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
- Logical theories
- Context-free languages
Hilbert’s 10th Problem

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

\[ 4x^3 + 7xy + 2z^2 - 23x^4z = 0 \]

The problem: given a Diophantine system, does it have an integer solution?

Or, let Tenth = \{ <w> : w is a Diophantine system with an integer solution \}.

Is Tenth in D?

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

Consider two equal length, finite lists, \( X \) and \( Y \), of strings over some alphabet \( \Sigma \):

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

Does there exist some finite sequence of integers that can be viewed as indexes of \( X \) and \( Y \) such that, when elements of \( A \) are selected as specified and concatenated together, we get the same string that we get when elements of \( Y \) are also concatenated together as specified?

For example, if we assert that \( 1, 3, 4 \) is such a sequence, we’re asserting that \( x_1x_3x_4 = y_1y_3y_4 \).
A PCP Instance With a Simple Solution

<table>
<thead>
<tr>
<th>i</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
<td>aab</td>
</tr>
<tr>
<td>2</td>
<td>abb</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>aba</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>baaa</td>
<td>baba</td>
</tr>
</tbody>
</table>
A PCP Instance With a Simple Solution

<table>
<thead>
<tr>
<th>i</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
<td>aab</td>
</tr>
<tr>
<td>2</td>
<td>abb</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>aba</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>baaa</td>
<td>baba</td>
</tr>
</tbody>
</table>

Solution: 3, 4, 1
Another PCP Instance

<table>
<thead>
<tr>
<th>i</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>11</td>
<td>011</td>
</tr>
<tr>
<td>2</td>
<td>01</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>001</td>
<td>110</td>
</tr>
</tbody>
</table>
A PCP Instance With No Simple Solution

<table>
<thead>
<tr>
<th>i</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1101</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0110</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>110</td>
</tr>
</tbody>
</table>
# A PCP Instance With No Simple Solution

<table>
<thead>
<tr>
<th>i</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1101</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0110</td>
<td>11</td>
</tr>
<tr>
<td>3</td>
<td>1</td>
<td>110</td>
</tr>
</tbody>
</table>

Shortest solution has length 252.

[http://web.cs.ualberta.ca/~zhao/PCP/intro.htm](http://web.cs.ualberta.ca/~zhao/PCP/intro.htm)
The Language PCP

\[ <P> = (x_1, x_2, x_3, \ldots, x_n)(y_1, y_2, y_3, \ldots, y_n) \]

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:

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

The language PCP is in SD/D.
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 Tiles That 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

\[
\begin{array}{cccc}
G & W & W & W \\
\hline
W & W & B & G \\
B & G & G & W
\end{array}
\]

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 \(T\).

**Wang’s conjecture:** If a given set of tiles can be used to tile an arbitrary surface, then it can always do so periodically. In other words, there must exist a finite area that can be tiled and then repeated infinitely often to cover any desired surface.

But Wang’s conjecture is false.
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 D or SD.

**Proof:** 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 $\varepsilon$. But $\neg H_\varepsilon$ is not in SD. So neither is TILES.
The Entscheidungsproblem, Again

Does there exist an algorithm to determine, given a statement in a logical language, whether or not it is a theorem?

Suppose the answer, for a sufficiently powerful logical language, is yes. Then we could:

- Decide whether other programs are correct.
- Determine whether a plan for controlling a manufacturing robot is correct.
- Find an interpretation that makes sense for a complex English sentence.
Boolean Logic

All of the following languages are in D:

- $\text{VALID} = \{ w : w \text{ is a wff in Boolean logic and } w \text{ is valid} \}$.
- $\text{SAT} = \{ w : w \text{ is a wff in Boolean logic and } w \text{ is satisfiable} \}$.
- $\text{PROVABLE} = \{ <A, w> : w \text{ is a wff in Boolean logic, } A \text{ is a set of axioms in Boolean logic and } w \text{ is provable from } A \}$.
First-Order Logic

\( FOL_{\text{theorem}} = \{ <A, w> : A \text{ is a decidable set of axioms in first-order logic, } w \text{ is a sentence in first-order logic, and } w \text{ is entailed by } A \} \).

Example:

\[ \forall x \ (\text{bear}(x) \rightarrow \text{mammal}(x)) \]

\text{bear}(\text{Smoky})

\text{?mammal(}\text{Smoky})
First-Order Logic

$FOL_{\text{theorem}}$ is semidecidable:

$proveFOL(A, w) =$
1. Lexicographically enumerate sound proofs.
2. Check each proof as it is created. If it succeeds in proving $w$, halt and accept.

By Gödel’s Completeness Theorem, we know that there exists a complete set of inference rules for first order logic.

So step 1 of $proveFOL$ can be correctly implemented.
Complete Theories are Decidable

If $T$ is complete then, for any sentence $w$, either $w$ or $\neg w$ is a theorem. So the set of theorems is decidable by:

$\text{decidecompletetheory}(A: \text{set of axioms}, w: \text{sentence}) =$

1. In parallel, use $\text{proveFOL}$ to try to prove $w$ and $\neg w$.
2. One of the proof attempts will eventually succeed.
   If the attempt to prove $w$ succeeded, then return $True$. If the attempt to prove $\neg w$ succeeded, then return $False$.

But we must also consider the case in which $T$ is not complete. Now it is possible that neither $w$ nor $\neg w$ is a theorem.
First-Order Logic is Not Decidable

We reduce $H_\varepsilon = \{<M> : M \text{ halts on } \varepsilon\}$ to $FOL_{\text{theorem}}$:

$$R(<M>) =$$
1. From $<M>$, construct a sentence $F$ in the language of Peano arithmetic, such that $F$ is a theorem of Peano arithmetic iff $M$ halts on $\varepsilon$.
2. Let $P$ be the axioms of Peano arithmetic. Return $<P, F>$.

If Oracle exists, $C = Oracle(R(<M, w>))$ decides $H_\varepsilon$:
- $R$ exists (as shown by Turing) and is correct:
  - $<M> \in H_\varepsilon$: $M$ halts on $\varepsilon$. $F$ is a theorem of Peano arithmetic. Oracle accepts.
  - $<M> \notin H_\varepsilon$: $M$ does not halt on $\varepsilon$. $F$ is not a theorem of Peano arithmetic. Oracle rejects.

But no machine to decide $H_\varepsilon$ can exist, so neither does Oracle.
Example – The Legal System

We want a system such that:

- All allowable actions are legal and all unallowable actions are not legal.

- It is possible to answer the question, “Given an action A, is A legal?”

- The set of laws is consistent. (No action can be shown to be both legal and illegal.)

- The set of laws is finite and reasonably maintainable. So, for example, we must reject any system that requires a separate law, for each specific citizen, mandating that that citizen pay taxes.
The Legal System

Can we use Boolean logic?

Can we find a complete and consistent first-order system?
1. Given a CFL $L$ and a string $s$, is $s \in L$?
2. Given a CFL $L$, is $L = \emptyset$?
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 $L_1 \cap L_2 = \emptyset$?
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:
$C_0$, $C_1$, …, $C_n$ for some $n \geq 0$ such that:

- $C_0$ is the initial configuration of $M$,
- $C_n$ is a halting configuration of $M$, and:
- $C_0 \ |-_M\ C_1 \ |-_M\ C_2 \ |-_M\ …\ |-_M\ C_n$. 
Computation Histories

A *computation history* encodes a computation:

\[(s, \varepsilon, \square, x)(q_1, \varepsilon, a, z)(\ldots)(\ldots)(q_n, r, s, t),\]

where \(q_n \in H_M\).

Example:

\[(s, \varepsilon, \square, x)\]
\[\ldots\]
\[(q_1, aaabbbbaa, a, bbbbcccc)\]
\[(q_2, aaabbbbaaa, b, bbbccc)\]
\[\ldots\]
$\text{CFG}_{\text{ALL}} = \{ <G> : G \text{ is a cfg and } L(G) = \Sigma^* \}$ is not in D

We show that $\text{CFG}_{\text{ALL}}$ is not in D by reduction from H:

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

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

Then:

- If $M$ does not halt on $w$, there are no computation histories of $M$ on $w$ so $G$ generates $\Sigma^*$ and $\text{Oracle}$ will accept.

- If there exists a computation history of $M$ on $w$, there will be a string that $G$ will not generate; $\text{Oracle}$ will reject.
But:

- If $M$ does not halt on $w$, there are no computation histories of $M$ on $w$ so $G$ generates $\Sigma^*$ and Oracle will accept.

- If there exists a computation history of $M$ on $w$, there will be a string that $G$ will not generate; Oracle will reject.

*Oracle* gets it backwards, so $R$ must invert its response.

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.
Computation Histories as Strings

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 correspond to $M$ being in its start state, with $w$ on the tape, and with the read head positioned just to the left of $w$.

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 $\delta_M$. 
How to test (4), that each configuration after $C_0$ must be derivable from the previous one according to the rules in $\delta_M$?

$$(q_1, \text{aaaa}, b, \text{aaaa})(q_2, \text{aaa}, a, \text{baaaa}). \text{ Okay.}$$

$$(q_1, \text{aaaa}, b, \text{aaaa})(q_2, \text{bbbb}, a, \text{bbbb}). \text{ Not okay.}$$

$P$ will have to use its stack to record the first configuration and then compare it to the second. But what’s wrong?
The Boustrophedon Version

Write every other configuration backwards.

Let $B\#$ be the language of computation histories of $M$ except in boustrophedon form.

- A boustrophedon example
- Generating boustrophedon text
The Boustrophedon Version

\( R(<M, w>) = \)
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. \( R \) accepts.
- \( <M, w> \notin H \): \( M \) does not halt on \( \varepsilon \), so there exists no computation history of \( M \) on \( w \). \( G \) generates \( \Sigma^* \). \( Oracle \) accepts. \( R \) rejects.

But no machine to decide \( H \) can exist, so neither does \( Oracle \).
$\text{GG}_= = \{<G_1, G_2> : G_1 \text{ and } G_2 \text{ are cfgs, } L(G_1) = L(G_2)\}$

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

$R$ is a reduction from $\text{CFG}_\text{ALL}$ to $\text{GG}_= \text{ defined as follows:}$

$R(<M>) =$

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

If Oracle exists, then $C = \text{Oracle}(R(<M>))$ decides $\text{CFG}_\text{ALL}$:

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

But no machine to decide $\text{CFG}_\text{ALL}$ can exist, so neither does Oracle.
PDA_{MIN} = \{<M_1, M_2>: M_2 \text{ is a minimization of } M_1\} \text{ is undecidable.} 

Recall that \( M_2 \) is a minimization of \( M_1 \) iff:

\((L(M_1) = L(M_2)) \land M_2 \text{ is minimal.}\)

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

1. Invoke \( CFGtoPDA_{topdown}(G) \) to construct the description \( <P> \) of a PDA that accepts the language that \( G \) generates.

2. Write \( <P#> : P# \text{ is a PDA with a single state } s \text{ that is both the start state and an accepting state. Make a transition from } s \text{ back to itself on each input symbol. Never push anything onto the stack. } L(P#) = \Sigma^* \text{ and } P# \text{ is minimal.} \)

3. Return \( <P, P#> \).

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

- \( <G> \in \text{CFG}_{ALL} : L(G) = \Sigma^* \). So \( L(P) = \Sigma^* \). Since \( L(P#) = \Sigma^* \), \( L(P) = L(P#) \). And \( P# \) is minimal. Thus \( P# \) is a minimization of \( P \). \( \text{Oracle} \) accepts.

- \( <G> \notin \text{CFG}_{ALL} : L(G) \neq \Sigma^* \). So \( L(P) \neq \Sigma^* \). But \( L(P#) = \Sigma^* \). So \( L(P) \neq L(P#) \). So \( \text{Oracle} \) rejects.

No machine to decide CFG_{ALL} can exist, so neither does \( \text{Oracle} \).
Reductions from PCP

\[ \langle P \rangle = (x_1, x_2, x_3, \ldots, x_n)(y_1, y_2, y_3, \ldots, y_n), \]
where \( \forall j \ (x_j \in \Sigma^+ \text{ and } y_j \in \Sigma^+) \)

Example:

<table>
<thead>
<tr>
<th>i</th>
<th>X</th>
<th>Y</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b</td>
<td>bab</td>
</tr>
<tr>
<td>2</td>
<td>abb</td>
<td>b</td>
</tr>
<tr>
<td>3</td>
<td>aba</td>
<td>a</td>
</tr>
<tr>
<td>4</td>
<td>bbaaa</td>
<td>babaaa</td>
</tr>
</tbody>
</table>

\((b, abb, aba, bbaaa)(bab, b, a, babaaa)\).
From PCP to Grammar

\[ G_x: \quad S_x \rightarrow bS_x1 \]
\[ S_x \rightarrow b1 \]
\[ S_x \rightarrow babbbS_x2 \]
\[ S_x \rightarrow babbb2 \]

\[ S_x \rightarrow baS_x3 \]
\[ S_x \rightarrow ba3 \]

\[ G_y: \quad S_y \rightarrow bbbS_y1 \]
\[ S_y \rightarrow bbb1 \]
\[ S_y \rightarrow baS_y2 \]
\[ S_y \rightarrow ba2 \]
\[ S_y \rightarrow aS_y3 \]
\[ S_y \rightarrow a3 \]

\[ G_x \text{ could generate:} \]

b babbb ba babbb 2 3 2 1
\{\langle G_1, G_2 \rangle : L(G_1) \cap L(G_2) = \emptyset \}\}

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

\[ R \]

(?Oracle) \quad L_2 = \{\langle G_1, G_2 \rangle : L(G_1) \cap L(G_2) = \emptyset \}\}

\[ R(\langle P \rangle) = \]
1. From \( P \) construct \( G_x \) and \( G_y \).
2. Return \( \langle G_x, G_y \rangle \).

If Oracle exists, then \( C = \neg \text{Oracle}(R(\langle P \rangle)) \) decides PCP:
- \( \langle P \rangle \in \text{PCP}: P \) 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 \( w = x_{i_1}x_{i_2}\ldots x_{i_k} = y_{i_1}y_{i_2}\ldots y_{i_k} \).
  So \( L(G_1) \cap L(G_2) \neq \emptyset \). Oracle rejects, so \( C \) accepts.
- \( \langle P \rangle \notin \text{PCP}: P \) has no solution. So there is no string that can be generated by both \( G_x \) and \( G_y \). So \( L(G_1) \cap L(G_2) = \emptyset \). Oracle accepts, so \( C \) rejects.

But no machine to decide PCP can exist, so neither does Oracle.
**CFG_{UNAMBIG} = \{<G> : G is a CFG and G is ambiguous\}**

\[ PCP = \{<P> : P has a solution\} \]

\[ R \]

(Oracle)

\[ \text{CFG}_{UNAMBIG} = \{<G> : G is ambiguous\} \]

\[ R(<P>) = \]

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 \( <G> \).

\( G \) 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:

\[ S \rightarrow S_x \text{ or } S \rightarrow S_y. \]
\( CFG_{UNAMBIG} = \{\langle G \rangle : G \text{ is a CFG and } G \text{ is ambiguous}\} \)

\( R(<P>) = \)
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:
- \( <P> \in \text{PCP}: P \) has a solution. Both \( G_x \) and \( G_y \) generate some string:
  \( w(i_1, i_2, \ldots i_k)^R, \) where \( w = x_{i_1}x_{i_2}\ldots x_{i_k} = y_{i_1}yx_{i_2}\ldots y_{i_k} \).
  So \( G \) can generate that string in two different ways. \( G \) is ambiguous. \( Oracle \) accepts.
- \( <P> \notin \text{PCP}: P \) 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 \( Oracle \).