Decidability of Languages That Do Not Ask Questions about Turing Machines

Chapter 22

Undecidable Languages That Do Not Ask Questions About TMs

- Diophantine Equations, Hilbert’s 10th Problem
- Post Correspondence Problem
- Tiling problems
- Context-free languages

Hilbert's 10th Problem

A ***Diophantine system*** is a system of ***Diophantine equations*** such as:

$$4x^3 + 7xy + 2z^2 - 23x^4z = 0$$

Hilbert's 10th problem: given a Diophantine system, does it have an integer solution?

Tenth = {<*w*> : *w* is a Diophantine system with integer solution}.

Tenth is not in D?

Proved in 1970 by Yuri Matiyasevich.

Restricted Diophantine Problems

Suppose all exponents are 1:

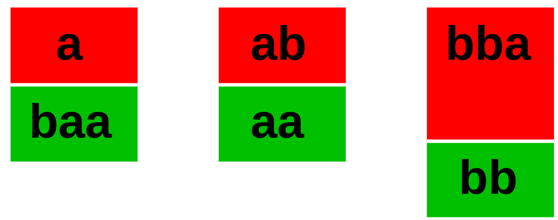
> A farmer buys 100 animals for $100.00. The animals include at least one cow, one pig, and one chicken, but no other kind. If a cow costs $10.00, a pig costs $3.00, and a chicken costs $0.50, how many of each did he buy?

- Diophantine problems of degree 1 and Diophantine problems of a single variable of the form $ax^k = c$ are efficiently solvable.
- The quadratic Diophantine problem is NP-complete.
- The general Diophantine problem is undecidable, so not even an inefficient algorithm for it exists.

Post Correspondence Problem

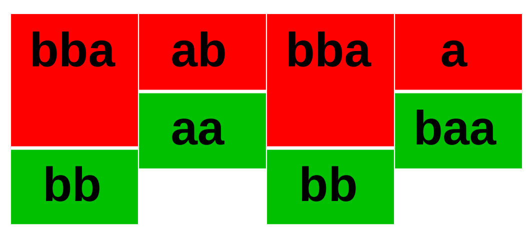
Emil Post (1940)

Consider the following blocks:



Assuming an unlimited supply of each block, does there exist a sequence of such blocks such that the strings on top and bottom are the same?

Solution:



Post Correspondence Problem

Consider two equal length, finite lists, X and Y, of strings over some alphabet Σ:

$$X = x_1, x_2, x_3, \ldots, x_n$$
$$Y = y_1, y_2, y_3, \ldots, y_n$$

Does there exist a finite sequence of integers *i1, i2,…, ik* such that

$$x_{i1}x_{i2}\ldots x_{ik} = y_{i1}y_{i2}\ldots y_{ik} \quad ?$$

PCP Example

b	abb	aba	baaa
aab	b	a	baba

Solution:

aba	baaa	b
a	baba	aab

PCP Example

bb	ab	aab
abb	a	bba

No solution!

PCP Example

bbab	abba	b
b	bb	bba

Shortest solution has length 252!

The Language PCP

$\langle P \rangle = (x_1, x_2, x_3, \ldots, x_n)(y_1, y_2, y_3, \ldots, y_n)$

The problem of determining whether a particular instance *P* of the Post Correspondence Problem has a solution can be recast as the problem of deciding the language:

$\text{PCP} = \{\langle P \rangle : P \text{ has a solution}\}$

The language PCP is in SD \ D.

PCP is in SD

Theorem. PCP is in SD.

Proof: Build a Turing Machine that tries all possible solutions of length 1, then all possible solutions of length 2, and so on.

PCP is not in D

Theorem. PCP is not in D.

Proof: Two reductions:

- $L_a \leq MPCP$
- $MPCP \leq PCP$

$L_a = \{\langle G, w\rangle : G$ unrestricted grammar, $w \in L(G)\}$;

L_a is not in D

MPCP (modified PCP): defined in the same way except that any solution must start with the first block:

$$x_1 x_{i2} \ldots x_{ik} = y_1 y_{i2} \ldots y_{ik}$$

PCP is not in D

$L_a \leq MPCP$

Given an instance <*G,w*> of L_a, we construct an instance <*P*> of MPCP able to simulate derivations in $G = (V, \Sigma, R, S)$.

Using % and & new symbols, not in *V*, a derivation

$$S \Rightarrow x_1 \Rightarrow x_2 \Rightarrow \ldots \Rightarrow w$$

is simulated as

$$\%S \Rightarrow x_1 \Rightarrow x_2 \Rightarrow \ldots \Rightarrow w\&$$

PCP is not in D

<P>:

start: $\left[\frac{\%S\Rightarrow}{\%}\right]$ copy the rest: $\left[\frac{c}{c}\right]$, for $c \in V$

end: $\left[\frac{\&}{\Rightarrow w\&}\right]$ $\left[\frac{\Rightarrow}{\Rightarrow}\right]$

derivate: $\left[\frac{\beta}{\alpha}\right]$, for each rule $\alpha \rightarrow \beta$ in R

It can be formally proved by induction on the length of the derivation that G can derive w iff P has a solution.

The idea is that any solution of P must start with the first block (it is an MPCP) and then the only way to continue is by the bottom string to catch up to the top by using blocks corresponding to rules of G.

PCP is not in D

Example:

G = ({*S*,*A*,*B*,*a*,*b*,*c*}, {*a*,*b*,*c*}, *R*, *S*), *w* = *ac*

R: S → *Abc*

S → *ABSc*

AB → *BA*

Bc → *bc*

BA → *a*

A → *a*

P:

%S⇒	S	A	B	a	b	c	⇒
%	S	A	B	a	b	c	⇒

&	*ABc*	*ABSc*	*BA*	*bc*	*a*	*a*
⇒ac&	S	S	AB	Bc	BA	A

PCP is not in D

For the derivation of *w*:

$$S \Rightarrow ABc \Rightarrow BAc \Rightarrow ac$$

P simulates:

$$\%S \Rightarrow ABc \Rightarrow BAc \Rightarrow ac\&$$

as follows:

%S⇒	***ABc***	**⇒**	***BA***	**c**	**⇒**	***a***	**c**	**&**
%	**S**	**⇒**	**AB**	**c**	**⇒**	**BA**	**c**	**⇒ac&**

PCP is not in D

MPCP ≤ PCP

Given an instance <X,Y> of MPCP, construct an instance <A,B> of PCP.

Assume $X = x_1, x_2, \ldots, x_n$

$Y = y_1, y_2, \ldots, y_n$

Construct $A = a_0, a_1, a_2, \ldots, a_n, a_{n+1}$

$B = b_0, b_1, b_2, \ldots, b_{n,} b_{n+1}$

$a_i = x_i$ with # after each symbol, for $1 \le i \le n$

$b_i = y_i$ with # before each symbol, for $1 \le i \le n$

$a_0 = \#a_1$ $\qquad b_0 = b_1$

$a_{n+1} = \$b_{n+1} = \#\$$

PCP is not in D

Example:

<*X*, *Y*>:

a	baa
ab	aa

<*A*, *B*>:

#a#	a#	b#a#a#	$
#a#b	#a#b	#a#a	#$

PCP is not in D

$<X,Y>$ has a solution iff $<A,B>$ has a solution

If $<X,Y>$ has a solution, then it is of the form $1, i_{2,} i_{3,} \ldots, i_k$.
Then $<A,B>$ has the solution $0, i_{2,} i_{3,} \ldots, i_k, n+1$.

If $<A,B>$ has a solution, then it is of the form $0, i_{2,} i_{3,} \ldots, i_k, n+1$.
Then $<X,Y>$ has the solution $1, i_{2,} i_{3,} \ldots, i_k$.

Example:

$<X,Y>$ solution:

a	**baa**
ab	**aa**

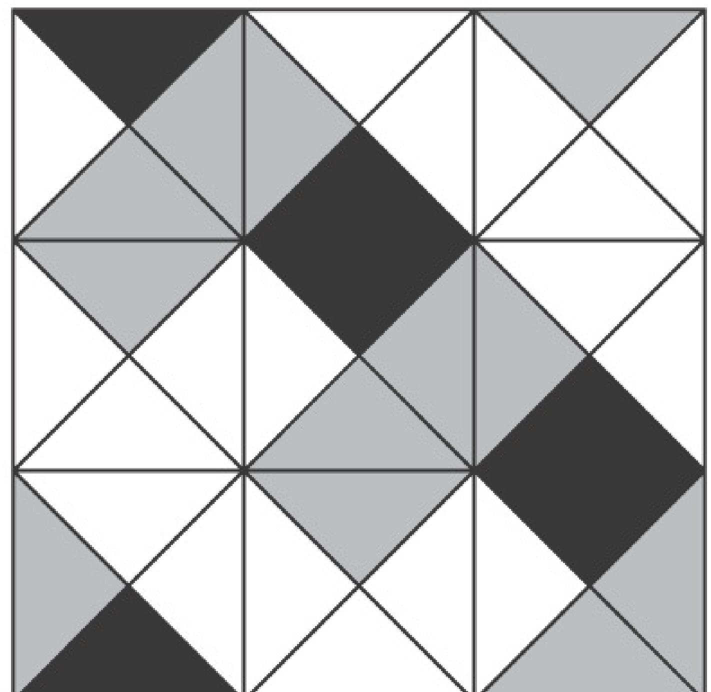
$<A,B>$ solution:

#a#	**b#a#a#**	**$**
#a#b	**#a#a**	**#$**

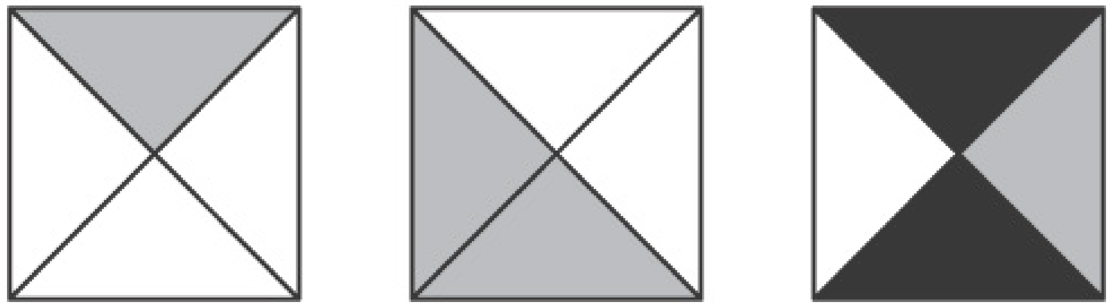
A Tiling Problem

Given a finite set *T* of tiles of the form:

Is it possible to tile an arbitrary surface in the plane?



A Set of TilesThat Cannot Tile the Plane



Is the Tiling Language in D?

We can represent any set of tiles as a string. For example, we could represent

as <*T*> = (`G W W W`) (`W W B G`) (`B G G W`).

Let **TILES** = {<*T*>: every finite surface on the plane can be tiled, according to the rules, with the tile set *T*}

Is the Tiling Language in D?

Theorem: $\neg$TILES is in SD.

Proof: Lexicographically enumerate partial solutions.

The Undecidability of the Tiling Language

Theorem: TILES is not in SD.

Proof (sketch): If TILES were in SD, then it would be in D.

But we show that it is not by reduction from $\neg H_\varepsilon$. We map an arbitrary Turing machine *M* into a set of tiles *T*:

- Each row of tiles corresponds to a configuration of *M*.
- The first row corresponds to *M*'s initial configuration when started on a blank tape.
- The next row corresponds to *M*'s next configuration, and so forth.
- There is always a next configuration of *M* and thus a next row in the tiling iff *M* does not halt.
- *T* is in TILES iff there is always a next row.
- So if it were possible to semidecide whether *T* is in TILES it would be possible to semidecide whether *M* fails to halt on ε. But $\neg H_\varepsilon$ is not in SD. So neither is TILES.

Context-Free Languages

1. Given a CFL L and a string s, is $s \in L$?
2. Given a CFL L, is $L = \varnothing$?
3. Given a CFL L, is $L = \Sigma^*$?
4. Given CFLs L_1 and L_2, is $L_1 = L_2$?
5. Given CFLs L_1 and L_2, is $L_1 \subseteq L_2$?
6. Given a CFL L, is $\neg L$ context-free?
7. Given a CFL L, is L regular?
8. Given two CFLs L_1 and L_2, is $L_1 \cap L_2 = \varnothing$?
9. Given a CFL L, is L inherently ambiguous?
10. Given PDAs M_1 and M_2, is M_2 a minimization of M_1?
11. Given a CFG G, is G ambiguous?

Reduction via Computation History

A ***configuration of a TM M*** is a 4 tuple:

(*M*'s current state,
the nonblank portion of the tape before the read head,
the character under the read head,
the nonblank portion of the tape after the read head).

A ***computation of M*** is a sequence of configurations:

$C_0, C_1, \ldots, C_n$ for some $n \geq 0$ such that:

- C_0 is the initial configuration of *M*,
- C_n is a halting configuration of *M*, and:
- $C_0 \mid\!-_M C_1 \mid\!-_M C_2 \mid\!-_M \ldots \mid\!-_M C_n$.

Computation Histories

A ***computation history*** encodes a computation:

$$(s, \varepsilon, \square, x)(q_1, \varepsilon, \text{a}, z)(\quad \ldots. \quad)(\quad \ldots. \quad)(q_n, r, s, t),$$

where $q_n \in H_M$.

Example:

$(s, \varepsilon, \square, x)$

…

$(q_1$, aaabbbaa, a, bbbbccc)

$(q_2$, aaabbbaaa, b, bbbccc)

…

CFG_{ALL} = {<*G*> : *G* cfg, *L*(*G*) = Σ^*} is not in D

We show that CFG_{ALL} is not in D.

The proof is by reduction (*R*) from H:

R will build *G* to generate the language *L#* composed of:

- all strings in Σ^*,
- ***except*** any that represent a computation history of *M* on *w*.

Then:

- If *M* halts on *w*, then there exists a computation history of *M* on *w*, so there will be a string that *G* will not generate; ¬*Oracle* will accept.

- If *M* does not halt on *w*, there are no computation histories of *M* on *w* so *G* generates Σ^* and ¬*Oracle* will reject.

Computation Histories as Strings

It is easier for *R* to build a PDA than a grammar. So *R* will first build a PDA *P*, then convert *P* to a grammar.

For a string *s* to be a *computation history* of *M* on *w*:

1. It must be a *syntactically valid* computation history.

2. C_0 must be a *starting configuration*.

3. The last configuration must be a *halting configuration*.

4. Each configuration after C_0 must be *derivable* from the previous one according to the rules in δ_M.

Computation Histories as Strings

(1) valid syntax can be checked with a regular expression
(2) hardwire into P the initial configuration
(3) hardwire into P the final configuration
(4) P will nondeterministically check that a flaw exists.

Precisely, P will guess that a configuration is not derivable from the previous one and check that.

$(q_1$, aaaa, b, aaaa$)(q_2$, aaa, a, baaaa$)$. Okay.
$(q_1$, aaaa, b, aaaa$)(q_2$, bbbb, a, bbbb$)$. Not okay.

P has to use its stack to record the first configuration and then compare it to the second. But …

Modified Computation Histories

The stack cannot be used to check that two strings are equal but it can be used to check that two strings are the reverse of each other (WW^R is context-free, WW is not).

Consider **modified** computation histories:

For C_1C_2… computation history, the modified one is:

$C_1C_2^RC_3C_4^R$…

Consider the language B# of all strings **except** modified computation histories.

P will accept B#.

The Conclusion

$R(<M, w>)$ =

1. Construct $<P>$, where P accepts all strings in $B\#$.
2. From P, construct a grammar G that generates $L(P)$.
3. Return $<G>$.

If *Oracle* exists, then $C = \neg Oracle(R(<M, w>))$ decides H:

- $<M, w> \in$ H: M halts on w. There exists a computation history of M on w. So there is a string that G does not generate. *Oracle* rejects. C accepts.
- $<M, w> \notin$ H: M does not halt on w, so there exists no computation history of M on w. G generates Σ^*. *Oracle* accepts. C rejects.

But no machine to decide H can exist, so neither does *Oracle*.

$CFG_= = \{<G_1, G_2> : G_1, G_2$ are cfgs, $L(G_1) = L(G_2)\}$

Proof by reduction from: $CFG_{ALL} = \{<G> : L(G) = \Sigma^*\}$:

R is a reduction from CFG_{ALL} to $CFG_=$ defined as follows:

$R(<M>) =$

1. Construct the description $<G\#>$ of a new grammar $G\#$ that generates Σ^*.
2. Return $<G\#, G>$.

If *Oracle* exists, then $C = Oracle(R(<M>))$ decides CFG_{ALL}:

- *R* is correct:
  - $<G> \in CFG_{ALL}$: *G* is equivalent to *G#*, which generates everything. *Oracle* accepts.
  - $<G> \notin CFG_{ALL}$: *G* is not equivalent to *G#*, which generates everything. *Oracle* rejects.

But no machine to decide CFG_{ALL} can exist, so neither does *Oracle*.

PDA_{MIN} = {<M_1, M_2>: M_2 is a minimization of M_1} is undecidable.

Recall that M_2 is a minimization of M_1 iff:

$(L(M_1) = L(M_2)) \wedge M_2$ is minimal.

R(<*G*>) is a reduction from CFG_{ALL} to PDA_{MIN} :

1. Invoke *CFGtoPDAtopdown*(*G*) to construct the description <*P*> of a PDA that accepts the language that *G* generates.
2. Write <*P#*> : *P#* is a PDA with a single state *s* that is both the start state and an accepting state. Make a transition from *s* back to itself on each input symbol. Never push anything onto the stack. $L(P\#) = \Sigma^*$ and *P#* is minimal.
3. Return <*P*, *P#*>.

If *Oracle* exists, then *C* = *Oracle*((<*G*>)) decides CFG_{ALL}:

- <*G*> $\in$ CFG_{ALL}: $L(G) = \Sigma^*$. So $L(P) = \Sigma^*$. Since $L(P\#) = \Sigma^*$, $L(P) = L(P\#)$. And *P#* is minimal. Thus *P#* is a minimization of *P*. *Oracle* accepts.
- <*G*> $\notin$ CFG_{ALL}: $L(G) \neq \Sigma^*$. So $L(P) \neq \Sigma^*$. But $L(P\#) = \Sigma^*$. So $L(P) \neq L(P\#)$. So *Oracle* rejects.

No machine to decide CFG_{ALL} can exist, so neither does *Oracle*.

Reductions from PCP

$<P> = (x_1, x_2, x_3, \ldots, x_n)(y_1, y_2, y_3, \ldots, y_n)$,
where $\forall j\ (x_j \in \Sigma+ \text{ and } y_j \in \Sigma+)$

Example:

i	X	Y
1	b	bab
2	abb	b
3	aba	a
4	bbaaa	babaaa

(b, abb, aba, bbaaa)(bab, b, a, babaaa).

From PCP to Grammar

G_x: $S_x \rightarrow bS_x1$ $\quad S_x \rightarrow b1$

$S_x \rightarrow abbS_x2$ $\quad S_x \rightarrow abb2$

$S_x \rightarrow abaS_x3$ $\quad S_x \rightarrow aba3$

$S_x \rightarrow bbaaaS_x4$ $\quad S_x \rightarrow bbaaa4$

G_y: $S_y \rightarrow babS_y1$ $\quad S_y \rightarrow bab1$

$S_y \rightarrow bS_y2$ $\quad S_y \rightarrow b2$

$S_y \rightarrow aS_y3$ $\quad S_y \rightarrow a3$

$S_y \rightarrow babaaaS_y4$ $\quad S_y \rightarrow babaaa4$

G_x could generate:

b babbb ba babbb 2 3 2 1

i	X	Y
1	b	bab
2	abb	b
3	aba	a
4	bbaaa	babaaa

IntEmpty = $\{<G_1, G_2> : L(G_1) \cap L(G_2) = \varnothing\}$

$$\text{PCP} = \{<P> : P \text{ has a solution}\}$$

$$\big\downarrow R$$

(?*Oracle*) $L_2 = \{<G_1, G_2> : L(G_1) \cap L(G_2) = \varnothing\}$

$R(<P>) =$

1. From P construct G_x and G_y.
2. Return $<G_x, G_y>$.

If *Oracle* exists, then $C = \neg Oracle(R(<P>))$ decides PCP:

- $<P> \in$ PCP: P has at least one solution. So both G_x and G_y will generate some string:

  $w(i_1, i_2, \ldots i_k)$R, where $w = x_{i1}x_{i2}\ldots x_{ik} = y_{i1}yx_{i2}\ldots y_{ik}$.

  So $L(G_1) \cap L(G_2) \neq \varnothing$. *Oracle* rejects, so C accepts.
- $<P> \notin$ PCP: P has no solution. So there is no string that can be generated by both G_x and G_y. So $L(G_1) \cap L(G_2) = \varnothing$. *Oracle* accepts, so C rejects.

But no machine to decide PCP can exist, so neither does *Oracle*.

$\text{CFG}_{\text{UNAMBIG}} = \{\langle G \rangle : G$ is a CFG and G is ambiguous$\}$

$$
\begin{array}{c}
PCP = \{\langle P \rangle : P \text{ has a solution}\} \\
\Big\downarrow R \\
\text{CFG}_{\text{UNAMBIG}} = \{\langle G \rangle : G \text{ is ambiguous}\}
\end{array}
$$

(?*Oracle*)

$R(\langle P \rangle) =$

1. From P construct G_x and G_y.
2. Construct G as follows:
   - 2.1. Add to G all the rules of both G_x and G_y.
   - 2.2. Add S and the two rules $S \rightarrow S_x$ and $S \rightarrow S_y$.
3. Return $\langle G \rangle$.

G generates $L(G_1) \cup L(G_2)$ by generating all the derivations that G_1 can produce plus all the ones that G_2 can produce, except that each has a prepended:

$$S \Rightarrow S_x \text{ or } S \Rightarrow S_y.$$

$CFG_{UNAMBIG}$ = {<*G*> : *G* is a CFG and *G* is ambiguous}

R(<*P*>) =

1. From *P* construct G_x and G_y.
2. Construct *G* as follows:
   - 2.1. Add to *G* all the rules of both G_x and G_y.
   - 2.2. Add *S* and the two rules $S \rightarrow S_x$ and $S \rightarrow S_y$.
3. Return <*G*>.

If *Oracle* exists, then *C* = *Oracle*(*R*(<*P*>)) decides PCP:

- <*P*> $\in$ PCP: *P* has a solution. Both G_x and G_y generate some string:

  $w(i_1, i_2, \ldots i_k)^R$, where $w = x_{i1}x_{i2}\ldots x_{ik} = y_{i1}yx_{i2}\ldots y_{ik}$.

  So *G* can generate that string in two different ways. *G* is ambiguous. *Oracle* accepts.
- <*P*> $\notin$ PCP: *P* has no solution. No string can be generated by both G_x and G_y. Since both G_x and G_y are unambiguous, so is *G*. *Oracle* rejects.

But no machine to decide PCP can exist, so neither does *Oracle*.