CSCC63 – Week 5

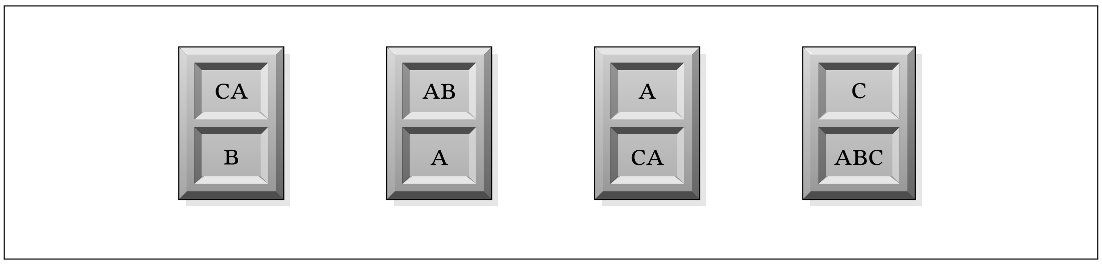
This week: We'll play with a simple undecidable problem that doesn't use TMs. We'll also relate this to a few other examples.

The main idea of this week's material is to relate the undecidability techniques we've already covered to non-TM problems. With that in mind, let's go on to the *Post Correspondence Problem.*

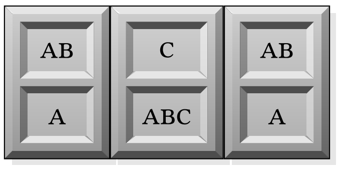
The Post Correspondence Problem

You've seen what the Post Correspondence Problem is in tutorials, but we'll briefly review it here.

Suppose you have a list of tiles like the ones below:



We can take copies of these tiles and line them up:



Notice how we use tile number two twice in this sequence. We can use each tile type as often as we want.

When we take a sequence of tiles like the one above, we can read off two strings: one from the top and one from the bottom:

Top string: ABCAB

Bottom String: AABCA

In this case the top and the bottom string are different.

The question is this: If I give you a set of tiles such as the one above, can you find a non-empty sequence of tiles such that the top and bottom strings are the same?

The Post Correspondence Problem (the PCP**)**:

Input: A finite set of tiles.

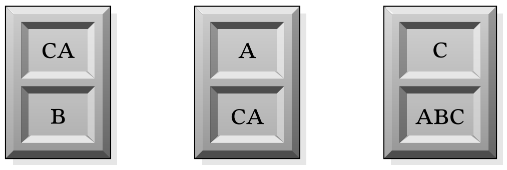
Question: Can we find a non-empty sequence of tiles such that the top and bottom strings read from that sequence are equal?

As always, the string encodings of yes-instances for this problem form a language. We'll refer to this language as the PCP*, as well.*

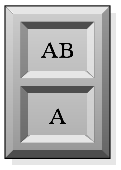
This is a simple problem, so it's probably surprising to find that it's undecidable!

Let's have a look at how we might try to solve it.

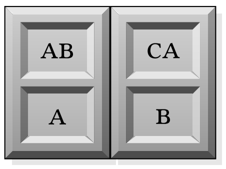
In the above tile set, we have four tiles. If we want the top and bottom strings to be equal, could we start with tiles one, three, or four?



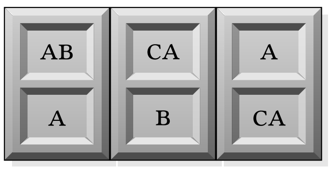
So there's only one possible first tile — tile number two:



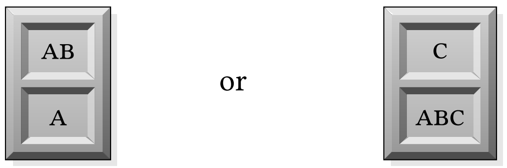
But once we have this tile set, if we want the two strings to be equal, the choice for the second string is forced — the top string has to start with a B. So any matching sequence must start with



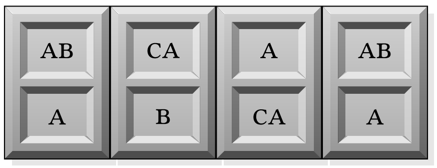
Now, the top and bottom strings match on the first two characters, but the bottom one has an extra CA. So in order to find a matching, we need a tile whose top string starts with a C:



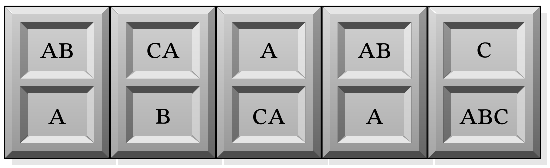
The lower string now has an extra A, and so we now have two choices for the next string:



But the second tile won't work (look at the character after the A), so we have to choose



We can now choose the ABC tile to get



And at this point, you can see that the top and bottom strings are equal, and so we would accept this tile set as a *yes*-instance of the PCP.

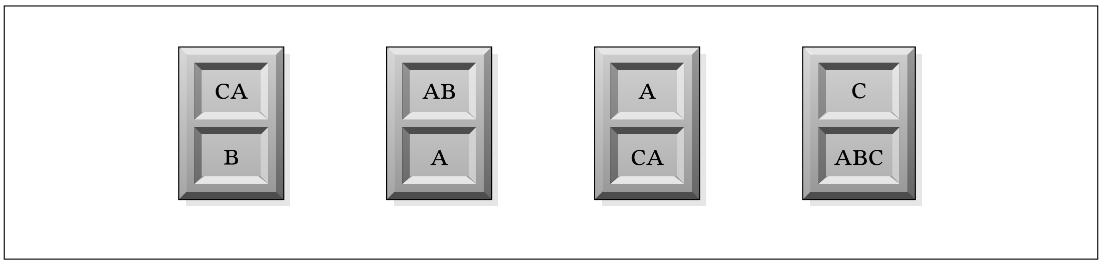
This approach to finding a solution to a PCP instance — i.e., starting with one side and growing the solution as far as we can — can leave us in a situation in which:

- we have one choice of starting tile, and
- Once we have chosen the first n tiles there is only one choice for the $n + 1^{st}$ tile.

So the steps we take when these tile sequences take are a little bit like the steps taken by a TM. We'll use this similarity to emulate a TM inside if a PCP instance.

Why is the PCP Undecidable?

The idea is that we'll prove this using a mapping reduction — we'll emulate a TM operation with a carefully chosen set of tiles.
Recall our example — when we had the tile list



- there was only one possible starting tile, and
- once we chose that starting tile, the rest of the sequence could be "grown" be predicting the one possible next tile.

We'll try to do the same thing using TM configurations.

Looking Back: the Church-Turing Thesis

You'll have noticed that we started the course by defining what a TM is, but once we did that we basically never used them — all of our undecidability results have been done using pseudocode.

And while TMs can be useful, and even interesting, they aren't exactly the most user-friendly programming formalism out there. Actually, they're more like an assembly language than anything.

So why do we bother defining them at all? Why don't we just skip straight to doing everything in pseudocode and avoid the extra source of confusion?

There are a couple of reasons, actually — one reason is to ensure that all of our code is based on a well-defined mathematical construct, even if that construct only shows up in the background.

But there's another reason, as well, and it'll show up here. The central idea of this course is that we can sometimes describe the relative difficulty of problems A and B by "embedding" problem A into problem B. If we do this, then any method of solving problem B would also solve problem A. That's basically what a mapping reduction does.

If we're trying to show a problem is undecidable, we do so by emulating a computer in that problem. This is easiest to do if the computer is mathematically simple to describe. And what would be easier to encode into something like PCP? A TM, or, say, the Python programming language?

What the Church-Turing thesis implies is that TMs can act as an assembly language for the rest of our algorithms.

A First Attempt:

We want to do a mapping reduction from HALT. So let's suppose we've been given an instance $\langle M, w\rangle$ of HALT.

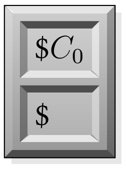
We're going to try to use M and w to build a PCP instance that has a match iff M halts on w.

Consider what happens if we run M on w. When we do, we'll get a sequence of configurations

$$\begin{array}{lcl} C_0 & = & q_0 w \\ C_1 & & \\ C_2 & & \\ & \vdots & \end{array}$$

These configurations are strings — and we build our PCP tiles using strings. So we can use the configurations to build PCP tiles.

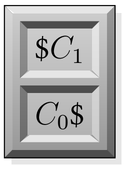
Let's build a set of PCP tiles like so: our first tile will be



> *As usual, we're using '\$' to denote a character that doesn't occur in any of the configurations.*

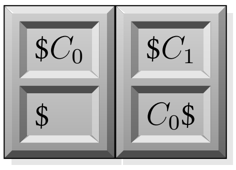
This will be like our first character from our Monday class — as long as we don't let any other tile have top and bottom strings with the same initial character, this will have to be the starting tile.

Now, we can extend the tile set by calculating the next configuration and placing it on a tile like so:



We can do this — these are just strings, after all, and we can figure out what they will be by looking at M and w.

If we try to use these tiles to build a sequence like we did last time, we get

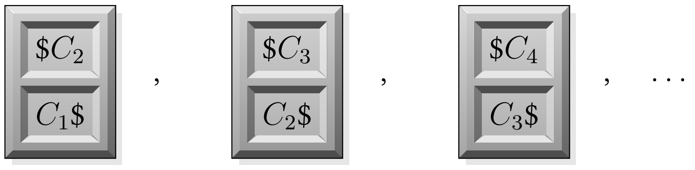


which gives us the strings

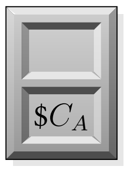
$$\begin{array}{l} \$C_0\$ \\ \$C_0\$C_1 \end{array}$$

at which point we're stuck.

…but there's no reason we can't repeat the process, right?



If we reach an accepting configuration, we just add in a finishing tile



and we're done!

…but that's way too easy. Why doesn't this work?

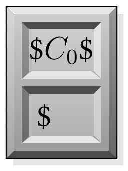
> *We need one tile for every step the TM takes. If it loops, we'd need an infinite number of tiles. But a* PCP *instance should have a finite number of tiles (the mapping reduction can't take an infinite amount of time, either).*

OK, so the specific attempt had problems, but the overall idea was pretty sound — we didn't really have any problems building the individual tiles, and the whole issue of matching the top and bottom strings worked pretty well, too.

Let's try again, but we'll be a bit more careful.

A Second Attempt:

This time around, we'll still try to build the TM configurations into tiles. In fact, we'll more or less use the same starting tile:



But this is the only configuration we'll actually write out onto a single tile. Everything else will be worked out differently — we'll try to build the configurations character by character. To see how we can do this, let's use following TM as an example (which may bring back nightmares from the first week).
In the first week, we fed the string $w = 010\#011$ into the machine. When we did, the first few configurations were:

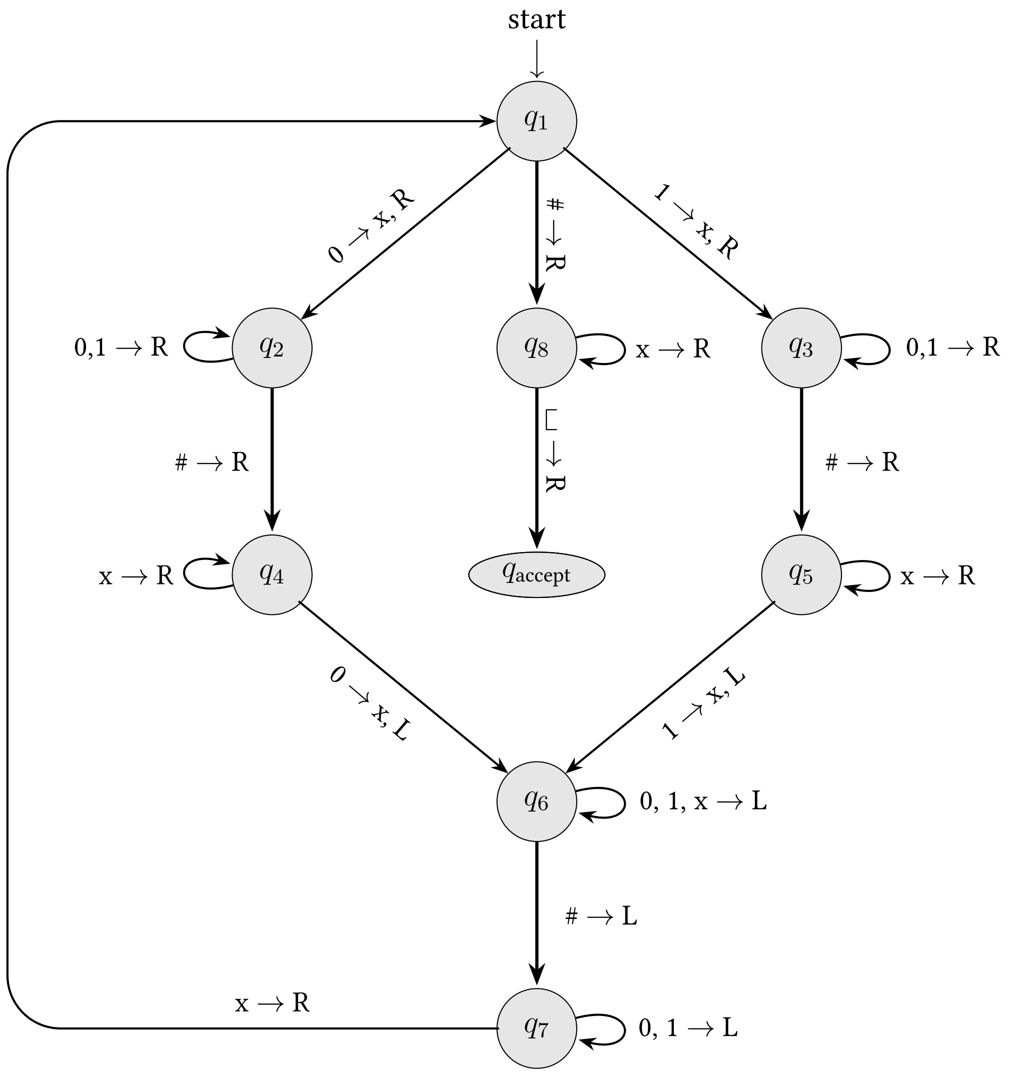
$$\begin{aligned}
C_0 &= q_1 010\#011\sqcup \\
C_1 &= \text{x}q_2 10\#011\sqcup \\
C_2 &= \text{x}1q_2 0\#011\sqcup \\
C_3 &= \text{x}10q_2\#011\sqcup \\
C_4 &= \text{x}10\#q_4 011\sqcup \\
&\ \vdots
\end{aligned}$$

OK, now: we want to build C_5. What will the first character of C_5 be?

It'll be x.

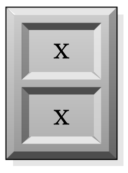
Do we need to know what the head is going to do in the next step to figure this out? Why not?

No. None of the possible steps moves the head enough to change this character.



The TM from week 1

So is there anything that prevents us from writing a tile like, say,



into our set of tiles, and then using this tile to write the first character in C_5?

Actually…yes — this tile has a matching top and bottom string. So we can use it as a sequence, and this sequence will match. If we do this we'll always end up with a yes-*instance of the* PCP.

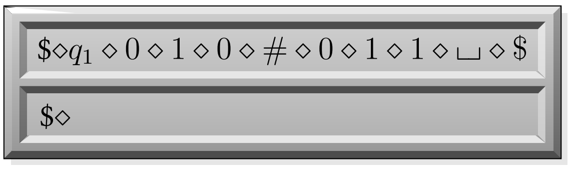
On the other hand, you can see that we can deal with this the same way that we dealt with repeating configurations in our first attempt — we can interleave yet another special character into our strings — so instead of

$$\$q_1 010\#011\sqcup$$

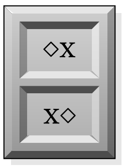
we'll write

$$\$ \diamond q_1 \diamond 0 \diamond 1 \diamond 0 \diamond \# \diamond 0 \diamond 1 \diamond 1 \diamond \sqcup.$$

We'll just stagger these characters when we build the tiles:


$\$\diamond q_1 \diamond 0 \diamond 1 \diamond 0 \diamond \# \diamond 0 \diamond 1 \diamond 1 \diamond \sqcup \diamond \$$
$\$\diamond$

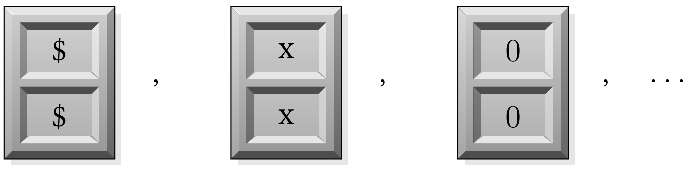
and



If we do this we can use use the single x *tile as wanted, and it won't mess up our choice of starting tile.*

. . . but that's really messy to write, so we'll ignore the $\diamond$ *characters with the understanding that they're there in the tiles, but we're not writing them unless we have to.*

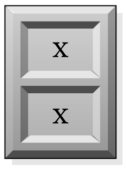
With that settled, we can write tiles for every non-state character in our TM, as well as the $ character:



There are a finite number of these tiles, and if we use the interleaving trick they won't be possible starting tiles. So we can extend most of our configurations character by character.

The only trick left if dealing with the TM head — when we use it to build C_5 *we'll be able to get the initial* x*10 using only these tiles, but then we have to deal with the* $\#$ *character, which will change.*

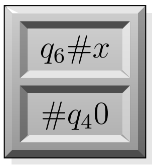
But here's the thing — tiles like



cover most of our configurations — in our assignment 1, you have to argue that there's a maximum number of characters that can change between configurations. You'll see that these characters are all adjacent, too.

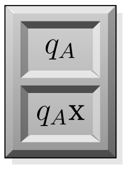
> *So if the maximum number of characters that can change is n, and if $|\Gamma| + |Q| = k$, the total number of tiles we'd need to describe these changes would be $(k+1)^{2n}$. Can you see why?*

As a further example, we'd need a tile of the form



to encode the difference between C_4 and C_5 above.

And that's the idea! The only issue is dealing with what happens if we ever reach a halting state q_A or q_R. There are a couple of ways of dealing with this, but a fairly quick way would be to let the accepting state "eat" its neighbouring characters in the configurations, e.g., using tiles like



to whittle away from the configurations when we reach an accept state.

And that's all we really need — if we start with an M that halts on w, our construction will allow us to build a matching sequence of tiles using the configuration history of M. On the other hand if we have an M that loops on w, the fact that we can only halt by building a configuration history that ends in q_A or q_R means that there will be no matching tile sequence.

And that shows that the PCP is undecidable.

Here are a few high-level takeaways to remember:

- We can encode TM operation into problems not involving TMs to get a wide range of undecidability results.
- These encodings are possible because of the local nature of head movement on the tape in TMs.

- Because of these reductions, we can effectively run TMs in many different media: if there's a problem that we can do using the PCP, we can also solve it using a TM, and so on.
- We'll see a similar TM embedding later in the course when we look at Cook's theorem: the same basic idea works (with some modifications) even if make M a nondeterministic TM, or if we say that it has to finish in a particular number of steps.

So we've shown that the PCP is undecidable — but why are we interested in this problem anyway?

Undecidability and CFGs

The reason we like the PCP is because it is a very good jumping-off point for many mapping reductions that don't use TMs.

> *How many of you remember the CFGs from CSCB36?*

As a recap, a *Context-Free Grammar (a CFG)* is a tuple (V, Σ, P, S) made up of the following elements:

- A set V of non-terminal elements.
- The set Σ is the set of terminal elements — or our alphabet. Note that no terminal is a non-terminal: $V \cap \Sigma = \varnothing$.
- A set P of production rules.
- A special start nonterminal, usually denoted S.

As an example, the CFG

$$S \to \varepsilon \,|\, 0 \,|\, 1 \,|0S0 \,|\, 1S1$$

would generate all of the palindromes over the alphabet $\Sigma = \{0, 1\}$: The set V of non-terminals would be $\{S\}$, the production rules P would be the five right-hand side options above, and the start symbol would be S.

To refresh your memory, how could we use this grammar to generate the strings 001100 and 01010?

$$S \Rightarrow 0S0 \Rightarrow 00S00 \Rightarrow 001S100 \Rightarrow 001\varepsilon100 = 001100$$

> *and*

$$S \Rightarrow 0S0 \Rightarrow 01S10 \Rightarrow 01010$$

What we want to talk about today is the fact that there are undecidable problems relating to CFGs. In particular, the language

$$\text{CFG-INTERSECTION} = \left\{ \langle G_1, G_2 \rangle \,\middle|\, G_1 \text{ and } G_2 \text{ are CFGs such that } L(G_1) \cap L(G_2) \neq \varnothing \right\}$$

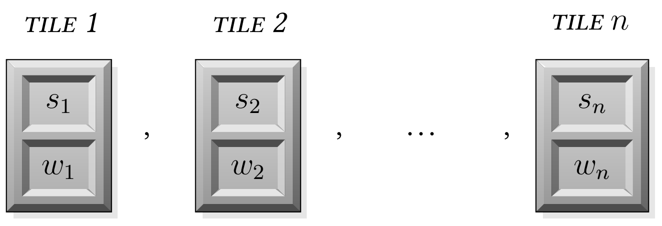
is undecidable. That is, there's no general algorithm that can tell us, given an input of two CFGs, if there's any string can be generated by both grammars!

So how can we prove this?

The overall idea is to reduce from HALT, but writing out this particular mapping reduction can be a pain. But we already said that mapping reductions are transitive (e.g., when we said that $L_2 \leqslant_m \text{ALL}_{\text{TM}}$ implies that ALL_{TM} is neither recognizable nor co-recognizable). But since we know that the PCP is undecidable, we can reduce from that instead.

OK then, how can we reduce from the PCP*?*

Here's the idea: Suppose we have a PCP instance:



Note that s_i is the string on the top of the i^{th} tile, and w_i is the string on the bottom.

Let's try writing two CFGs — one will generate the top strings from tile sequences, and the other will generate the bottom strings.

In these grammars, let t_1, t_2, t_n be special characters not present in any of the w_i or s_i. We'll use these symbols to remember which tile has been placed.

With that in mind, here are our two grammars:

$$\begin{aligned} G_1: \quad & S_1 \to s_1A_1t_1 \mid s_2A_1t_2 \mid \ldots \mid s_nA_1t_n \\ & A_1 \to \varepsilon \mid s_1A_1t_1 \mid s_2A_1t_2 \mid \ldots \mid s_nA_1t_n \end{aligned}$$

and

$$\begin{aligned} G_2: \quad & S_2 \to w_1A_2t_1 \mid w_2A_2t_2 \mid \ldots \mid w_nA_2t_n \\ & A_2 \to \varepsilon \mid w_1A_2t_1 \mid w_2A_2t_2 \mid \ldots \mid w_nA_2t_n. \end{aligned}$$

So a choice of tile 1 followed by tile 2 would correspond to a string $s_1s_2t_2t_1$ from G_1 — we'd generate

$$S_1 \Rightarrow s_1A_1t_1 \Rightarrow s_1s_2A_1t_2t_1 \Rightarrow s_1s_2\varepsilon t_2t_1 = s_1s_2t_2t_1.$$

Similarly, this choice of tiles would correspond to the string $w_1w_2t_2t_1$ in G_2.

We can see that if the top and bottom strings from the tiles are equal — that is, if $s_1s_2 = w_1w_2$, then the two strings $s_1s_2t_2t_1$ and $w_1w_2t_2t_1$ are also the same.

This works for all tile sequences, and so if the PCP instance has a match, then the two CFGs G_1 and G_2 share a common string.

We also want to prove that if our tiles are a *no*-instance of the PCP, then G_1 and G_2 have an empty intersection. Let's do this by proving the contrapositive: suppose that the two CFGs G_1 and G_2 share a common string X.

This string must be of the form $X = xt_{i_k}t_{i_{k-1}} \dots t_{i_2}t_{i_1}$, since we cannot have an empty derivation, and since every rule ends in some t_i.

Since the t_i cannot share characters with the strings on the tiles, we know G_1 must generate this string as follows:

$$S_1 \Rightarrow s_{i_1}A_1t_{i_1} \Rightarrow s_{i_1}s_{i_2}A_1t_{i_2}t_{i_1} \Rightarrow \dots \Rightarrow s_{i_1}s_{i_2} \dots s_{i_{k-1}}s_{i_k}t_{i_k}t_{i_{k-1}} \dots t_{i_2}t_{i_1} = X.$$

similarly, G_2 must generate

$$S_2 \Rightarrow w_{i_1}A_2t_{i_1} \Rightarrow w_{i_1}w_{i_2}A_2t_{i_2}t_{i_1} \Rightarrow \dots \Rightarrow w_{i_1}w_{i_2} \dots w_{i_{k-1}}w_{i_k}t_{i_k}t_{i_{k-1}} \dots t_{i_2}t_{i_1} = X.$$

For these two string to be equal, we must have that $s_{i_1}s_{i_2} \dots s_{i_{k-1}}s_{i_k} = w_{i_1}w_{i_2} \dots w_{i_{k-1}}w_{i_k}$.

But since the common sequence $t_{i_k}t_{i_{k-1}} \dots t_{i_2}t_{i_1}$ indicates that both strings are generated using the same tile sequence, this means that we have a sequence of tiles whose top and bottom strings are equal — so this is a *yes*-instance of the PCP.

So G_1 and G_2 share a common string iff our PCP instance is a *yes*-instance — in other words, this is a mapping reduction from PCP to CFG-INTERSECTION.

This means that once we've proven the PCP is undecidable, we also know that CFG-INTERSECTION is undecidable.

And that's how we can use the fact that the PCP is undecidable.

This finishes our discussion of computability. The basic ideas we've covered are:

- Formalizing our notion of computation so that we can formally talk about what is computable.
- A specific formalization of computation: Turing machines and their configurations.
- An undecidable problem: HALT.
- Different levels of undecidability: recognizability and co-recognizability.
- Characterizations of recognizability: Recognizers, Certificates and Verifiers, and Enumerators.
- Using mapping reductions to get lots of results showing non-decidability / recognizability / co-recognizability.

- Embedding TMs into non-TM problems to get general undecidability results.

The Take-Away from this Lesson:

- We introduced the Post Correspondence Problem.
- We proved that the PCP is undecidable.
- We used the fact that the PCP is undecidable to show that the intersection problem for CFGs is undecidable.
- We used the CFG intersection result to argue that CFG ambiguity is also undecidable, and that this is a result that is relevant even if you're not using TMs.

Glossary:

Oracles
The Post Correspondence Problem

Languages:

CFG-Intersection, PCP