_M^1 C'$.

Note that the tree is finitely branching.

Exercise 8. Show that one can safely assume that the computation tree of a nondeterministic Turing machine is binary.

Many-Worlds Interpretation

A nondeterministic Turing machine is not a model of any classical computational device: it makes nondeterministic choices.

Stealing an idea from quantum physics, you can think of a NTM as running in a multiverse.

Every time the machine makes a nondeterministic choice, the world splits into two strands, two branches that evolve in parallel.

After n choices, we have 2^n strands, 2^n copies of the machine running in parallel.

If and when one of them accepts, the whole ghostly affair ends in acceptance, and rejection otherwise.

Choice Sequences

It is easy to see that choices can be limited to binary.

Given a nondeterministic acceptor M with binary choices and input x , every branch in the computation tree is determined by a bit-sequence $S \in 2^n$ where $n \leq T_M(x)$:

Every time M makes a nondeterministic move, take the next bit in S to decide whether to go left or right.

Note that we can test whether S is a good choice sequence (i.e., leads to acceptance of M) a deterministic Turing machine M' . So we are back to our definability approach:

$$\exists S \in 2^n (M \text{ accepts } x \text{ via } S).$$

Exercise 9. Prove that binary choices suffice.

Nondeterminism is Useless

Is there anything a nondeterministic acceptor can do that a deterministic one cannot?

Yes and no.

Theorem 1. Deterministic Simulation

Every nondeterministic Turing machine acceptor can be simulated by a deterministic one.

Proof.

Use the machine $M'(S, x)$ from above, and systematically check all possible choice sequences $S \in 2^n$ where n is the running time bound of M on x . If one of these simulated computations ends in acceptance of M , accept likewise, otherwise reject.

Clearly, all of this can be handled by a deterministic Turing machine. \square

Of course, the new machine will not run in polynomial time.

Nondeterminism is Great

How much does it cost to eliminate the nondeterministic search? The running time of the deterministic simulator machine is

$$O(2^c f(n))$$

where $c > 0$ is some constant, and f is the running time of the nondeterministic machine.

In other words, brute force deterministic simulation is exponential.

For abstract computations this exponential slow-down is trifling. But for real algorithms it makes a huge difference.

A warning: the fact that the obvious deterministic simulation method is exponential does not mean that there might not be a better algorithm.

This is most emphatically not a lower bound result.

NP is Everywhere

The reason NP is important is that there are many interesting combinatorial problems that are easily seen to be in NP, but at present are not known to have a polynomial time algorithm.

Moreover, most of these have the property that they indeed cannot have a polynomial time algorithm unless already $\mathbb{P} = \text{NP}$, an unlikely proposition.

The bible for NP-completeness:

M. R. Garey and D. S. Johnson
Computers and Intractability
A Guide to the Theory of NP-Completeness
Freeman, 1979

Contains hundreds of examples of problems in NP (and worse). Many more are known today, but these are the most important ones.

Nondeterministic Complexity Classes

We can easily extend the definitions of deterministic time/space complexity classes to nondeterministic machines.

Bear in mind, though, that the time and space bounds have to apply to all possible computations in this setting – we are dealing with computation trees rather than paths.

Definition 4. Let $f : \mathbb{N} \rightarrow \mathbb{N}$ be a function.

$$\text{NTIME}(f) = \{ \mathcal{L}(M) \mid M \text{ a nondeterministic TM, } T_M(n) = O(f(n)) \}$$

$$\text{NSPACE}(f) = \{ \mathcal{L}(M) \mid M \text{ a nondeterministic TM, } S_M(n) = O(f(n)) \}$$

As before, this generalizes easily to $\text{NTIME}(\mathcal{F})$ and $\text{NSPACE}(\mathcal{F})$.

So $\text{NP} = \text{NTIME}(\text{polynomials})$.

Some Results

From the definitions

$$\text{TIME}(f) \subseteq \text{NTIME}(f) \subseteq \text{NSPACE}(f)$$

$$\text{SPACE}(f) \subseteq \text{NSPACE}(f)$$

More interesting is the question how much deterministic time/space is required to capture a nondeterministic class.

Theorem 2. Assume that $\log n \leq g(n)$. Then

$$\text{NTIME}(f) \subseteq \text{SPACE}(f)$$

$$\text{NSPACE}(g) \subseteq \text{TIME}(2^{O(g)})$$

Exercise 10. Prove the last theorem by simulating a nondeterministic machine on the left, carefully keeping track of the resources required to do so.

Savitch's Theorem

Theorem 3. Assume that $\log n \leq g(n)$. Then

$$\text{NSPACE}(g(n)) \subseteq \text{SPACE}(g(n)^2).$$

Sketch of proof.

Again this is a simulation result.

The trick is to use a divide-and-conquer approach: a computation of length t is broken up into two subcomputations of length $t/2$.

Since the target is a space class and space, unlike time, can be reused, one can show with some effort that the divide-and-conquer algorithm requires no more than $g(n)^2$ space.

□

Immermann-Szelepsényi Theorem

Here is a result that was proven almost simultaneously indepently by two researchers in the late 80's.

Theorem 4. Assume that $\log n \leq g(n)$. Then $\text{NSPACE}(g)$ is closed under complements.

In particular $\text{NSPACE}(n) = \text{co-NSPACE}(n)$.

This solved a long-standing open problem in language theory: the complement of every context-sensitive language is again context-sensitive.

The argument uses counting and is based on the following result.

Lemma 4. Let G be a graph of size n , and s a vertex in G . Then the number of nodes reachable from s can be computed in NLOG.

Summary

- In analogy to semi-decidability one can introduce the class NP of nondeterministic polynomial time solvable problems.
- There are many practical computational problems in P and NP.
- Alas, life becomes much harder in low complexity classes such as P and NP: it is a major open problem whether $\mathbb{P} = \text{NP}$.
- There is a natural hierarchy of nondeterministic time and space complexity classes, analogous to their deterministic counterparts.