9. POST'S CORRESPONDENCE PROBLEM

Mihai-Valentin DUMITRU
mihai.dumitru2201@upb.ro

October 2024

At the first lecture, one of the three problems presented was "Pairs". The input to the problem was a set of tiles, with a top side and a bottom side. The question was whether you can arrange the tiles, with repetitions, such that the top sequence matches the bottom sequence.

This is actually known as *"Post's Correspondence Problem"* (PCP). It turns out that the problem is undecidable and we shall prove that result in this lecture.

Up until now, all undecidability results seemed to be very "meta" in some way: they all concerned problems whose input was a Turing Machine. What's interesting about PCP is that it will be our first example of a "natural" problem, formally proven to be undecidable.

The proof of PCP's undecidability is quite a laborious task, for which we need to introduce a few new concepts and adopt conventions about the machines considered.

But first, we need formal definitions to capture what the problem asks, what a tileset is, what a solution is etc.

1 Formal definition

1.1 tileset

Definition 9.1. A *"tileset"* is a finite sequence of pairs of strings over some alphabet Σ: $T = ((t_1, b_1), (t_2, b_2), ..., (t_n, b_n))$.

We shall refer to each pair of the sequence T by using a subscript, i.e.:

1. $T_1 = (t_1, b_1)$
2. $T_2 = (t_2, b_2)$
3. ...
4. $T_n = (t_n, b_n)$

Consider the following tileset:

$$\left(\begin{bmatrix}001\\0\end{bmatrix}, \begin{bmatrix}0\\01\end{bmatrix}, \begin{bmatrix}1\\0011\end{bmatrix}\right)$$

Here, $\Sigma = \{0, 1\}$.

This is just a sequence of three pairs; the first pair is formed of strings 001 and 0, the second of strings 0 and 01 and the third of strings 1 and 0011.

So we can formally represent the tileset above as: $((001, 0), (0, 01), (1, 0011))$.

We will define PCP as a decision problem, which means its input has to be a string over some alphabet. The formalism above provides us with a hint of how to encode a tileset as a string.

Let Σ be the alphabet of the strings which appear in the top/bottom part of a tile. We will use a new alphabet $\Sigma' = \Sigma \cup \{(,\ ,,\ )\}$; i.e. the original symbols plus the three new symbols for: open parentheses, comma, closed parentheses.

The encoding of the above tileset is: "$(001, 0)(0, 01)(1, 0011)$".

We are interested in whether a particular tileset has a "solution", so we must formally define what a solution is.

1.2 Solutions

Definition 9.2. A *solution* to a tileset $T = ((t_1, b_1), (t_2, b_2), ..., (t_n, b_n))$ is a finite sequence of indices $(i_1, i_2, ..., i_m)$ where:

$$\begin{cases} m \geq 1 \\ 1 \leq i_j \leq n, \quad (1 \leq j \leq m) \\ t_{i_1} t_{i_2} ... t_{i_m} = b_{i_1} b_{i_2} ... b_{i_m}. \end{cases}$$

Intuitively, a solution is an arrangement of one or more tiles (first line), such that if we read all symbols from the top side of the tiles left-to-right we get the exact same thing as reading all symbols from the bottom side of the tiles left-to-right (third line). A solution is allowed to use a tile multiple times.

The formal definition captures the idea that it's sufficient to provide a list of the tile indexes which appear in the solution.

The indexes are relative to the original tileset.

Consider the example:

$$\left(\begin{bmatrix} 001 \\ 0 \end{bmatrix}, \begin{bmatrix} 0 \\ 01 \end{bmatrix}, \begin{bmatrix} 1 \\ 0011 \end{bmatrix} \right)$$

One solution for this is:

$$\begin{bmatrix} 001 \\ 0 \end{bmatrix} \begin{bmatrix} 0 \\ 01 \end{bmatrix} \begin{bmatrix} 001 \\ 0 \end{bmatrix} \begin{bmatrix} 1 \\ 0011 \end{bmatrix}$$

Which can be expressed as: $(1, 2, 1, 3)$.

Bot the top and bottom parts, concatenated, form the string: 00100011.

We can now present the formal definition of PCP:

Definition 9.3.

$$\texttt{PCP}(enc(T)) = \begin{cases} TRUE & \text{there exists a solution for tileset } T \\ FALSE & otherwise \end{cases}$$

Our main result for this lecture will be proving that PCP is undecidable. The key idea is to describe a reduction from HALT. For reasons that will soon be clear, such a reduction is cumbersome, so we opt for an intermediate step. We introduce a modified version of PCP; we reduce HALT to it and, later, we will reduce this version to PCP. By the transitivity of $\leq_m$, we'll get HALT $\leq_m$ PCP, but breaking the transformation into two steps will allow us to better focus on the relevant aspects.

2 The Modified Post's Correspondence Problem (MPCP)

Our modified version of the problem is nearly-identical to the original one; inputs are still tilesets, as defined in subsection 1.1. The only difference is what constitutes a *solution*.

Definition 9.4. A *tile1-first solution* to a tileset $T = ((t_1, b_1), (t_2, b_2), ..., (t_n, b_n))$ is a finite sequence of indices $(i_1, i_2, ..., i_m)$ where:

$$\begin{cases} m \geq 0 \\ 1 \leq i_j \leq n, \quad (1 \leq j \leq m) \\ t_1 t_{i_1} t_{i_2} ... t_{i_m} = b_1 b_{i_1} b_{i_2} ... b_{i_m} \end{cases}$$

A "tile1-first solution", requires any solution to start with tile #1 of the tileset. Note that the list of indices can be empty (in the uninteresting cases where the top and bottom halves of tile #1 are identical).

Definition 9.5.

$$\texttt{MPCP}(enc(T)) = \begin{cases} TRUE & \text{there exists a tile1-first solution for tileset } T \\ FALSE & otherwise \end{cases}$$

It is MPCP to whom we'll reduce HALT. This might seem complicated, because the input to HALT is so different than the input to MPCP. To understand the transformation, we need first to introduce the notion of *"a computational history"* and how to encode one.

3 Computational histories

Definition 9.6. The *computational history* of machine M on input w is a sequence of configurations $h = (C_1, C_2, ...)$ where C_1 is the initial configuration of M on input w and, for each i, $C_i \vdash_M C_{i+1}$.
If $M[w]$ halts, then the history is a finite sequence of length n and C_n is a final configuration.

Going forward, it will be useful for us to be able to encode configurations and computational histories.

In order to do this, we employ the following convention: our configuration string will consist of all the symbols of the tape contents, but to the left of the currently scanned symbol, we shall place a new symbol that represents the current state. Thus we shall represent each configuration of a machine $M = (Q, \Sigma, \Gamma, B, q_1, \delta)$, as a string over a new alphabet $\Sigma' = \Gamma \cup Q$.

Remember machine isEven from the first lecture and consider its run on input 10010.

The *initial configuration*, before any computation step takes place is: $\mathbf{q_1}10010$[†]

The machine goes through the following configurations:

$1\mathbf{q_1}0010$

$10\mathbf{q_1}010$

$100\mathbf{q_1}10$

$1001\mathbf{q_1}0$

$10010\mathbf{q_1}\square$

$1001\mathbf{q_2}0$

$1001\mathbf{Y}0$

For non-empty $w = \mathtt{w_1w_2w_3...w_n}$, $enc(C_0^M(w)) = \mathbf{q_1}\mathtt{w_1w_2w_3...w_n}$. In the example above, enc($C_0^M(10010)$) = $\mathbf{q_1}10010$.

For the empty string ε, the encoding of the initial configuration will be $enc(C_0^M(\varepsilon)) = \mathbf{q_1}\square$ (instead of simply $\mathbf{q_1}$).

The encoding of a computational history is simply the encoding of all configurations, interspersed with a #:

$enc(h = (C_1, C_2, ..., C_n)) = \#enc(C_1)\#enc(C_2)\#...\#enc(C_n)\#$[‡].

Before we start describing the transformations from a pair (M, w) to a tileset T, we'll revisit our model for Turing Machines and simplify their structure, so that we have less edge-cases to consider. This is done with no loss of generality and, as you may later infer, will reduce our burden significantly.

4 Restricted machines

To keep our proof simple, our transformation from machines to tilesets will take as input only machines with the following properties:

- the machines never keep the head stationary; every transition moves it either left or right
- the machines never move their head to the left of the initial position (as if the tape was bounded on the left)

[†]Don't get too hung-up on the idea that "$\mathbf{q_1}$" is a symbol here. It doesn't matter that to us it looks as if it's two symbols: "q" and "1". In this context, it's a symbol specially chosen to be unambiguous with other states or any tape symbol from Γ.

[‡]You might reasonably wonder: "What about non-halting machines?". We established that an encoding should be *a string* and that strings are *finite sequences* of symbols, so we cannot encode infinite histories; we don't need to, anyway.

We can prove that, for any machine M matching our classic definition (i.e. can hold its head still, has unbounded tape in both directions), we can construct a machine M' with the restrictions above, such that $M \equiv M'$.

At the lab on computability, you have already proven that you can transform a regular machine to an equivalent one with each of the two properties, by describing two transformations. To get a machine that respects both, simply apply both transformations, one after the other.

5 Building a tileset from a (M, w) pair

To prove that HALT $\leq_m$ MPCP, we need to describe a computable transformation t whose input is the encoding of a (M, w) pair and whose output is the encoding of a tileset T, such that: $\texttt{HALT}(enc((M, w))) = TRUE \Leftrightarrow \texttt{MPCP}(enc(T)) = TRUE$.

The basic idea is to create a tileset in which the fragments on the top and bottom side of each tile can be concatenated together to form the computational history of $M[w]$. In any partial solution, the bottom half will be "ahead" in the computational history, while the top part, representing the same computational history, will lag behind, by one configuration. Only if $M[w]$ halts (its computational history is finite), will the top part be able to "catch up to" the bottom part. Completing the match entails creating the entire computational history of $M[w]$ and thus simulating $M[w]$.

Let $M = (Q, \Sigma, \Gamma, B, q_1, \delta)$ and $w = w_1 w_2 w_3 ... w_n$.

First, the transformation creates the initial tile:

$$\left[\begin{array}{c} \# \\ \#\mathbf{q_1}\mathtt{w_1}\mathtt{w_2}\mathtt{w_3}...\mathtt{w_n}\# \end{array}\right]$$

(To keep the transformation easy to follow, we'll ignore the case where $w = \varepsilon$; but you should be able to see the few changes that need to be made to also cover that case).

Remember that we are creating a tileset for MPCP; so a solution, if it exists, must start with this first tile. Notice how this tile illustrates the idea above, that the top part is "lagging behind" the bottom part – in this case, by the entire first configuration. This is the only tile that depends on the word w; the rest of the tiles depend simply on M.

For all $a, b, c \in \Gamma$ and all $r, s \in Q$:

If $\delta(r, a) = (s, b, \rightarrow)$, then add all tiles of the form:

$$\left[\begin{array}{c} \mathtt{ra} \\ \mathtt{bs} \end{array}\right]$$

except for $b = \square$, for which we add:

$$\left[\begin{array}{c} \mathtt{ra} \\ \mathtt{s} \end{array}\right]$$

If $\delta(r, a) = (s, b, \leftarrow)$, then add all tiles of the form:

$$\left[\begin{array}{c} \mathtt{cra} \\ \mathtt{scb} \end{array}\right]$$

except for $b = \square$, for which we add:

$$\left[\begin{array}{c} \mathtt{cra} \\ \mathtt{sc} \end{array}\right]$$

Remember the single-tape simulation of a multitape Turing machine, where we pointed out that, from one configuration to another, there's a limited number of symbols that can change – those around the head – the changes to the tape contents are *local*.

The two classes of tiles above capture both possible movements of the head (remember that we're now only dealing with machines which can't keep their head stationary).

In any partial solution, the top row would need to be filled in a way to resemble the last configuration of the bottom row (which is one configuration ahead!). Whatever tile we add that contains a state and surrounding symbols will

then have its bottom part contain the correct modifications for the *next configuration*, such that the bottom row will still be one configuration ahead.

We then add, for any symbol $a \in \Gamma$, the following tile:

$$\begin{bmatrix} \mathtt{a} \\ \mathtt{a} \end{bmatrix}$$

This allows matching the rest of the tape symbols, which are not in the vicinity of the head.

We also need a tile for the configuration delimiter $\#$:

$$\begin{bmatrix} \# \\ \# \end{bmatrix}$$

If the head ever reaches the end of the input and read a black symbol, we need to "create" it on demand, so we'll add a special version of the above tile, with one extra blank symbol in the bottom half:

$$\begin{bmatrix} \# \\ \square\# \end{bmatrix}$$

When the bottom half reaches a configuration in which the head has reached the end of the input, in some state r, this tile can be used to advance the partial solution, placing a blank on the bottom.

The top will have to catch up to it and it can only do so with a tile created for the transition $\delta(r, \square)$, namely one which has either $\mathbf{r}\square$ or $\mathtt{c}\mathbf{r}\square$ on top, which will force the bottom row to be completed with the correct effects of that transition.

If the bottom half ever reaches a halting configuration (one which contains a final state: Y, N, or H), we then need to let the top row "catch up" to the bottom row. We introduce several tiles to create *artificial configurations* in which the machine is in a final state and its tape symbols disappear one by one.

Thus, for each $X \in \{Y, N, H\}$ we add the tiles:

$$\begin{bmatrix} \mathtt{a}\mathbf{X} \\ \mathbf{X} \end{bmatrix}, \begin{bmatrix} \mathbf{X}\mathtt{a} \\ \mathbf{X} \end{bmatrix}$$

After adding several of these tiles, the bottom part will be ahead of the top part by just one *artificial configuration*: $\#X\#$, where $X \in \{Y, N, H\}$. I.e. the machine is in final state X and there's nothing left on the tape.

We then need to let the top part match the bottom part and we do this with the special tiles (where $X \in \{Y, N, H\}$):

$$\begin{bmatrix} \mathbf{X}\#\# \\ \# \end{bmatrix}$$

If there exists a solution, this has to be the final tile.

The tileset described here only has a solution if the top half can ever catch up to the bottom half, i.e. if the computation of $M[w]$ reaches on of the final states $\{Y, N, H\}$, halting.

The transformation t simply has to read the encoding of (M, w) and output the encoding of the tiles described above, so it is surely computable.

We have thus proven that $\texttt{HALT} \leq_m \texttt{MPCP}$, so $\texttt{MPCP} \notin R$.

We now need to return to our original problem, PCP.

6 MPCP $\leq_m$ PCP

MPCP and PCP are very similar problems; the key difference is that the solution of MPCP *has to start with the first pair of the tileset*.

To reduce MPCP to PCP means to find a transformation t that takes a tileset T and maps it to another tileset T' such that T' only has a *solution* if T has a *tile1-first solution*. I.e. $\texttt{MPCP}(enc(T)) = TRUE \Leftrightarrow \texttt{PCP}(enc(T')) = TRUE$.

We must build a tileset that obviously has no solution that doesn't start with the first pair.

Let Σ be an alphabet and $w = w_1 w_2 ... w_n$ be a string consisting of the symbols $w_1, w_2, w_3, ..., w_n \in \Sigma$. We introduce the following string operations:

1. $\star w = \diamond w_1 \diamond w_2 \diamond w_3 ... \diamond w_n$
2. $w\star = w_1 \diamond w_2 \diamond w_3 \diamond ... w_n \diamond$
3. $\star w \star = \diamond w_1 \diamond w_2 \diamond w_3 \diamond ... \diamond w_n \diamond$

Where $\diamond$ is a new symbol, i.e. $\diamond \notin \Sigma$.

The first operation adds a $\diamond$ *before* each symbol of string w, the second operation adds a $\diamond$ *after* each symbol of string w while the third one adds one $\diamond$ *before and after* each symbol of string w.

We also introduce a new "terminator symbol": $\blacksquare$.

The purpose of the three string operations and the terminator symbol is to convert each tileset that is an MPCP input:

$$\left(\begin{bmatrix} t_1 \\ b_1 \end{bmatrix}, \begin{bmatrix} t_2 \\ b_2 \end{bmatrix}, \begin{bmatrix} t_3 \\ b_3 \end{bmatrix}, ... \begin{bmatrix} t_n \\ b_n \end{bmatrix} \right)$$

To a PCP input that looks like this:

$$\left(\begin{bmatrix} \star t_1 \\ \star b_1 \star \end{bmatrix}, \begin{bmatrix} \star t_1 \\ b_1 \star \end{bmatrix}, \begin{bmatrix} \star t_2 \\ b_2 \star \end{bmatrix}, \begin{bmatrix} \star t_3 \\ b_3 \star \end{bmatrix}, ... \begin{bmatrix} \star t_n \\ b_n \star \end{bmatrix}, \begin{bmatrix} \diamond\blacksquare \\ \blacksquare \end{bmatrix} \right)$$

Each top-side string starts with a $\diamond$; the only bottom-side string that starts with a $\diamond$ is $\star b_1 \star$. It is thus obvious that, if there is a solution, it must start with the first tile:

$$\begin{bmatrix} \star t_1 \\ \star b_1 \star \end{bmatrix}$$

This is the only tile for which both the top and bottom part start with the same symbol, namely $\diamond$.

Why then did we not just add one $\diamond$ before/after the strings and we went to the trouble of defining the three $\star$ operations that intersperse $\diamond$ among all of a string's symbols? If we did that, we would invalidate solutions in which two or more top/bottom parts are used to match one bottom/top part etc.

The fact that the $\star$ operations interleave all symbols with $\diamond$ means that all matches are unaffected (notice how the first tile of the MPCP input appears in two forms: one with the bottom part surrounded by $\star$, forcing it to be the first tile, and another "normal" one, in which the bottom part is only *succeeded* by a star. This is because the tile could appear multiple times as part of the solution to MPCP, besides its first occurrence.)

Whatever solution exists for the MPCP tileset also applies to the PCP tileset, by taking the corresponding tiles. The strings formed will have the same symbols, but interspersed with $\diamond$. There is only one problem: the bottom string of the solution (the one formed by concatenating the bottom part of the tiles in the solution) will have an extra $\diamond$, compared to the top string. That is why we need the last special tile with the terminator symbol: it allows the top part to add the missing $\diamond$.

This means that MPCP $\leq_m$ PCP. Combined with the result from section 5 (HALT $\leq_m$ MPCP) and the transitivity of the $\leq_m$ relation, we obtain: HALT $\leq_m$ PCP. Thus PCP $\notin R$.

7 PCP $\in RE$

The proof of this is very simple, so we will just sketch the main idea: a machine that accepts PCP can, for any tileset T, start trying all possible sequences of length 1 to see if they represent a solution; then move on to all possible sequences of length 2, then to length 3 and so on.

If there is a solution, it will be found at some point, so the machine will halt, accepting. Otherwise, if there is no solution, the machine will keep trying longer and longer sequences, forever.

8 References and further reading

PCP was first presented and proven undecidable by Emil L. Post in his 1945 article: "A variant of a recursively unsolvable problem" [1]. However, Post's proof does not involve a reduction from HALT, using computational histories. In fact, it doesn't use the formalism of Turing Machines at all.

The proof in this lecture is adapted from chapter 5.2 of the textbook "Introduction to the Theory of Computation" [2] and chapter 9.4 of the textbook "Introduction to Automata Theory, Languages, and Computation" [3]. The latter credits it to unpublished notes of Robert W. Floyd.

Bibliography

[1] Emil L Post. "A variant of a recursively unsolvable problem". In: *Bulletin of the American Mathematical Society* 52.4 (1946), pp. 264–268.

[2] Michael Sipser. *Introduction to the Theory of Computation, Third Edition*. CENGAGE Learning, 2012. Chap. 5.2 A simple undecidable problem, pp. 227–233.

[3] John E Hopcroft, Rajeev Motwani, and Jeffrey D Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2007. Chap. 9.4 Post's Correspondence Problem, pp. 401–412.