$\mathbb{NP}$ and Completeness

Klaus Sutner

Carnegie Mellon University

20-Cook-Levin 2017/12/15 23:17

We have a whole collection of complexity classes based on

- time bounds,
- space bounds,
- determinism vs. nondeterminism

While there are some separation results (based on diagonalization) it is in general exceedingly difficult to distinguish between various low complexity classes.

The most famous case being $\mathbb{P}$ versus $\mathbb{NP}$.

In the absence of a separation result, a reasonable attempt is to consider a natural "difficulty order" on a class, and identify maximal elements.

This is strictly analogous to our old notion of reduction $\preceq$ in classical computability theory. Ideally we want a natural decision problem C that is $\mathbb{NP}$-complete:

- C is in $\mathbb{NP}$, and
- for all $A \in \mathbb{NP}$, $A \preceq C$.

Technically, we want $\preceq$ to be a preorder (reflexive and transitive).

Perhaps the most obvious attempt would be to use Turing reductions, but with the additional constraint that the Turing machine must have polynomial running time. This is called a polynomial time Turing reduction.

Notation: $A \leq_T^p B$.

Note that this actually makes sense for any kind of problem, not just decision problems. Also, it really captures nicely the idea that problem A is easier than B since we could use a fast algorithm for B to construct a fast algorithm for A.

Just to be clear: in a computation with oracle B we only charge for the steps taken by the ordinary part of the Turing machine, the solutions provided by the oracle do not count towards the time complexity.

Proposition

The decision version of Vertex Cover is polynomial time Turing reducible to the function version.

This is not hard:

- On input G and k hand over G to the oracle.
- Get back a minimal vertex cover C.
- Check whether $|C| \leq k$.

This is clearly linear time in the size of the input – recall that we charge nothing for the work of the oracle.

Proposition

The function version of Vertex Cover is polynomial time Turing reducible to the decision version.

This requires some work. Let n be the number of vertices in G, say, $V = \{v_1, \ldots, v_n\}$.

- First, do a binary search to find the size k_0 of a minimal cover, using the oracle for the decision problem.
- Then find the least vertex v such that $G - v$ has a cover of size $k - 1$. Place v into C and remove it from G.
- Recurse.

This is no longer linear but still polynomial time.

Exercise

Determine the exact running time of this oracle algorithm.

The last argument is fairly typical: many function and search problems can be rephrased as decision problems without losing much: they are polynomial time Turing reducible to the decision version.

Hence for difficult problems we can focus on the decision version and still get a good idea what the complexity of the more natural function/search version is.

BTW: I could not figure out how to reduce the function version of the Pebbling Problem to the decision version. Excellent extra credit project.

So let's focus on decision problems from now on.

Proposition

Polynomial time Turing reducibility is a preorder.

For transitivity, this works since polynomials are closed under substitution. Hence we can from equivalence classes as usual, the polynomial time Turing degrees. We won't pursue this idea here.

Proposition

$B \in \mathbb{P}$ *and* $A \leq_T^p B$ *implies* $A \in \mathbb{P}$.

Here we can simply replace the oracle by a real polynomial algorithm.

Our polynomial time Turing reductions are very natural, but a bit too powerful for $\mathbb{NP}$: it seems that co-$\mathbb{NP}$ is different from $\mathbb{NP}$, but these reductions do not distinguish between a set and its complement.

$$\overline{A} \leq_T^p A \qquad \text{and} \qquad A \leq_T^p \overline{A}$$

for any set A.

So $B \in \mathbb{NP}$ and $A \leq_T^p B$ is not known to imply $A \in \mathbb{NP}$, which is a bit awkward.

In fact, there is no reason to assume that $\mathbb{NP}$ is closed under complementation: it appears rather likely that $\mathbb{NP} \neq$ co-$\mathbb{NP}$.

As before in classical recursion theory, we can consider weaker reductions instead. In this case, many-one reducibility turns out to be the right notion.

Let A and B be two decision problems, with Yes-instances Y_A and Y_B, respectively.

Definition

A is polynomial time reducible to B if there is a polynomial time computable function f such that $x \in Y_A \iff f(x) \in Y_B$.

Notation: $A \leq_m^p B$.

Proposition

Polynomial time reducibility is a preorder.

Proof.

Reflexivity is trivial. For transitivity, consider a polynomial time reduction f from A to B and g from B to C.

Obviously $h(x) = g(f(x))$ is a reduction from A to C.

h is still polynomial time computable since polynomials are closed under substitution.

□

This may seem obvious, but note that it does not work for other classes (e.g., exponential time computable reductions).

Incidentally, it does work for linear time computable reductions.

Lemma

- *If B is in $\mathbb{P}$ and $A \leq_m^p B$ then A is also in $\mathbb{P}$.*
- *If B is in $\mathbb{NP}$ and $A \leq_m^p B$ then A is also in $\mathbb{NP}$.*

Proof.

(1) Replace the oracle B by a polynomial time algorithm.

(2) Combine the transducer for the reduction and the nondeterministic acceptor for B: produces a nondeterministic polynomial time acceptor for A.

□

Definition

B is $\mathbb{NP}$-hard if for all A in $\mathbb{NP}$: $A \leq_m^p B$.

B is $\mathbb{NP}$-complete if B is in $\mathbb{NP}$ and is also $\mathbb{NP}$-hard.

The existence of $\mathbb{NP}$-hard sets is trivial, but existence of $\mathbb{NP}$-complete sets is far from obvious. Also note

Proposition

If B in $\mathbb{NP}$-complete and B is in $\mathbb{P}$ then $\mathbb{P} = \mathbb{NP}$.

If indeed $\mathbb{P} \neq \mathbb{NP}$ as expected, then no $\mathbb{NP}$-complete problem can have a polynomial time algorithm. This is the weak lower bound result that we are after.

We can construct an enumeration $(M_e)_e$ of all polynomial time Turing machines: just run a clock and stop the machine after $n^e + e$ steps if it has not halted already.

But then there is a universal, deterministic machine $\mathcal{U}$ that simulates M_e with only a polynomial slowdown. More precisely, $\mathcal{U}$ on input $e\#x$ simulates M_e on x in running time $q(n^e + e)$ where q is some polynomial (even low degree).

Of course, $\mathcal{U}$ itself is not a polynomial time machine, the running time increases for different choices of e.

Likewise, there is a universal, nondeterministic machine $\mathcal{U}_{\mathsf{nd}}$ that simulates nondeterministic polynomial time machines N_e with only a polynomial slowdown in the same sense as above. Again, $\mathcal{U}_{\mathsf{nd}}$ itself is not polynomial time.

So, one has to be a bit careful where the polynomial time bounds should go.

How do we get our hands on an $\mathbb{NP}$-complete problem?

It is tempting to try to scale down the Halting set.

So we want to use the universal machine $\mathcal{U}_{\text{nd}}$ that can simulate machines in the enumeration (N_e) of nondeterministic, polynomial time Turing machines just mentioned. A first shot would be to define

$$K = \{\, e \# x \mid x \text{ accepted by } N_e \,\}$$

It is easy to see that K' is $\mathbb{NP}$-hard, but there is no reason why it should be in $\mathbb{NP}$; simulation of N_e is not a task that can be handled within a fixed polynomial time bound ($\mathcal{U}_{\text{nd}}$ itself is not polynomial time).

So a simple head-on universality argument fails here.

We can fix the problem by padding the input so that we can compensate for the running time of N_e.

$$K = \{\, e \# x \# 1^t \mid x \text{ accepted by } N_e \text{ in } t = |x|^e + |x| \text{ steps} \,\}$$

Proposition

K is in $\mathbb{NP}$.

Proof.

To see this note that the slowdown by $\mathcal{U}_{\text{nd}}$ is polynomial, say, the simulation takes $q(n^e + e)$ steps.

But then $\mathcal{U}_{\text{nd}}$ can test, in time polynomial in $|e\#x\#1^t| = |x| + t + c$, whether N_e indeed accepts x.

□

Proposition

K is $\mathbb{NP}$-hard.

Proof.

Consider $A = \mathcal{L}(N_e) \in \mathbb{NP}$ arbitrary. Then the function

$$x \mapsto e \# x \# 1^{|x|^e + |x|}$$

is polynomial time computable and shows that $A \leq_m^p K$.

□

Hence, K is indeed $\mathbb{NP}$-complete.

So we have the desired existence theorem.

Theorem

There is an $\mathbb{NP}$-complete language.

Alas, this result is perfectly useless when it comes to our list of $\mathbb{NP}$ problems: they bear no resemblance whatsoever to K.

We have a foothold in the world of $\mathbb{NP}$-completeness, but to show that one of these natural problems is $\mathbb{NP}$-complete we would have to find a reduction from K to, say, Pebbling or Vertex Cover.

Good luck on that.

- Reductions

2. Cook-Levin

- LOOP Programs

- Program Equivalence

Problem:	**Satisfiability**
Instance:	A Boolean formula φ.
Question:	Is φ satisfiable?

Problem:	**Tautology**
Instance:	A Boolean formula φ.
Question:	Is φ a tautology?

SAT is clearly in $\mathbb{NP}$, TAUT in co-$\mathbb{NP}$

Of course, there might be some other, clever, structural approach to satisfiability testing. That seems unlikely in light of the following result.

Theorem (Cook-Levin 1971/1973)

The Satisfiability Problem is $\mathbb{NP}$-complete.

Membership in $\mathbb{NP}$ is easy using the standard guess-and-verify approach.

But hardness takes work: we have to express the computation of a Turing machine as a Boolean formula.

Note that the following proof should be read just once. Then throw it away and reconstruct your own proof.

Let A be an arbitrary set (of yes-instances) in $\mathbb{NP}$.

There is a deterministic polynomial time Turing machine M such that M accepts (x, w) iff $x \in A$ where w is some witness of length polynomial in $n = |x|$.

The idea is to construct a (rather large) Boolean formula Φ_x such that

$$\Phi_x \text{ is satisfiable} \iff M \text{ accepts } (x, w) \text{ for some } w.$$

While the formula is fairly long, it has a clear structure and it can easily be constructed in time polynomial in n.

It has lots of variables that express information about states, head position and tape inscriptions: the satisfying truth assignment translates directly into an accepting computation of M.

First off, let $N = p(n)$ be the running time of the machine.

If we have a list of Boolean variables

$$X_0, X_1, \ldots, X_N$$

and a truth assignment σ we can think of $\sigma(X_t)$ as the value of variable X at time t.

We can use logic to pin down the value of X_{t+1} in terms of X_t (and other variables).

$$\bigwedge_{t<N} X_{t+1} \iff \varphi(X_t, \ldots)$$

If we need to code a number r in a certain range, say $1 \leq r \leq m$, we can simply use variables

$$X(1), X(2), \ldots, X(m)$$

plus a stipulation that exactly one of them is true under σ:

$$\mathsf{EO}_m\big(X(1), X(2), \ldots, X(m)\big)$$

Here

$$\mathsf{EO}_k(x_1, \ldots, x_k) = (x_1 \vee x_2 \ldots \vee x_k) \wedge \bigwedge_{1 \leq i < j \leq k} \neg(x_i \wedge x_j).$$

Note that the size of EO_k is $O(k^2)$.

Combining these two ideas we can set up polynomially many Boolean variables

states	$S_t(p)$, $0 \leq t \leq N$, $1 \leq p \leq m$,
head position	$H_t(i)$, $0 \leq t, i \leq N$,
tape inscription	$C_t(i)$, $0 \leq t, i \leq N$.

that express, for each time $0 \leq t \leq N$, which state the machine is in, where the head is, and what's on the tape.

For simplicity, we assume here that the only tape symbols are 0 and 1, it is not hard to deal with the general situation.

We then have to express the constraint that the variables change from time t to time $t+1$ only in accordance with the transition function of the Turing machine.

For example

$$\bigwedge_{t<N} H_t(i) \Rightarrow \mathsf{EO}_3\big(H_{t+1}(i-1), H_{t+1}(i), H_{t+1}(i+1)\big)$$

$$\bigwedge_{t<N} S_t(p) \wedge H_t(i) \wedge \neg C_t(i) \Rightarrow S_{t+1}(q_0)$$

$$\bigwedge_{t<N} S_t(p) \wedge H_t(i) \wedge C_t(i) \Rightarrow S_{t+1}(q_1)$$

And so on and on.

Initially the input is on the first part of the tape

$$H_0(0) \wedge S_0(q_0) \wedge C_0(1) = x_1 \wedge C_0(2) = x_2 \wedge \ldots \wedge C_0(n) = x_n$$

and at the end we accept:

$$S_N(q_Y)$$

Note that $C_0(n+1), \ldots, C_0(N)$ is not fixed and can be set arbitrarily by σ.

It is not too hard to see that the whole formula Φ_x in the end has size polynomial in n.

Now suppose Φ_x is satisfied by truth assignment σ.

Then M accepts the original tape inscription (x, w) where x is the real input and w is the bits chosen freely by σ.

Since Φ_x forces the values of all the variables to correspond to a computation of M on input (x, w) we have the desired witness and x must be a Yes-instance.

Conversely, every witness plus corresponding accepting computation can be translated into a satisfying truth assignment σ.

That's it. □

The hardness of Satisfiability holds up even when the formulae in question are rather restricted.

Definition

A literal is a variable or negated variable. A formula is in conjunctive normal form (CNF) if it is a conjunction of disjunctions of literals: $\Phi_1 \wedge \Phi_2 \wedge \ldots \wedge \Phi_n$ where $\Phi_i = z_{i,1} \vee z_{i,2} \vee \ldots \vee z_{i,k(i)}$, $z_{i,j}$ a literal.

It is in k-CNF if $k(i) = 3$ for all i.

Theorem

The Satisfiability Problem is $\mathbb{NP}$-complete for formulae in 3-CNF.

The last result can be used to establish the $\mathbb{NP}$-hardness of many problems such as Vertex Cover.

It now suffices to find a polynomial time computable function

$$f : \text{3-CNF} \longrightarrow \text{Graphs} \times \text{Integers}$$

such that φ in 3-CNF is satisfiable iff for $f(\varphi) = (G, k)$ the graph G has a vertex cover of size k.

This is much, much easier than having to deal with general formulae.

Satisfiability is a tremendously important practical problem, but if this were the only relevant $\mathbb{NP}$-complete problem the whole notion would still be somewhat academic.

But as Dick Karp realized after reading Cook's paper, there are dozens (actually: thousands) of combinatorial problems that all turn out to be $\mathbb{NP}$-complete. So none of them will admit a polynomial time solution unless $\mathbb{P} = \mathbb{NP}$.

The proof method is interesting: some problems are proven hard by direct reduction from SAT, then these are used to show other problems are hard, and so on . . . By transitivity one could, in principle, produce a direct reduction from SAT, but in reality these direct reductions are often very hard to find.

Theorem

Vertex Cover is $\mathbb{NP}$-complete.

Proof.

Suppose we have a 3-CNF formula $\Phi = \Phi_1 \wedge \Phi_2 \wedge \ldots \wedge \Phi_m$ where $\Phi_i = z_{i,1} \vee z_{i,2} \vee z_{i,3}$.

The Boolean variables are $x_1, \ldots, x_n$.

We start with a graph G' on $2n + 3m$ vertices.

- Vertices: x_i, $\overline{x}_i$ for $i = 1, \ldots, n$ and $u_{i,1}$, $u_{i,2}$, $u_{i,3}$ for $i = 1, \ldots, m$.
- Edges: one edge between x_i and $\overline{x}_i$, and three edges that turn $u_{i,1}$, $u_{i,2}$, $u_{i,3}$ into a triangle.

It is easy to see that every vertex cover of G' must have at least $n + 2m$ vertices (one for each x-edge, and two for each triangle).

And such covers exist, lots of them: you can pick at random one of x_i or $\overline{x}_i$, and exactly two of $u_{i,1}$, $u_{i,2}$, $u_{i,3}$.

So far we have only used n and m, but not the formula itself.

Let G be the graph obtained by adding $3m$ more edges to G':

connect $u_{i,j}$ to x_s if $z_{i,j} = x_s$ and connect $u_{i,j}$ to $\overline{x}_s$ if $z_{i,j} = \neg x_s$.

Lastly, set the bound to $k = n + 2m$.

Claim

G has a cover of size $k = n + 2m$ iff the formula is satisfiable.

To see this, note that any cover C defines an assignment σ:

$$\sigma(x_i) = \begin{cases} 1 & \text{if } x_i \in C, \\ 0 & \text{otherwise.} \end{cases}$$

Then σ satisfies the formula.

Conversely, every satisfying assignment translates into a cover.

□

For this construction to work we need two crucial ingredients:

- The graph G and the bound k can be computed from Φ in polynomial time.
- G has a vertex cover of size k if, and only if, Φ is satisfiable.

Many other completeness proofs look very similar: it is trivial to see that the problem is in $\mathbb{NP}$, and it requires work (sometimes a lot of it) to produce hardness.

- Reductions
- Cook-Levin

3. LOOP Programs

- Program Equivalence

We have defined primitive recursive functions in an algebraic manner; here is the analogous programming language.

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

initialize	$x = 0$
assignments	$x = y$
increment	$x{+}{+}$
sequential composition	$P; Q$
control	do $x : P$ od

Note that there are no arithmetic operations, no conditionals, no subroutines, ...

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.

Note that this would not be true for while-loops. As we will see, while-loops are significantly more powerful than our do-loops.

Here are some typical examples for the use of loops.

Addition is easy to implement addition in a LOOP program:

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

Here is multiplication:

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

How about the predecessor function? In some frameworks (like the λ-calculus) 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++;
7       od
```

The sign function $\text{sign}(x) = \min(x, 1)$ can also be implemented by abusing the loop construct as a Boolean test.

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

Now we can build conditionals.

```
1  // if( x > 0 ) P;
2       z = sign(x);
3       do z :
4            P;
5       od
```

Theorem

The loop-computable functions are precisely the primitive recursive functions.

Proof.

The only difficult part in the argument is to show that

- loop-computable functions are closed under primitive recursion, and
- application of a loop operation preserves primitive recursiveness.

For simplicity assume that there is only one parameter y. Suppose $P : y \to z$ and $Q : x, u, y \to z$ are two loop programs computing $g : \mathbb{N} \to \mathbb{N}$ and $h : \mathbb{N}^3 \to \mathbb{N}$.

We show that $f = \mathsf{Prec}[g, h] : \mathbb{N}^2 \to \mathbb{N}$ is loop-computable by constructing a loop program for f directly.

Here is a program `Pf` for f. Variable s is new.

```
// Pf:   x, y --> z
    s = x;
    x = 0;
    P;              // y --> z
    do s:
        u = z;
        Q;          // x,u,y --> z
        x++;
    od
```

It is a good exercise to determine the semantics $[\![\texttt{Pf}]\!]$ in detail.

For the opposite direction consider a loop program P with variables $x_1, x_2, \ldots, x_k$ and semantics $f = [\![P]\!] : \mathbb{N}^k \to \mathbb{N}^k$.

We may assume by induction that $f = (f_1, \ldots, f_k)$ is primitive recursive in the sense that each of the f_i is so p.r. Now consider program `Q`:

```
do x: P od
```

For simplicity, assume $x = x_1$. Define the vector valued function

$$F(0, \boldsymbol{x}) = \boldsymbol{x}$$
$$F(n+1, \boldsymbol{x}) = [\![P]\!](F(n, \boldsymbol{x}))$$

Using sequence numbers we can show that all the component functions of F are primitive recursive. But then $[\![Q]\!](\boldsymbol{x}) = F(x_1, \boldsymbol{x})$.

□

There is a natural way to distinguish between loop programs of different complexity: we can count the nesting depth of the loops in the program.

Definition

Let Loop(k) be the class of loop programs that have nesting depth at most k.

Level 0 is not very interesting: it is easy to see that any scalar function that is Loop(0)-computable is an affine function of the form

$$f(\boldsymbol{x}) = c \cdot x_i + d$$

where c and d are both constant, $c \in \{0, 1\}$.

Exercise

Give a detailed proof of the last claim.

But Loop(1) turns out to be somewhat more complicated.

All the examples of Loop-programs from above are in fact Loop(1), multiplication being the one notable exception.

It is easy to modify the argument for conditionals above to show that if-then applied to Loop(1)-computable functions produces another Loop(1)-computable function.

Here are two more critical examples of Loop(1)-computable functions.

The following program computes the remainder for modulus 2.

```
1  // mod2 : x -> u
2       u = 0;
3       v = 1;
4       do x :
5            t = u;     // swap u and v
6            u = v;
7            v = t;
8       od
```

Exercise

Show that $x \bmod m$ *is* Loop(1)*-computable for every fixed modulus* m*.*

The following program computes the integer quotient $x/2$.

```
1  // div2 : x -> u
2       u = 0;
3       v = 0;
4       do x :
5            u++;
6            t = u;     // swap u and v
7            u = v;
8            v = t;
9       od
```

Exercise

Show that $x \operatorname{div} m$ *is* Loop(1)*-computable for ever fixed modulus* m.

As it turns out, there is a well-known class of number-theoretic functions $\mathbb{N}^n \to \mathbb{N}$ that looks very similar to Loop 1 programs.

Definition

A function is (Kalmar) rudimentary if it belongs to the clone generated from 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:

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

This is similar to classical the question-mark operator in the C programming language (the third argument is fixed to 0 in this case): `x ? y : 0`.

Theorem

A function is Loop(1)*-computable if, and only if, it is rudimentary.*

Proof.

We have just verified that rudimentary programs are Loop(1)-computable.

The other direction is more difficult: every Loop(1) program is already rudimentary. For the proof, consider a single loop `Q`:

```
do x: P od
```

Suppose the variables are $\boldsymbol{x}$ where $x = x_1$, as before. The loop body `P` is Loop(0), so the effect of executing it `P` is an affine function on each of the variables:

$$x_i' = c_i \cdot x_{p(i)} + d_i$$

where $c_i \in \{0, 1\}$ and $d_i \geq 0$ constant.

Here $p : [n] \to [n]$ is an arbitrary dependency map that indicates how values are passed from one variable to another in an assignment `xi = xj;`

To deal with assignments of the form `xi = 0;` it is convenient to introduce a phantom variable x_0 whose value is fixed at 0. Then we can rewrite the new values as

$$x'_i = x_{p(i)} + d_i$$

Now consider the dependency graph $G = \langle V, E \rangle$ where

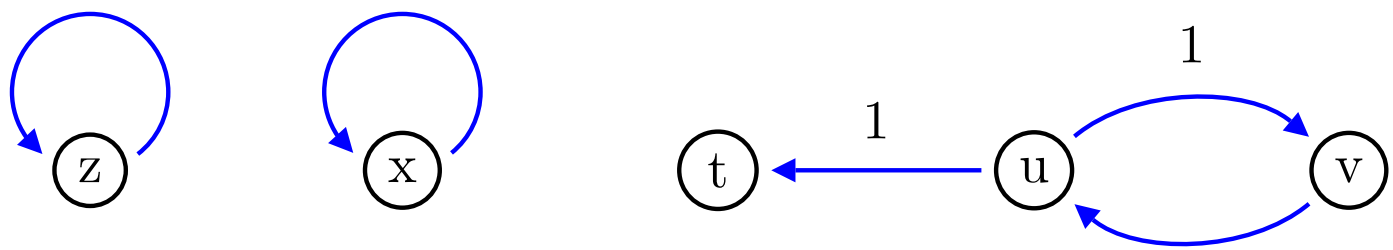
- $V = \{x_0, x_1, \ldots, x_n\}$
- $E = \{ (x_j, x_i) \mid j = p(i) \}$

Thus, there is a path in G from x_j to x_i if the value of x_j propagates to x_i, provided the loop is executed sufficiently often.

Variables are $\{z, x, u, v, t\}$ where z is clamped at 0.

```
1   u = z; v = z;
2   do x :
3      u++;
4      t = u; u = v; v = t;
5   od
```

The dependency graph for the loop body looks like so:



The edge labels indicate the increment factor d_i and we pretend that there are instructions $z = z$ and $x = x$.

Note that the indegree of every node in G is at most 1. Hence the strongly connected components of G are just cycles, and all these components are isolated from each other.

But then the values of the registers on a cycle at time t (after t executions of the loop) are of the form

$$x = \begin{cases} a_0 & \text{if } t = 0, \\ a \cdot (t \operatorname{div} l) + \sum_{i < t \bmod l} a_i & \text{otherwise.} \end{cases}$$

where l is the length of the cycle, and the a_i are the labels, a being the sum of all these labels.

Hence the final value of a variable after execution of `P` is a rudimentary function of the initial values. Since rudimentary functions are closed under composition, the whole Loop(1) program can only compute a rudimentary function.

□

- Reductions
- Cook-Levin
- LOOP Programs

4. Program Equivalence

We define equivalence of programs in terms of the function computed by the respective programs, a purely extensional characterization.

Definition

Two programs P and Q are equivalent if they compute the same function $\mathbb{N}^n \to \mathbb{N}$.

Note that this is a weaker notion than having the same semantics, $[\![P]\!] = [\![Q]\!]$. For example, there is no reason why equivalent programs should use the same internal variables.

More importantly, the two programs can be based on two different ways of computing the function in question. Clearly, this type of equivalence is very hard to detect.

For each class $\mathcal{C}$ of programs such as rudimentary, elementary, Loop(k), primitive recursive we now have a decision problem:

Problem:	**Program Equivalence for $\mathcal{C}$**
Instance:	Two programs P and Q in class $\mathcal{C}$.
Question:	Are P and Q equivalent?

This is the problem one would like to solve when one verifies the correctness of program transformations: Q is obtained from P by applying a transformation, and we want to make sure that the meaning of P has not changed.

Alas, for technical reasons it is often better to consider Inequivalence: Is there some input on which the two programs yield different output?

One reason for this twist is that Inequivalence of programs is semi-decidable, even if we have no constraints on the class $\mathcal{C}$ (actually, we have to insist on total functions rather than partial ones as in the case of arbitrary computable functions): For Inequivalence one can conduct a brute-force search over all possible inputs to find an $\boldsymbol{x}$ such that

$$P(\boldsymbol{x}) \neq Q(\boldsymbol{x}).$$

This may appear rather nonsensical, but in lower complexity classes the distinction between Equivalence and Inequivalence becomes quite important. Of course, as far as decidability is concerned, there is no difference between Equivalence and Inequivalence.

Proposition

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

Proof.

Given the program P, we can easily compute an index i and constants $c \in \{0, 1\}$, $d \geq 0$ such that P computes

$$\boldsymbol{x} \mapsto c \cdot x_i + d$$

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

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

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.

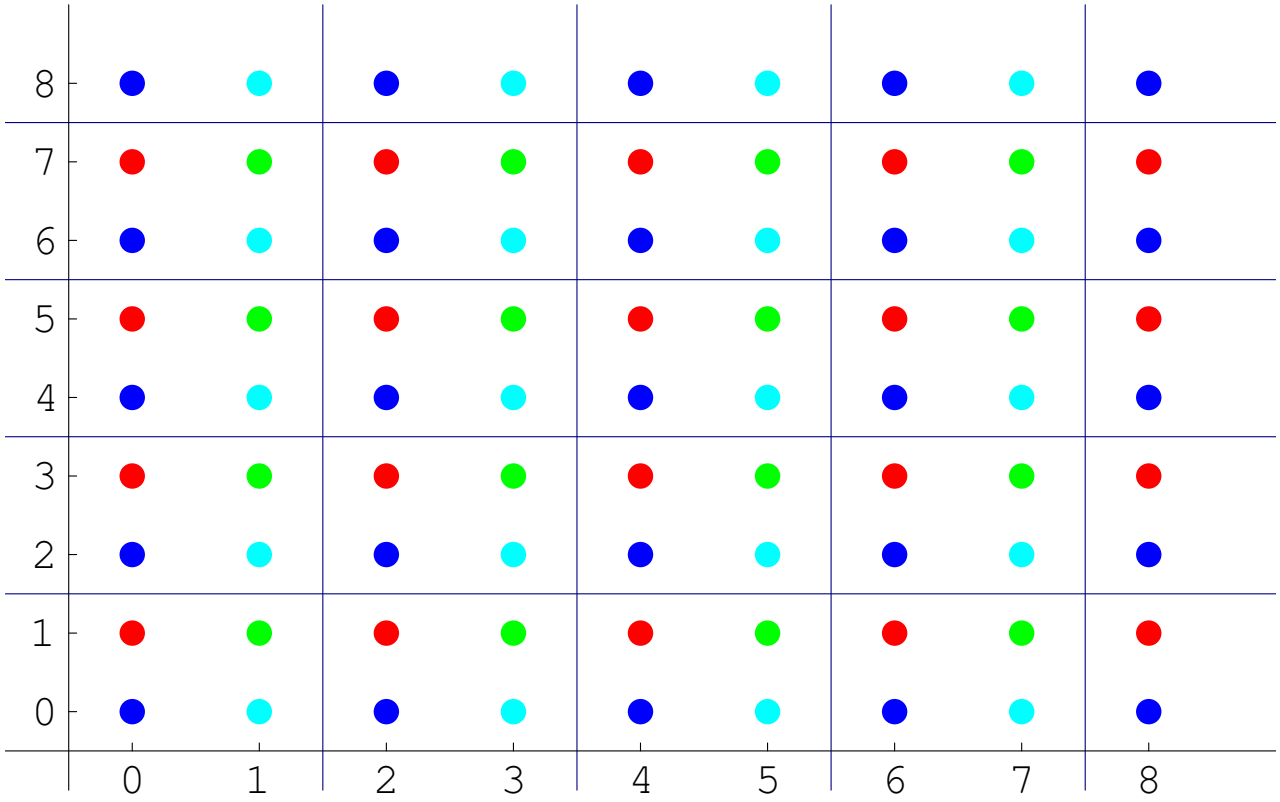
The number of equivalence classes is $(\beta + \mu)^n$: each component of the vector is either completely fixed (if it is less than β) or fixed modulo μ.

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

Consider the function

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

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



$$f(x, y) = x + x \bmod 2 + y \operatorname{div} 2 + 1$$

The corresponding four component functions for the equivalence classes are

$$x + \tfrac{1}{2}y + \tfrac{1}{2} \qquad x + \tfrac{1}{2}y + \tfrac{3}{2}$$

$$x + \tfrac{1}{2}y + 1 \qquad x + \tfrac{1}{2}y + 2$$

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

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.

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.

□