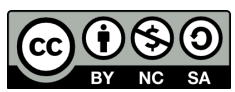
UCT

Non-Determinism and $\mathbb{NP}$

Klaus Sutner

Carnegie Mellon University

Spring 2024



- SAT is a natural decision problem that appears to be just a little bit outside of the realm of feasible computation.
- Lots of combinatorial problems are reducible to SAT, so pinning down its complexity is of great theoretical and practical interest.
- One question we need to answer: is there a natural complexity class where SAT lives?

Recall from HW two languages based on Kleene's star operation:

$$L^\star = \{\, x_1 x_2 \dots x_k \mid k \geq 0, x_i \in L \,\}$$

$$L^\star_\# = \{\, x_1 \# x_2 \# \dots \# x_k \mid k \geq 0, x_i \in L \,\}$$

where $\#$ is a new symbol not in Σ.

Claim: If L is in $\mathbb{P}$, then both $L^\star$ and $L^\star_\#$ are also in $\mathbb{P}$.

This is fairly easy to see for $L^\star_\#$: here we are given the factors x_i directly and can just check that they are all in L.

But $L^\star$ is more complicated: we cannot simply enumerate all possible factorizations

$$x = x_1 x_2 \dots x_k$$

since there are exponentially many.

> **Wild Idea I:** how about we change our algorithms so that they can guess the right factorization, and then verify that it works?

This will horrify an algorithms person, but, as we will see, it works out just fine. We will switch from ordinary, deterministic algorithms to nondeterministic ones.

Another way to approach this problem is to try to exploit the old decidable versus semidecidable distinction, the bottom level of the arithmetical hierarchy (we will deal with the full hierarchy later).

> **Wild Idea II:** Suppose polynomial time corresponds to decidable. What is the analogue of semidecidable?

Well, we need to do a projection. That should produce a class that closely mirrors semidecidable.

In the context of languages, let us define projections as follows: Given a marked language

$$L \subseteq \Sigma^\star \# \Sigma^\star$$

where $\#$ is a special symbol not in Σ we set

$$\mathsf{proj}(L) = \{\, x \in \Sigma^\star \mid \exists\, w \in \Sigma^\star\, (w\#x \in L) \,\}$$

For all practical purposes, this is the same as going from $\Sigma^\star \times \Sigma^\star$ to $\Sigma^\star$, but avoids Cartesian products.

We call w the witness and x the instance of a string $w\#x$.

For example, $\mathsf{proj}\big(\{\, a^i\#b^i \mid i \geq 0 \,\}\big) = b^\star$. Here projection reduces complexity, but that is far from true in general: recall that all semidecidable sets are projections of decidable ones.

Unfortunately, projecting $\mathbb{P}$ we get all semidecidable sets.

To see why, consider the following marked language, a tame version of the Halting set:

$$U = \{\, 0^\sigma \# e \mid \{e\}_\sigma(e) \downarrow \}$$

By adding a prefix 0^σ we greatly reduce the complexity of the language (this is actually an important technique called padding):

Claim

U is polynomial time.

But: proj(U) is the Halting set, the mother of all undecidable sets and complete for semidecidable sets (level Σ_1 in the AH).

Projecting $U = \{\, 0^\sigma \# e \mid \{e\}_\sigma(e) \downarrow \}$ is essentially the same as searching for some stage σ such that $\{e\}_\sigma(e) \downarrow$.

In general, this is an unbounded search and promptly wrecks decidability.

If we are interested in low complexity classes we cannot allow unbounded searches, we have to truncate the search somehow. It's a fair guess that we should only conduct a polynomial search: if nothing is found by then, we give up.

We replace full projections with polynomially bounded projections:

$$\text{proj}_P(L) = \{\, x \in \Sigma^\star \mid \exists\, w \in \Sigma^\star \left(|w| \leq p(|x|) \wedge w\#x \in L \right) \}$$

where p is some polynomial. Thus, we require the witness w to be relatively short wrto the instance x. We usually don't bother to state the polynomial p explicitly.

Definition

The collection of all polynomially bounded projections of $\mathbb{P}$ is called nondeterministic polynomial time, in symbols $\mathbb{NP}$.

Intuitively, $\mathbb{P}$ corresponds to decidable in the arithmetical hierarchy, and $\mathbb{NP}$ corresponds to semidecidable.

Example 1

Suppose we encode ugraphs as $n \times n$ Boolean strings, together with a number k in binary as

$$x = \underbrace{u_1 u_2 \ldots u_{n^2}}_{\text{graph}} \underbrace{k_1 \ldots k_{\log n}}_{\text{number}} \in \mathbf{2}^\star$$

Filtering out the strings for which the graph does indeed have a vertex cover of size k we have a natural encoding of the vertex cover decision problem. Call this language VC.

Claim

VC *is in* $\mathbb{NP}$.

Just add a bitvector for the cover to get an appropriate marked language:

$$w\#x = w_1 w_2 \ldots w_n \,\#\, u_1 \ldots u_{n^2}\; k_1 \ldots k_{\log n}$$

This language is easily in $\mathbb{P}$, and VC is its polynomially bounded projection: $n = |w| \leq |x| = n^2 + \log n$.

Example 2

Similarly we can encode a propositional formula φ as some bit-string $\langle\varphi\rangle = u_1 u_2 \dots u_n$ (the encoding details don't matter). Then SAT can be thought of as the language

$$\mathsf{SAT} = \{\, \langle\varphi\rangle \mid \varphi \text{ is satisfiable} \,\} \subseteq \mathbf{2}^\star$$

Claim

SAT *is in* $\mathbb{NP}$.

Just add a bitvector for the satisfying truth assignment to get the marked language:

$$w\#u = w_1 w_2 \dots w_m \# u_1 \dots u_n$$

where $m \leq n$ is the number of variables in φ. Clearly, the last language is in $\mathbb{P}$, and SAT is its polynomially bounded projection.

Exercise

Go through the various combinatorial problems we have mentioned (Independent Set, Clique, Subset Sum, Hamiltonian Cycle) and verify that they are all in $\mathbb{NP}$.

Exercise

How about Tautology?

Exercise

How about Inequivalence of programs?

Problem: ***Inequivalence***
Instance: *Two total programs e_1 and e_2.*
Question: *Is there some x such that $\{e_1\}(x) \neq \{e_2\}(x)$?*

Lemma

The class $\mathbb{NP}$ is closed under intersection and union.

Proof.

Let $K_i = \mathsf{proj}_P(L_i) \in \mathbb{NP}$ where $L_i \subseteq \Sigma^\star \# \Sigma^\star$ are the polynomial time witness languages, $i = 1, 2$.

Then $K_1 \cup K_2 = \mathsf{proj}_P(L_1 \cup L_2)$, where the bounding polynomial is suitably chosen.

Intersection requires a slight modification $K_1 \cap K_2 = \mathsf{proj}_P(L_1 \sqcap L_2)$.

□

Exercise

What is this modification? What is the mysterious operator $\sqcap$?

And Complement?

Again, just like with semidecidable sets, complementation runs into difficulties:

$$x \notin K \iff \forall\, w, |w| \leq p(|x|) \left(w\#x \notin L \right)$$

Of course, $w\#x \notin L$ is still polynomial time decidable.

But, there is no obvious reason why it should be possible to replace the universal quantifier by an existential one, with a polynomial bound: it seems like we need to check exponentially many w.

So we get a class of problems is obtained by complementation, corresponding to Π_1 in the arithmetical hierarchy.

We write co-$\mathbb{NP}$ for this class, the negation of all problems in $\mathbb{NP}$.

$$\Sigma_0^P = \Pi_0^P = \mathbb{P}$$

$$\Sigma_{n+1}^P = \mathsf{proj}_P(\Pi_n^P)$$

$$\Pi_n^P = \mathsf{comp}(\Sigma_n^P)$$

$$\Delta_n^P = \Sigma_n^P \cap \Pi_n^P$$

$$\mathsf{PH} = \bigcup \Sigma_n^P$$

This is the same construction as for the arithmetical hierarchy, except that

- we start at $\mathbb{P}$, and
- we use polynomial bounded projections.

For the arithmetical hierarchy, we know that $\Sigma_n \neq \Sigma_{n+1}$.

For the polynomial hierarchy, one[†] suspects that $\Sigma_n^P \neq \Sigma_{n+1}^P$.

But we don't even know that $\Sigma_0^P \neq \Sigma_1^P$, i.e., whether $\mathbb{P} \neq \mathbb{NP}$.

Similarly, we don't know that $\Sigma_1^P \neq \Pi_1^P$, i.e., whether $\mathbb{NP} \neq$ co-$\mathbb{NP}$.

Remember, I always said that, in many ways, the classical theory is easier than the complexity version. Adding resource constraints makes life much, much harder.

[†]Not necessarily everyone, though. Some people think that the hierarchy collapses.

Our definition of $\mathbb{NP}$ is based on the geometric concept of projection (augmented a bit), so one might wonder if there is some alternative machine-based definition.

The main idea here is nondeterminism, first introduced in the seminal paper

> M. Rabin, D. Scott
>
> Finite Automata and Their Decision Problems
>
> IBM Journal of Research and Development
> Volume 3, Number 2, Page 114 (1959)

This is about finite state machines, but how about Turing machines?

We want to define a nondeterministic Turing machine acceptor.

Classical Turing machines have transition functions of the form

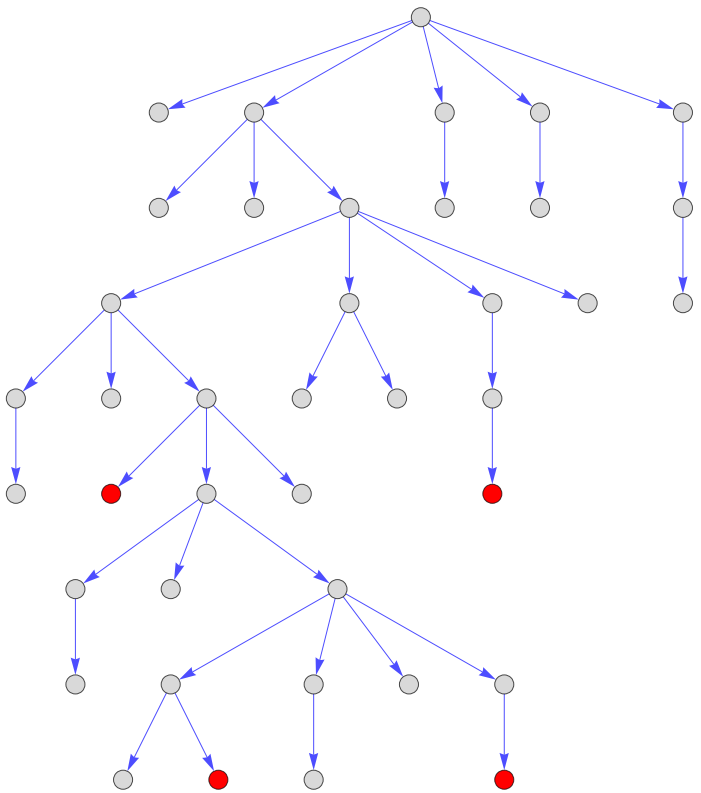
$$\delta : Q \times \Sigma \nrightarrow Q \times \Sigma \times \{\pm 1, 0\}.$$

To introduce nondeterminism, we switch to a more general transition relation

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

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, or get stuck.

In a nondeterministic Turing machine the one-step relation $C \vdash^1_{\mathcal{M}} C'$ is no longer single-valued. Hence, instead of a linear sequence, we get a computation tree $\mathcal{T}_x$.



Critical Question: how does this kind of machine compute?

If we are interested in computing a function $f : \Sigma^\star \to \Sigma^\star$ there is a problem: what if different branches come up with different answers? This is quite tricky.

But there is one easy scenario: if we want a decision algorithm, a function $f : \Sigma^\star \to \mathbf{2}$, we can use the following definition:

The machine accepts x if there is at least
one branch in $\mathcal{T}_x$ ending in the accept state.

This is a direct copy of the definition for nondeterministic finite state machines, which definition we already know to make perfect sense. Think about the two killer-apps for FSMs: pattern matching and decision algorithms.

How should one define running time in the setting?

$$T_{\mathcal{M}}(x) = \min\big(|\beta| \mid \beta \text{ accepting branch in } \mathcal{T}_x \big)$$

and similarly for $T_{\mathcal{M}}(n)$.

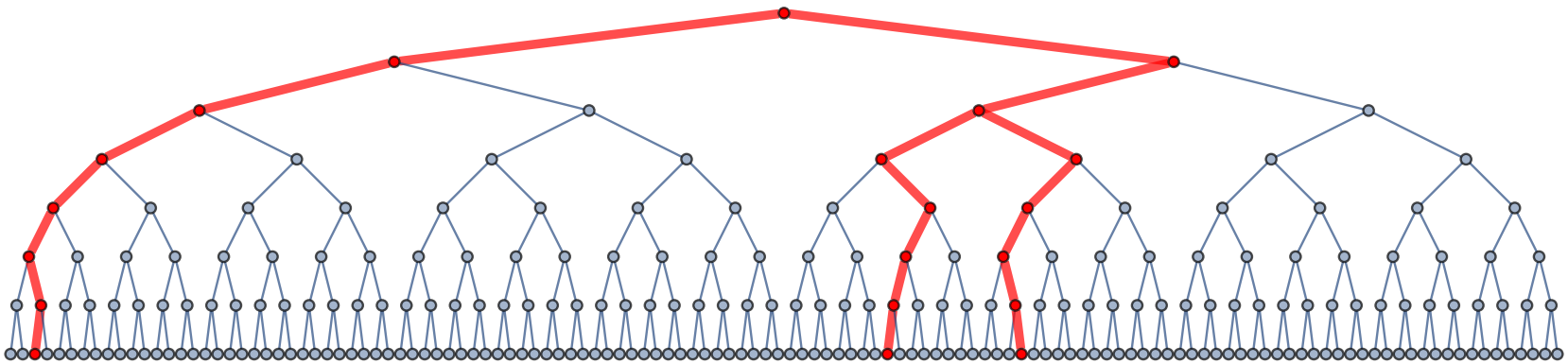
The min really is justified here: we could make the right guesses and choose a short accepting path. Of course, there could be longer accepting paths, but they really don't matter in this model.

By definition, we say that a nondeterministic Turing machine accepts input x if there is at least one accepting branch in the computation tree $\mathcal{T}_x$.

Just to be clear: there may be lots of non-accepting branches in the same tree. In fact, in most natural examples, this will be the case: wrong nondeterministic choices lead to rejection, only the "right" sequence of choices ultimately leads to acceptance.

Note that we are not insisting on a certain fraction of all branches being accepting, the idea behind probabilistic computation; a single one is enough. More on probabilistic algorithms later.

We can clean up nondeterministic Turing machines a bit. First, it is easy to see that the number of choices at each step can be limited to two. Second, it is safe to assume that there are exactly two choices at each step. Yes?



In this situation, every branch in the computation tree is determined by a choice sequence, a bit-sequence $S \in \mathbf{2}^\star$. The accepting branch (if any) has length $T_{\mathcal{M}}(x)$.

Is there anything a nondeterministic acceptor can do that a deterministic one cannot? Yes and no.

Theorem (Deterministic Simulation)

Let $\mathcal{M}$ be a nondeterministic acceptor running in time $t(n)$.
Then $\mathcal{M}$ can be simulated by a deterministic machine.

Proof.

We may assume that $\mathcal{M}$ is a nice binary-choice, uniform branch-length machine as above. We can systematically check all possible choice sequences $S \in \mathbf{2}^{t(n)}$ where n is the length of the input. If one of these simulated computations ends in acceptance, return Yes; otherwise return No.

□

Note that this is perfectly constructive: we can build the deterministic machine from the nondeterministic one.

The last argument also works for nondeterministic TMs that semidecide membership in some semidecidable set.

In this case, we enumerate possible choice sequences $S \in \mathbf{2}^\star$ in length-lex order and check for each one whether the machine accepts with these nondeterministic choices. If so, we halt (and thereby accept).

Otherwise, the simulation runs forever (and thereby rejects).

In CRT this pretty much ends the discussion: nondeterministic machines are perfectly useless, we can always build a better-behaved deterministic counterpart.

How much does it cost to eliminate the nondeterminism in an acceptor? Suppose $t(n)$ is the running time of the nondeterministic machine.

Then the running time of the deterministic simulator machine for civilized time complexities t is certainly

$$O(2^{c \cdot t(n)})$$

where $c > 0$ is some constant depending on the machine. Alas, in general that seems to be the only bound we can come up with.

In other words, brute force deterministic simulation produces an exponential blow-up in compute time. For abstract computations (say, primitive recursive running times) this exponential slow-down is trifling. But for real algorithms it makes a huge difference.

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.

We can now lift the definitions of deterministic time complexity classes to nondeterministic ones.

Definition

Let $f : \mathbb{N} \to \mathbb{N}$ be a time-constructible function.

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

As before, this generalizes easily to $\mathrm{NTIME}(\mathcal{F})$ for a class of functions $\mathcal{F}$.

From the definitions and deterministic simulation we have

$$\mathrm{TIME}(f) \subseteq \mathrm{NTIME}(f) \subseteq \mathrm{TIME}(2^{O(f)})$$

Theorem

$\mathbb{NP} = \mathrm{NTIME}(\mathit{poly})$

Proof. Assume $K = \mathrm{proj}_P(L) \in \mathbb{NP}$ where L is a marked language in $\mathbb{P}$, say, $L \subseteq \bigcup_n \mathbf{2}^{p(n)} \# \mathbf{2}^n$. There is a nondeterministic TM $\mathcal{M}$ that, on input instance $x \in \mathbf{2}^n$, generates a string $w \in \mathbf{2}^{p(n)}$ and then verifies in a polynomial time computation that w is a witness for x, i.e., $w\#x \in L$. Clearly, $\mathcal{M}$ runs in polynomial time.

On the other hand, assume $\mathcal{M}$ is a poly time nondeterministic TM. By using the choice sequence approach from above, we may assume that $\mathcal{M}$ nondeterministically generates a choice sequence w and then checks in deterministic polynomial time that $w\#x$ has some property. Use those strings to form a marked language whose projection is $\mathcal{L}(\mathcal{M})$.

□

The standard interpretation of this approach to "solving" a decision problem is as follows. Given an instance x:

- Guess a witness w, a bit-string of length polynomial in the length of x.
- Verify some polynomial time property of $w\#x$.

Witness w corresponds to a particular branch in $\mathcal{T}_x$, and the polynomial time test just verifies that this branch ends in the accept state. So 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.

Arguably, the most important "small" classes are $\mathbb{P}$ and $\mathbb{NP}$.

$$\mathbb{P} = \mathrm{TIME}(\mathsf{poly})$$

$$\mathbb{NP} = \mathrm{NTIME}(\mathsf{poly})$$

Clearly $\mathbb{P} \subseteq \mathbb{NP}$, but what is the difference? Can we separate the two classes?

More pointedly: is this definition at all useful?

Rant: No, definitions are not arbitrary.

They are right or wrong, just like conjectures, theorems, proofs.

Justification?

So far, there are two good reasons why the definition of $\mathbb{NP}$ might be useful:

- Nondeterminism is critical in the theory and in *applications* of finite state machines. Without it, `grep` and the like would disappear.
- There **seem** to be lots of examples of natural decision problems that live in $\mathbb{NP} - \mathbb{P}$: Satisfiability, Vertex Cover, Subset Sum, Hamiltonian Cycle, ...

All nice and good, but to really make a dent we need to

- fix a notion of reducibility suitable for $\mathbb{P}$ and $\mathbb{NP}$, and
- identify hard and complete problems wrto this reducibility.

It is natural to try polynomial time many-one reductions, or perhaps logarithmic space reductions (recall that polynomial time Turing reductions are a bit too coarse).

In either case, we have compatibility:

$$A \preceq B, B \in \mathbb{P} \quad \text{implies} \quad A \in \mathbb{P}$$

$$A \preceq B, B \in \mathbb{NP} \quad \text{implies} \quad A \in \mathbb{NP}$$

so this looks good.

As usual, constructing a hard set is easy, but the real challenge is to find an $\mathbb{NP}$-complete problem. Even better would be a *natural* $\mathbb{NP}$-complete problem.

Given an instance x, we are looking for the existence of some object w, a "witness" or"solution" for x, that is not too large, and that demonstrates that x is a yes-instance.

Usually we are in a situation where

witness obvious from the problem statement

size easily polynomial

Usually, both parts are fairly easy, even no-brainers. But **not** always.

It may be hard to find the right witness (primality testing), or it may be hard to show polynomial size (LOOP programs). This is admittedly rare, but it does happen.

You have all seen lots of problems in 251 that are clearly in $\mathbb{NP}$.

I will here focus on those where membership is far from obvious. They are part of $\mathbb{NP}$, just like their more popular peers.

In the general Traveling Salesman Problem, 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 $1 = \pi(1)$ to $\pi(2)$, then $\pi(3)$ and so on till $\pi(n)$, and ultimately return to city 1. The cost of this tour is

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

This is a classical minimization problem: we want a tour of minimal cost.

As usual, in the decision version there is an additional bound K and we ask whether there is a tour of cost at most K.

In the Metric (or Triangle) TSP the distances are required to be similar to actual geometric distances: they need to be symmetric and conform to the triangle inequality:

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

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

Thus, distances in the Euclidean TSP are also symmetric and obey the triangle inequality. But beware ...

Points in the Plane?

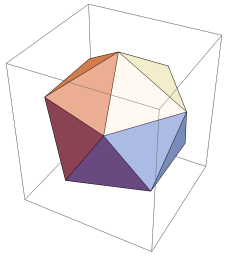
General TSP and Metric TSP are perfectly well-behaved finitary problems, just a square matrix of positive integers, and an integer bound. Clearly in $\mathbb{NP}$.

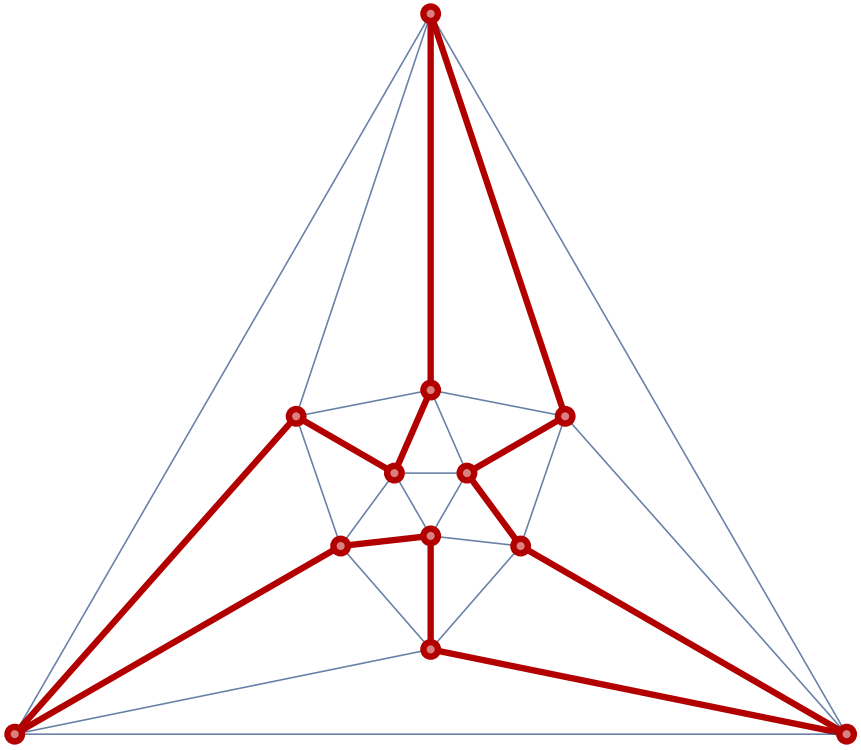
To make sense of the Euclidean version, we read "points in the plane" as points in $\mathbb{Q} \times \mathbb{Q}$. In fact, by scaling, we may assume the points are in $\mathbb{Z} \times \mathbb{Z}$.

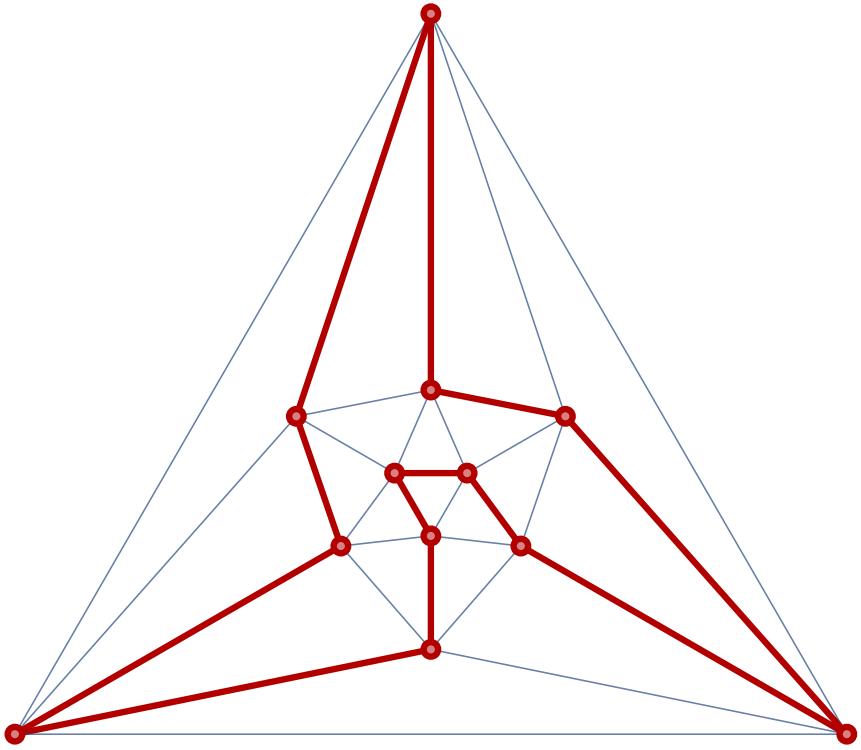
Much better, but there still is a problem: one has to compare expressions of the form

$$\sqrt{a_1} + \sqrt{a_2} + \ldots + \sqrt{a_k}$$

for integral a_i to compare the lengths of tours or parts thereof. The problem is, we need to do this in a polynomial amount of computation. Unfortunately, the roots are usually not rational numbers.





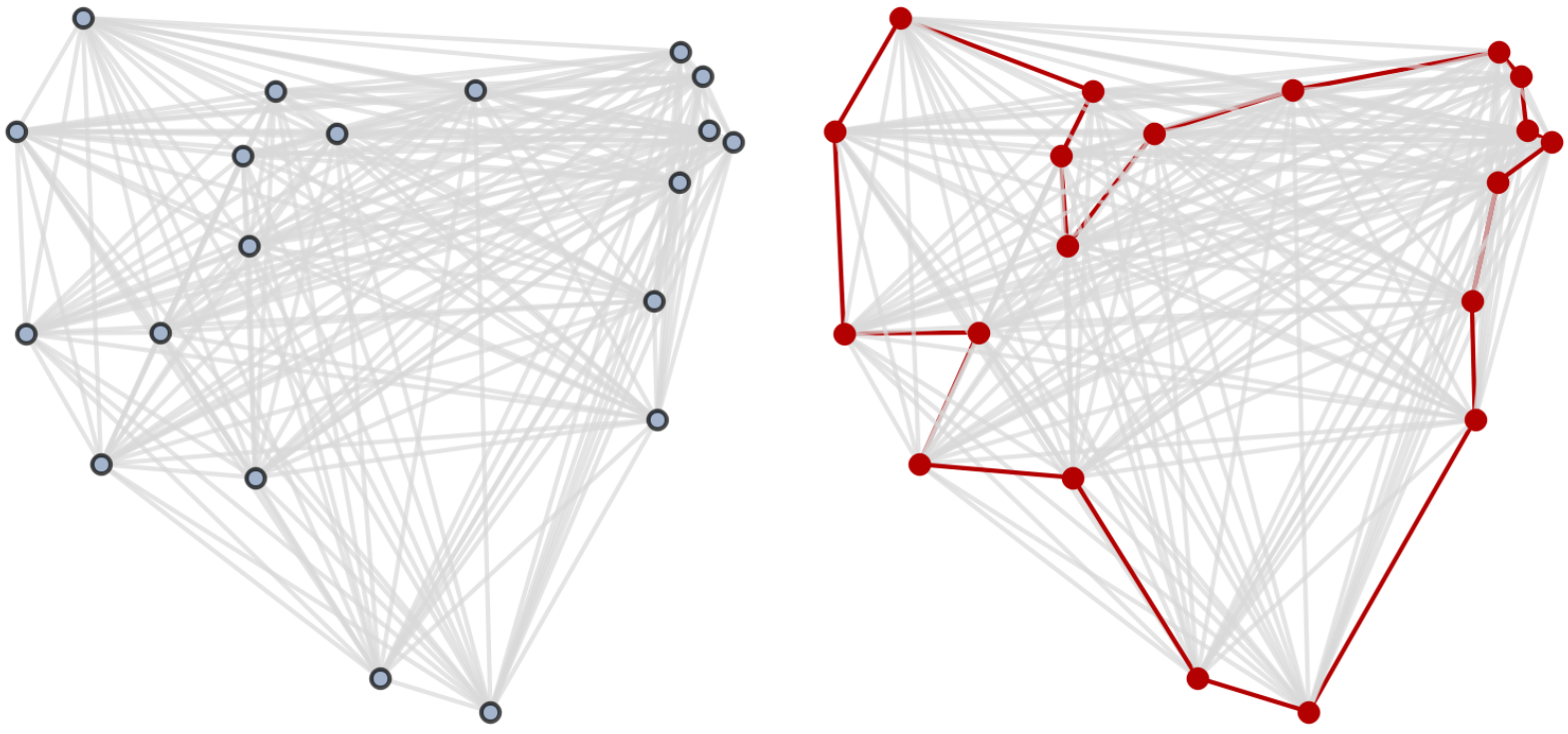


The last two icosahedron tours are both pretty good. To figure out which is better, one needs to determine the sign of

$$544000 + 5000\sqrt{11833} - 1000\sqrt{461401} - 1000\sqrt{462113} + \\ - 2000\sqrt{724645} + 1000\sqrt{1056890} + 1000\sqrt{1058285}$$

The numerical value is about 83103.7, so the first, more symmetric tour is better. But this requires an error analysis for the numerical algorithms used (the error here is guaranteed to be less than 1).

Bigger Example



To ensure membership in $\mathbb{NP}$ for Euclidean TSP, we need to limit precision to a polynomial number of digits. It is currently not known whether that is enough to deal with Euclidean TSP.

The standard workaround is to simply truncate, by brute-force. For example, one could use only the integer part of the actual value. Of course, that is a slightly different problem, a different metric.

At any rate, when it is claimed that Euclidean TSP is in $\mathbb{NP}$, it is always in reference to this modified version of the problem.

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

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

Note that finding a Hamiltonian cycle seems to be a little easier than tackling TSP: all we need is a cycle, there are no distances.

For suppose $G = \langle V, E \rangle$ is an undirected graph. Think of the vertices as cities, and define the following distances:

$$d(u, v) = \begin{cases} 1 & \text{if } uv \in E, \\ 2 & \text{otherwise.} \end{cases}$$

Then G has a Hamiltonian cycle iff there is a TSP tour of cost $n = |V|$.

Incidentally, this reduction produces a Metric TSP instance.

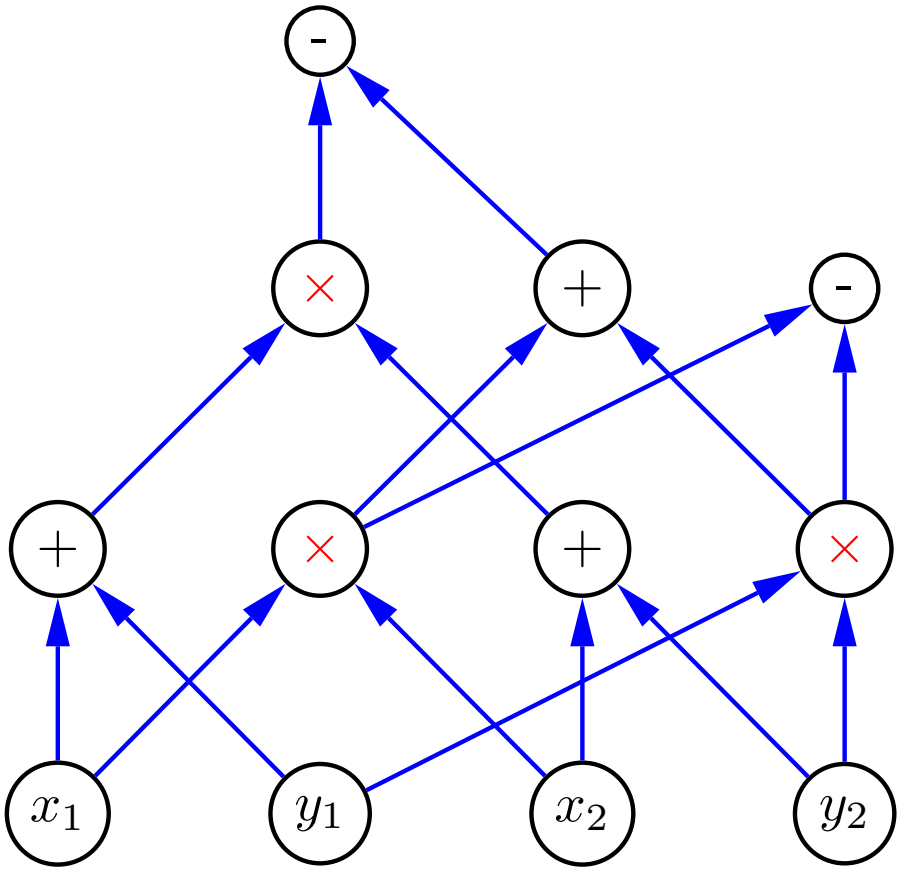
Here is a slightly more complicated example.

Consider a directed acyclic 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 predecessors: all y such that $yx \in E$. Hence we have to start at indegree 0 nodes.
- 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 goal is to minimize temporary storage when executing a straight-line program. For example, here is a circuit corresponding to a SLP for complex multiplication (given two pairs of real numbers):

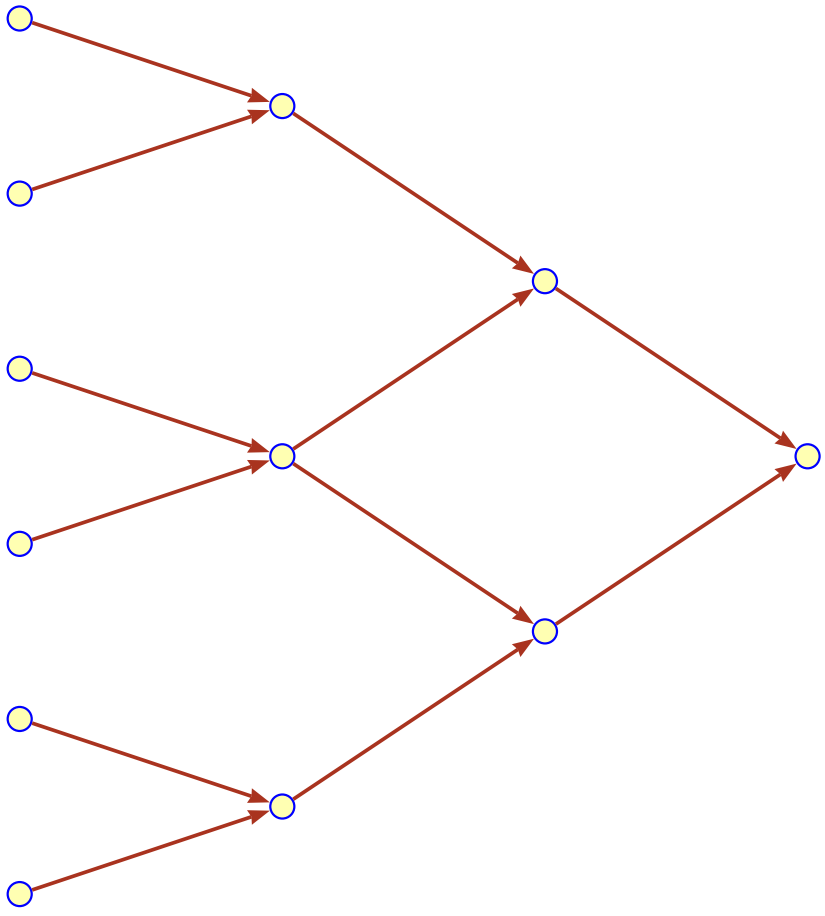


In general, the nodes of the digraph correspond to computational tasks, and execution of node/task x requires the results of all tasks y such that $yx \in E$.

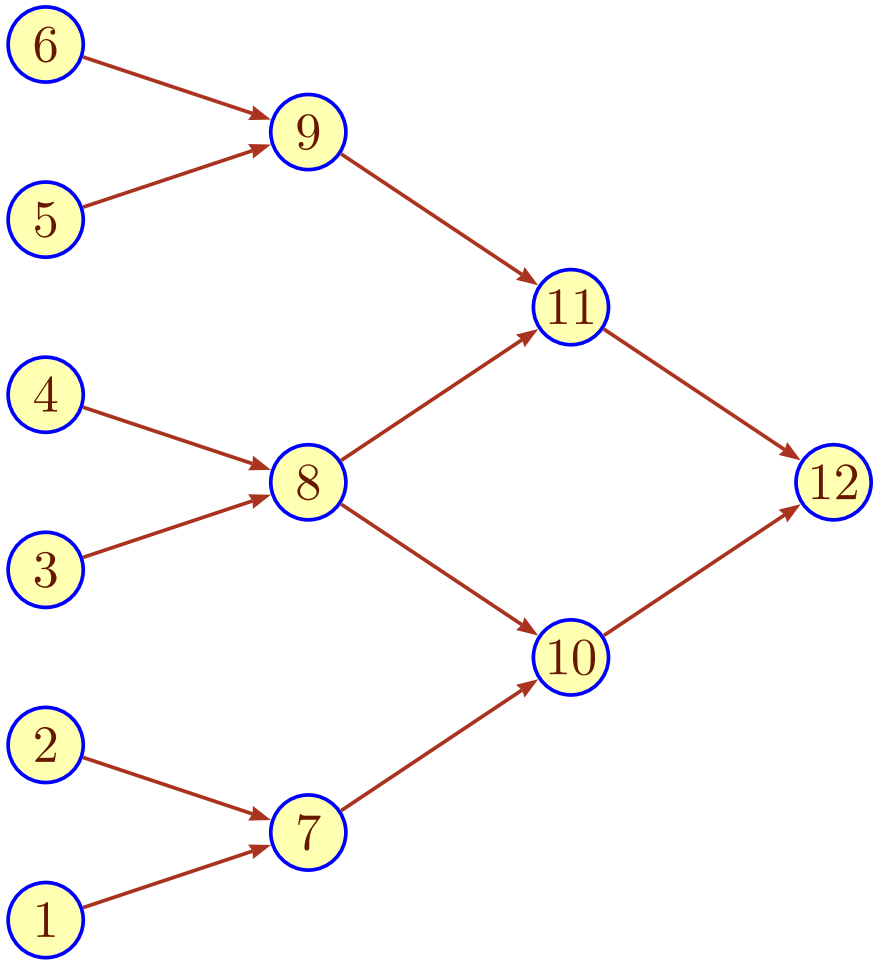
For simple arithmetic operations, the pebbles correspond to registers, so this is really about register allocation.

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.



Bad Solution



Given a permutation $\pi = x_1, \ldots, x_n$ 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

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.

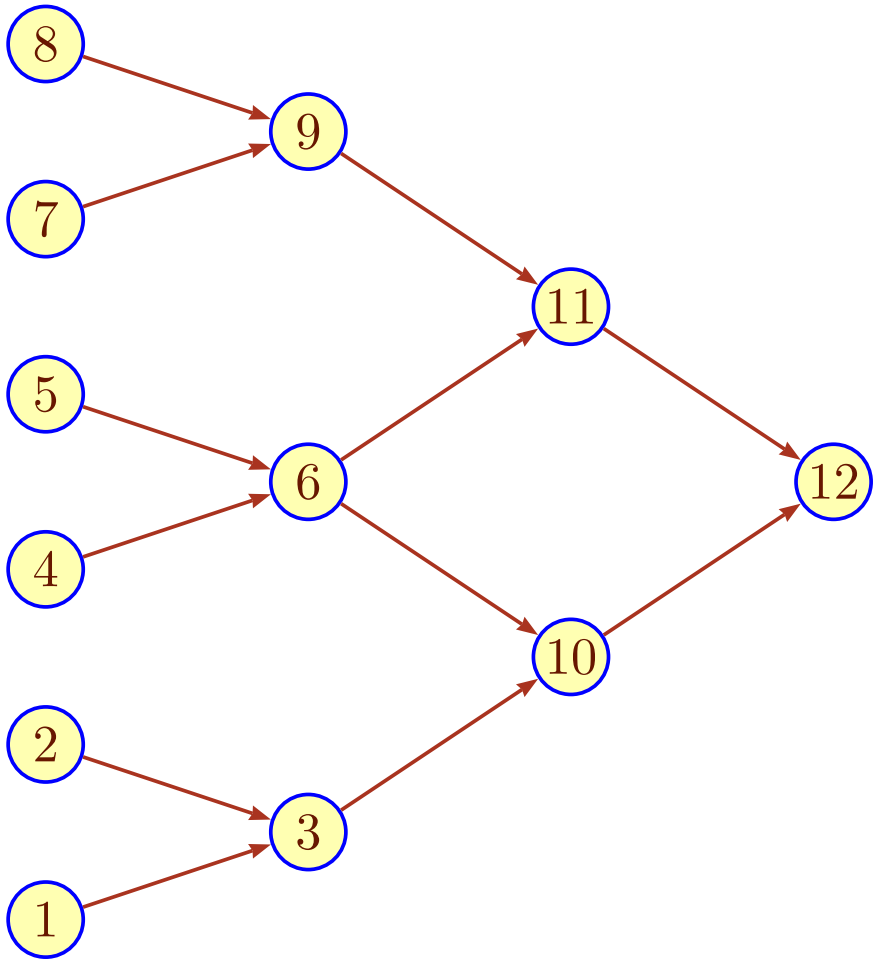
Bad Solution, Contd.

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	2	4	6	8	9	11	
			3	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.

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 $\mathbb{NP}$ 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 ...

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

This 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.

We now know that Primality is in $\mathbb{P}$, based on an absolutely brilliant high school algorithm that appears to be useless in practice. That result was foreshadowed a long time ago.

First, the complement of Primality, Composite, is obviously in $\mathbb{NP}$: guess the factors, and multiply them out.

Second, Pratt showed that Primality is actually in $\mathbb{NP}$.

Together this means Primality is in $\mathbb{NP} \cap$ co-$\mathbb{NP}$, and thus somewhat likely to slide down into $\mathbb{P}$. This is solely a matter of experience, not a hard technical result: natural problems in $\mathbb{NP} \cap$ co-$\mathbb{NP}$ tend to wind up in $\mathbb{P}$, possibly after a lot of work. Primality testing is now known to be in polynomial time.

The multiplicative subgroup of $\mathbb{Z}_n = \mathbb{Z}/n\mathbb{Z}$ for $n \geq 0$:

$$\mathbb{Z}_n^* = \{\, x \mid \gcd(x, n) = 1, 1 \leq x < n \,\},$$

Recall: Euler's totient function $\phi(n) = |\mathbb{Z}_n^*|$,

The order of $x \in \mathbb{Z}_n^*$, $\mathsf{ord}(x) = \min(k \geq 1 \mid x^k \equiv 1 \pmod{n})$,

g is a generator of $\mathbb{Z}_n^*$ iff $\mathsf{ord}(g) = \phi(n)$.

Proposition

$\phi(n) = \prod p_i^{e_i - 1}(p_i - 1)$ *where* $n = \prod p_i^{e_i}$.

Hence n *is prime iff* $\phi(n) = n - 1$ *iff* $\mathbb{Z}_n^*$ *has a generator of order* $n - 1$.

Just to be clear: the proposition says that primality testing is easy provided that factoring is easy—this is of no direct algorithmic use.

Theorem (Pratt 1975)

Primality is in $\mathbb{NP}$.

Proof.

procedure primeq(n)
  if $n \leq 42$
  then use brute force to test primality
  else
    guess a generator g of $\mathbb{Z}_n^*$
    guess the prime factorization of $n-1$, say $n-1 = \prod p_i^{e_i}$
    verify that $g^{n-1} \equiv 1 \pmod{n}$
    verify that for all $q = \frac{n-1}{p_i}$: $g^q \not\equiv 1 \pmod{n}$
    verify that all p_i are prime
  if everything checks
  then return Yes
  else return No

Verification of primality of the p_i is done recursively by making calls $\texttt{primeq}(p_i)$. The arithmetic for each call to `primeq` is clearly polynomial in $\Theta(\log n)$. The total number of recursive calls is $O(\log n)$ as all $p_i \leq n/2$. □

Example

The following tables show that 733 is prime.

$n = 733$
$n - 1 = 2^2 \cdot 3 \cdot 61,\ g = 6$
$6^{732} \equiv 1 \pmod{733}$
$6^{366} \equiv -1 \pmod{733}$
$6^{244} \equiv 425 \pmod{733}$
$6^{12} \equiv 299 \pmod{733}$

$n = 61$
$n - 1 = 2^2 \cdot 3 \cdot 5,\ g = 2$
$2^{60} \equiv 1 \pmod{61}$
$2^{30} \equiv -1 \pmod{61}$
$6^{20} \equiv 47 \pmod{61}$
$6^{12} \equiv 9 \pmod{61}$

Here is a simple toy language that nicely captures the notion of "easily computable" in the sense of CRT.

Again variables range over $\mathbb{N}$, and we have a single constant 0. The programs are described informally as follows:

reset	$x = 0$
increment	$x = x + 1$
assignments	$x = y$
sequential composition	$P; Q$
control	do $x : P$ od

Note that there are no arithmetic operations, no conditionals, no subroutines, nothing fancy. If you are familiar with primitive recursive functions: these are exactly the same as loop computable.

The semantics are clear except for the loop construct:

```
do x :  P  od
```

This is intended to mean: Execute P exactly n times where n is the value of x before the loop is entered.

In other words, if P changes the value of x the number of executions will still be the same. We could get the same effect by not allowing x to appear in P. It follows that all loop programs terminate, regardless of the input.

Here are some typical examples for the use of loops.

Addition and multiplication are not a primitive operation, but are easy to implement addition in a LOOP program. We indicate input and output variables in a comment line:

```
1 //  add : x, y --> z
2     z = x;
3     do y :
4           z = z+1;
5     od

1 //  mult : x, y -> z
2     z = 0;
3     do x :
4           do y :
5                 z = z+1;
6           od
7     od
```

How about the predecessor function? In some frameworks this is the first real challenge[†].

$$\mathsf{pred}(x) = x \dot{-} 1 = \begin{cases} 0 & \text{if } x = 0, \\ x - 1 & \text{otherwise.} \end{cases}$$

This requires a little trick, which is not totally obvious. We use an extra variable that lags behind.

```
1 // pred : x -> z
2     z = 0;
3     v = 0;
4     do x :
5           z = v;
6           v = v+1;
7     od
```

[†]Kleene figured out how to do this in the λ-calculus.

There is a natural way to measure the complexity of a loop program: the maximum nesting depth of all loops in the program.

Define LOOP_k to be the collection of all loop programs where the maximum nesting depth is k.

For example, addition is in LOOP_1, and multiplication is in LOOP_2.

As a consequence, any polynomial in $\mathbb{Z}[x_1, \ldots, x_n]$ can be calculated by a LOOP_2 program: represent every integer z by two natural numbers z_+ and z_- with the intent that $z = z_+ - z_-$.

Let's say that two loop programs P_1 and P_2 are inequivalent if there is some input $\boldsymbol{x}$ such that $P_1(\boldsymbol{x}) \neq P_2(\boldsymbol{x})$. Naturally one would like to understand the complexity of inequivalence testing for LOOP_k.

- LOOP_0 is trivially in $\mathbb{P}$. loop progs
- LOOP_2 is undecidable by Matiyasevic (Hilbert's 10th problem).
- LOOP_1 is not obvious.

Note the amazing and abrupt jump from polynomial time (and really totally trivial) to Σ_1-complete.

LOOP_1 must be somewhere in between, but it's not entirely clear where it will land.

We could guess a witness $\boldsymbol{x}$, and then check that $P_1(\boldsymbol{x}) \neq P_2(\boldsymbol{x})$.

More precisely, it is not hard to see that given $\boldsymbol{x}$ and a LOOP_1 program P, we can evaluate $P(\boldsymbol{x})$ in time polynomial in the size of P and $\boldsymbol{x}$.

So the real problem is: how long is the shortest witness $\boldsymbol{x}$?

Theorem (Tsichritzis 1970)

Inequivalence of LOOP_1 *programs is in* $\mathbb{NP}$.

$\mathbb{NP}$-Hardness is fairly easy to see, so we have an $\mathbb{NP}$-complete problem.

Proposition

Inequivalence for LOOP(0) *programs is decidable in polynomial time.*

Proof.

Given the program P, we can easily compute an index i and a constant $d \geq 0$ such that P computes

$$\boldsymbol{x} \mapsto d \qquad \text{or} \qquad \boldsymbol{x} \mapsto x_i + d$$

But then equivalence and hence inequivalence are trivial to check: index and constants have to be the same for both programs.

□

Exercise

Devise a fast algorithm to test Equivalence of LOOP(0) *programs.*

Lemma

Inequivalence for LOOP(2) *programs is undecidable.*

Proof. Given a multivariate integer polynomial $p(\boldsymbol{x})$ one can easily build a program P that computes

$$\overline{\mathsf{sign}}(p(\boldsymbol{x})) \in \{0, 1\}$$

P is naturally level 2 since all the arithmetic can be handled there.

Let Q be the trivial program that computes the constant 0 function. Then Inequivalence for P and Q comes down solving a Diophantine equation, which problem is undecidable by Matiyasevic's theorem.

□

That leaves Inequivalence for level 1 open: can we check if two rudimentary functions disagree on some input?

One might suspect that Inequivalence of rudimentary functions is indeed decidable since these functions are in some sense periodic or piecewise affine.

But the details bear some careful explanation: level 2 is not far away, and there Inequivalence is already undecidable.

As it turns out, Inequivalence for LOOP(1) program is decidable, but is already $\mathbb{NP}$-hard, so there is likely no fast algorithm for checking equivalence of such programs.

Definition

Define an equivalence relation $\equiv_{\beta,\mu}$ on $\mathbb{N}^n$ as follows: $\boldsymbol{x}$ and $\boldsymbol{y}$ are equivalent if

- $x_i < \beta \vee y_i < \beta$ implies $x_i = y_i$, and
- $x_i \geq \beta \wedge y_i \geq \beta$ implies $x_i = y_i \pmod{\mu}$.

Each equivalence class of $\equiv_{\beta,\mu}$ is either a singleton or infinite.

The number of equivalence classes is $(\beta + \mu)^n$: each component of the vector is either completely fixed (if it is less than β) or fixed modulo μ.

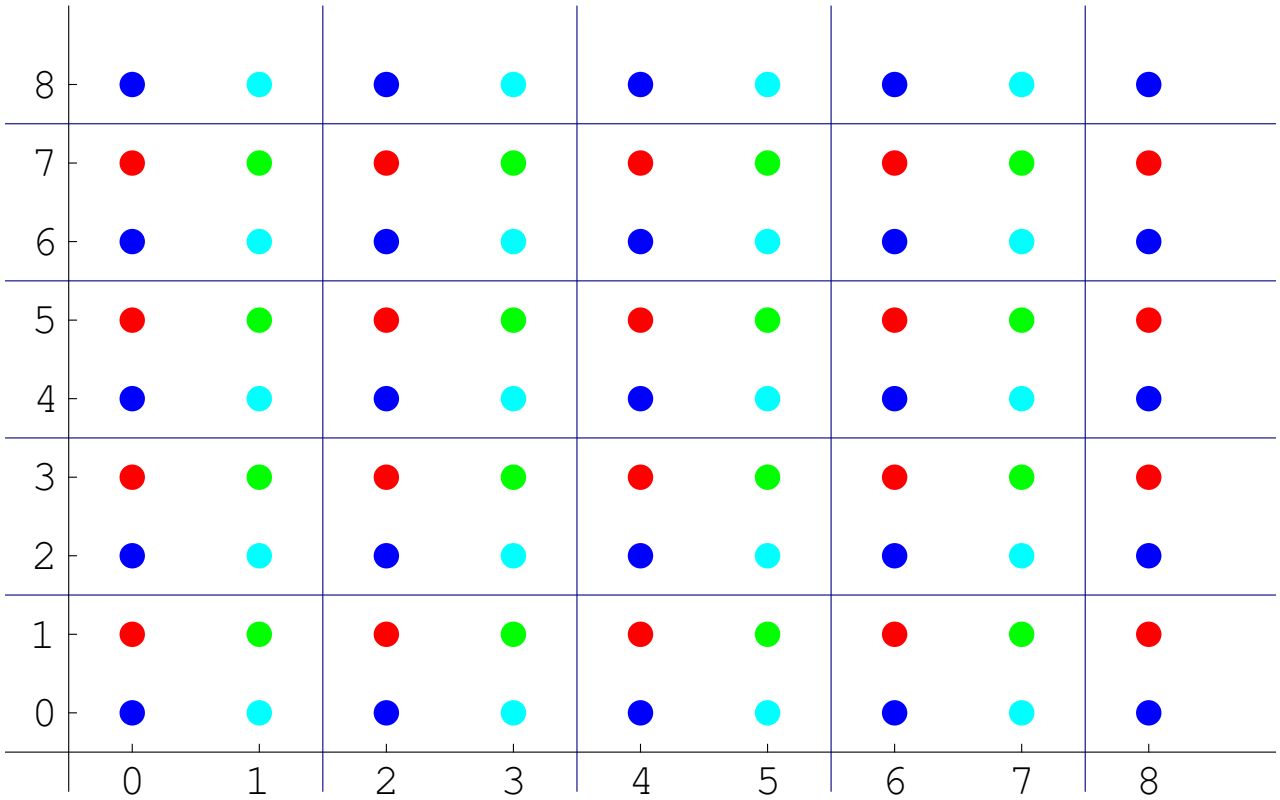
A simple case arises when $\beta = 0$: then we are simply subdividing $\mathbb{N}^n$ into hypercubes of size μ^n.

Example

Consider the function

$$f(x_1, x_2) = x_1 + x_1 \bmod 2 + x_2 \operatorname{div} 2 + 1.$$

For $\beta = 0$, $\mu = 2$ we get the following classes:



Example, contd.

$$f(x_1, x_2) = x_1 + x_1 \bmod 2 + x_2 \operatorname{div} 2 + 1$$

The corresponding four component functions for the equivalence classes are

$$x_1 + \tfrac{1}{2}x_2 + \tfrac{1}{2} \qquad x_1 + \tfrac{1}{2}x_2 + \tfrac{3}{2}$$

$$x_1 + \tfrac{1}{2}x_2 + 1 \qquad x_1 + \tfrac{1}{2}x_2 + 2$$

In essence, the mod terms affect the additive constant and the div terms produce the fractional coefficients.

Here is another way to describe LOOP(1) functions.

Definition

A class of functions is a clone if it closed under composition and projections $\boldsymbol{x} \mapsto x_i$.
A function is rudimentary if it is a member of the least clone containing constants 0, 1, addition, predecessor, division and remainder with fixed modulus, and an if-then function.

By an if-then function we mean the following:

$$W(x,y) = \begin{cases} y & \text{if } x > 0, \\ 0 & \text{otherwise.} \end{cases}$$

Theorem

For each rudimentary function $f : \mathbb{N}^n \to \mathbb{N}$ there are constants β and μ such the restriction of f to the equivalence classes of $\equiv_{\beta,\mu}$ is an affine function:

$$f(\boldsymbol{x}) = \sum_i c_i \cdot x_i + c.$$

Proof.

Use induction on the buildup of f. Here are the important cases.

$f_1 + f_2$	$\beta = \max(\beta_1, \beta_2)$	$\mu = \mu_1\mu_2$
$f_1 \dot{-} 1$	$\beta = \beta_1 + \mu_1$	$\mu = \mu_1$
$W(f_1, f_2)$	$\beta = \max(\beta_1, \beta_2) + \mu_2$	$\mu = \mu_1\mu_2$
$f_1 \operatorname{div} c$	$\beta = \beta_1$	$\mu = c\mu_1$
$f_1 \bmod c$	$\beta = \beta_1$	$\mu = c\mu_1$

□

Suppose $g(x)$ is affine on the classes of $\equiv_{0,\mu}$.

So there is a family of μ many affine functions G_i such that

$$g(x) = G_{x \bmod \mu}(x)$$

Set $f(x) = g(x) \bmod c$. Then

$$f(x) = G_{x \bmod \mu}(x) \bmod c = G_{x \bmod \mu}(x \bmod c) \bmod c$$

and we have to distinguish at most $c\mu$ classes for f.

A Basis Set

One can push things a bit further and show that if a rudimentary function f is piecewise affine with respect to $\equiv_{\beta,\mu}$ then f is completely determined by its values on the basis set

$$S = \{\, \boldsymbol{x} \mid x_i \leq \beta + 2\mu \,\}.$$

In other words, if g is another rudimentary function with parameters β and μ and we have

$$\forall\, \boldsymbol{x} \in S\, (f(\boldsymbol{x}) = g(\boldsymbol{x}))$$

then the two functions already agree everywhere.

Note that $\equiv_{\beta',\mu'}$ refines $\equiv_{\beta,\mu}$ whenever $\beta' \geq \beta$ and $\mu' = c\mu$.

Hence we can always choose the same parameters for any two functions.

Theorem

Let f_1 and f_2 be two rudimentary functions with common parameters β and μ. Then the two functions are equivalent iff they agree on

$$\{\, \boldsymbol{x} \mid x_i \leq \beta + 2\mu \,\}.$$

Theorem

Inequivalence for LOOP(1) *is* $\mathbb{NP}$*-complete*

Proof.

For membership, we can easily compute the common parameters β and μ from the programs.

We can then guess the witness in the test set

$$S = \{\, \boldsymbol{x} \mid x_i \leq \beta + 2\mu \,\}.$$

and verify that the two programs indeed differ on this input by running the corresponding computations.

Except for guessing the witness, all of this can be handled in in deterministic polynomial time. But the witnesses are short, so the whole procedure is in $\mathbb{NP}$.

For hardness we show how to reduce 3SAT to Inequivalence.

Suppose we have Boolean variables $x_1, x_2, \ldots, x_n$ and clauses $C_1, C_2, \ldots, C_m$.

Now consider a clause C_i, say, $C_i = x \vee \overline{y} \vee z$. We compute the truth value $c_i \in \mathbf{2}$ of C_i as follows:

```
z = 0;
if( x == 1 ) z = 1;
if( y == 0 ) z = 1;
if( z == 1 ) z = 1;
```

Lastly, we compute $\min(c_1, \ldots, c_m)$, the truth value of the whole formula.

The corresponding program is easily LOOP(1) (it operates solely on Boolean values and does not begin to exploit the possibilities of arithmetic) and inequivalent to $\mathbf{0}$ iff the formula is satisfiable.

□