

Computations: from Turing Machines to Tilings

Daria Pchelina

École Normale Supérieure de Paris, Высшая Школа Экономики.
dpchelina@clipper.ens.fr

Guilhem Gamard

Высшая Школа Экономики.
guilhem.gamard@normale.fr

Contents

1	A mathematical definition of computation	4
1.1	Introduction	4
1.2	Definition of Turing machines	5
1.3	Turing machines compute functions on integers	7
1.4	Turing machines can do arithmetic	9
1.5	A Turing machine interpreter in Python	10
1.6	A Turing machine that interprets Python	14
1.7	The Universality theorem	15
1.8	The Halting theorem	16
1.e	Exercises for Lecture 1	18
2	Wang tiles and the Domino Problem	20
2.1	Introduction	20
2.2	Tilings (at last!)	21
2.3	Wang's algorithm	23
2.4	Encoding Turing machines in tiles	25
2.e	Exercises for Lecture 2	33
3	The Robinson tileset	35
3.1	Introduction	35
3.2	Tiling in the large: macrotiles	36
3.3	Robinson's hierarchy of macrotiles	37
3.4	Robinson's tileset is solvable	42
3.5	The Domino Problem is undecidable (at last!)	43
3.e	Exercises for Lecture 3	49
A	Bibliography	51
B	List of figures	52

Foreword

This document exposes a few fundamental proofs about Turing machines and Wang tiles. They complement the lectures given during the “Contemporary Mathematics” summer school, in Dubna (Russia) between 19th and 30th July, 2018. We would like to thank the organizers for giving us the opportunity to speak there, as well as for making the whole event pleasant and interesting.

These notes were written for readers without deep knowledge about any specific area of mathematics. The prerequisites are: basic operations on integers (both in \mathbb{N} and in \mathbb{Z}), decomposition of an integer over a base (like $13 = \overline{1101}$ in base 2), the concept of *sequence* and the concept of *function*. We will occasionally use a few more advanced things, but you can always skip them and continue to read: you should be able to understand anyway.

On the other hand, just because you need little knowledge to start studying tilings, doesn't mean that the subject is easy. Some proofs presented (or outlined) here took years to develop. To make things worse, many of the original research papers (and even later books) on the subject omit a lot of details and let the readers do the work. In these notes, we tried to make the exposition as precise as possible: consequently, the result is sometimes a bit cumbersome to read. Please do not be afraid by the symbols or the definitions: the underlying ideas are often very simple. Take your time, read things slowly, and skip parts that seem too confusing. You do not need to understand all the technical details in order to extract interesting ideas from these notes.

There are lectures, divided into sections, divided into paragraphs. Thus a section number looks like 2.4 (Lecture 2, Section 4) and a paragraph number looks like 2.4.3 (Lecture 2, Section 4, Paragraph 3). When we want to refer to another paragraph, we write something like §1.1.1.

Numbers between brackets like [2] refer to the bibliography at the end of the notes. Numbers between parentheses like (2.1) refer to an equation (Chapter 2, Equation 1).

If you have any question (or comment for improving these notes), please do not hesitate to contact us. Our email addresses appear on the first page of this document. You can use either English, Russian or French. An up-to-date version of these notes should be available at <https://www.mccme.ru/dubna/2018/courses/gamard-pchelina.html> and, otherwise, please contact one of us.

Thanks for reading and attending!

Daria and Guilhem

Lecture 1

A mathematical definition of computation

1.1 Introduction

1.1.1. The purpose of this lecture is to show how we can talk mathematically about computers and programming. Our first step is to give a definition of a computer program. Since we want to build a mathematical theory, we won't rely on the specific details of any programming language; on the other hand, we would like to use the intuition that we have when writing code, so we will try to capture what underlies most programming languages available today.

1.1.2. There are two very important concepts found in (almost) all programming languages: *source code* and the *memory*.

The memory is just a huge array of integers, while the source code is a sequence of instructions that tells how to change the memory. These instructions sometimes depend on a condition, like

```
if  $x < 10$ , then  $x \leftarrow x - 10$ ; else  $x \leftarrow x + 1$ 
```

or should be repeated several times in a loop, like

```
while  $x > 1$ , do  $x \leftarrow x/2$ .
```

Therefore, the best representation of source code is a *graph*, like the one on Figure 1.2 (don't pay attention to the symbols in the nodes for now). Using graphs, instead of text, for the source code frees us from the syntactic details of the language: we do not need to care whether a loop is written `while ... end` or `loop ... end`: a loop is just a cycle in the graph.

Memory is less simple than it seems at first. Indeed, each memory cell can only contain finitely many different values: for instance, the integers between 0 and $2^{32} - 1$ (included). If memory cells were able to contain arbitrarily large integers, the resulting computer would be *much* more powerful than what we have today. Therefore this constraint is important, and we'll need to take it into account in our mathematical model. This means that we'll have to *encode* all the objects on which we would like to compute (integers, real numbers, functions, graphs, etc.) into sequences of integers bounded by $2^{32} - 1$.

1.1.3. Programs have *inputs* and *outputs*. Those concepts do not have any precise definition: the input of a program may be a file stored on the disk, something typed by the user on a keyboard, or even something pointed on a screen by a mouse. Similarly, the output may be written on a disk, displayed on a screen, sent to a printer, sent to another computer over the Internet, etc.

Our mathematical model will not deal with all these complicated kinds of input and output. We will assume that the input of our program has been loaded in memory somehow; we'll perform

computations, and then write the output back in the memory. The details about how this input arrived in memory and what the output becomes after we write it are not important to us.

1.1.4. Once we have a mathematical definition of a program, the first thing to do is to check that it is a *good* definition. Ideally, we would like that our mathematical “programs” are “equivalent” to some real programming language, such as Python. If we do this, then all the things that we prove mathematically apply at least to the Python programming language, so they have some practical consequences. This is easier said than done, though, because the Python programming language itself has no precise mathematical definition. We will see how to overcome these problems.

Finally, we can start to prove interesting things about programs. We will only show two easy, but very important results here: the *Universality theorem*, which says that what is true for one programming language is (to some extent) true for all of them; and the *Halting theorem*, which says that some *computations cannot be done by a computer program*.

1.2 Definition of Turing machines

1.2.1. Let \mathbb{I} denote the set $\{0, \dots, 2^{32} - 1\}$. Theoretically we could take any finite set for \mathbb{I} (with at least 2 elements), but the integers modulo 2^{32} is a popular choice for electronic computers these days.

A *memory* is an array of unbounded length, where each cell contains an element of \mathbb{I} . (Formally, a memory is a function from \mathbb{N} to \mathbb{I} .) Having arrays of unbounded length may seem unrealistic, but we will never use infinitely many cells at the same time. Thanks to this definition, the memory will always be big enough to run our (theoretical) programs, so we will never have to worry about running out of memory.

1.2.2. A *head* is a device which moves on a memory and is able to read and write numbers in cells, one cell at a time. There are four possible *operations*: L , R , W_i and T_i (where i is an integer in \mathbb{I}):

- L means “move the head by one cell to the left”;
- R means “move the head by one cell to the right”;
- W_i means “write the integer i in the cell under the head”;
- T_i means “test if the cell under the head is equal to i ” (more on this later).

In W_i and in T_i , the integer i is part of the operation. Intuitively, i is a constant “hardcoded in the source code” of a program. Figure 1.1 illustrates the memory and the head.

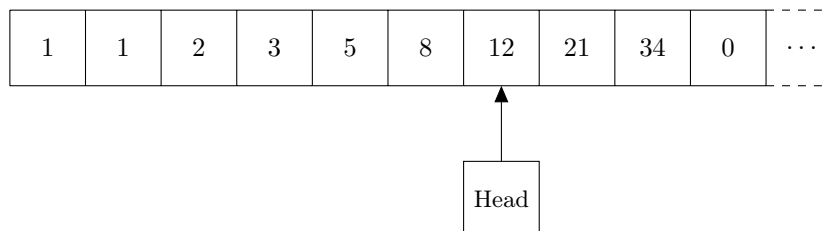


Figure 1.1: The memory (filled with random numbers) and the head.

Note that in electronic computers, memory does not quite work like that. Indeed, these computers usually have less than 2^{32} memory cells, so they can use elements of \mathbb{I} to tell in which memory cell

they want to read or write. However, our theoretical model allows to take any finite set for \mathbb{I} , even one with two elements, and also requires that the memory is unbounded. Considering a head which moves one cell at a time solves these problems: we do not use \mathbb{I} at all to talk about the memory. As we will see in the next section, this simplification does not change what is possible or not: computers with electronic memory and computers with heads can do exactly the same computations.

1.2.3. A *Turing machine* (or a *program*) is a graph with labeled nodes, such that one node is labeled **start**, one node is labeled **end**, and each other node is labeled with an operation (as defined in §1.2.2). Moreover the edges satisfy the following restrictions:

- if a node is labeled **start**, L , R , or W_i , then it has only one outgoing edge;
- if a node is labeled T_i , then it has exactly two outgoing edges (labeled **true** and **false**);
- if a node is labeled **end**, then it has no outgoing edge.

Note that several different vertices might be labeled with the same operation.

Intuitively, each node is an operation and an edge going out of a node points to the next operation to perform. The operations L , R and W_i only have one possible outcome, so there is an unique outgoing edge. By contrast, T_i is a test, so it has two possible outcomes: either the condition is true, or it is false. The edge **true** is followed in the first case, and the edge **false** is followed in the second case.

Figure 1.2 shows an example of a Turing machine.

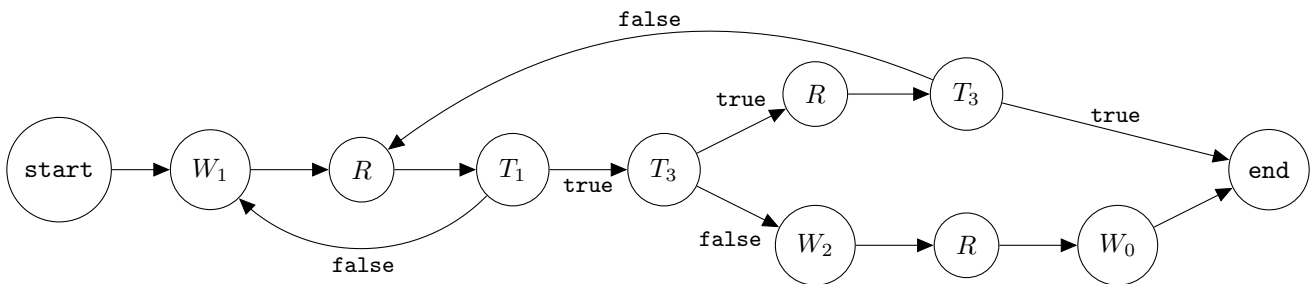


Figure 1.2: Example of a program (that doesn't do anything interesting).

1.2.4. A *configuration* is a tuple (P, v, M, h) where:

- P is a Turing machine (see §1.2.3);
- v is a node of P ;
- M is a memory (a function $\mathbb{N} \rightarrow \mathbb{I}$);
- h is a nonnegative integer (i.e. $h \in \mathbb{N}$).

Intuitively, P is the program that we are running, v is the next instruction we're about to execute, M is the contents of the memory, and h is the current position of the head in the memory. In short, a configuration is the state in which the computer is at a given point of time.

If (P, v, M, h) is a configuration, then the *successor configuration* is the configuration we get after executing the instruction in v . Let \mathcal{O} denote the label of v . If v has only one outgoing edge, then let v' denote the target of that edge; if v has two outgoing edges, then let v' and v'' denote respective the targets of these two edges. The successor configuration of (P, v, M, h) is formally defined as follows:

- if $\mathcal{O} = \text{start}$, then the successor is (P, v', M, h) ;
- if $\mathcal{O} = L$, then the successor is $(P, v', M, \max(h - 1, 0))$;
- if $\mathcal{O} = R$, then the successor is $(P, v', M, h + 1)$;
- if $\mathcal{O} = W_i$, then the successor is (P, v', M', h) , where M' is defined as $M'(h) = i$ and $M'(n) = M(n)$ for $n \neq h$;
- if $\mathcal{O} = T_i$ and $M(h) = i$, then the successor is (P, v', M', h) ;
if $\mathcal{O} = T_i$ and $M(h) \neq i$, then the successor is (P, v'', M', h) ;
- if $\mathcal{O} = \text{end}$, then there is no successor.

We write $\text{succ}(c)$ for the successor of a configuration c .

1.2.5. Let c_0 denote a configuration and for each positive integer n , let $c_{n+1} = \text{succ}(c_n)$, if it exists. The sequence c_0, c_1, c_2, \dots is called a *computation*. We write $c_n = (P, v_n, M_n, h_n)$ for each n where c_n is defined.

If we have $\text{label}(v_n) = \text{end}$ for some integer n , then the computation is finite, because c_n has no successor configuration. In this case we say that the computation *terminates* in n steps. Moreover, we'll say that M_0 is the *input* of the computation and that M_n is the *output*.

On the other hand, if the computation never reaches a configuration c_n with $\text{label}(v_n) = \text{end}$, then the sequence c_0, c_1, c_2, \dots is infinite. We say that the computation *does not terminate* on input M_0 .

If we have a Turing machine P and a memory M_0 , we'll write $T(M_0)$ for the output of the computation starting with $c_0 = (P, \text{start}, M_0, 0)$, if that computation terminates. Otherwise, $T(M_0)$ is not defined. We will also say things like *the output of P on M_0 is...* to talk about $T(M_0)$.

1.2.6. A computation may be represented as a *space-time diagram*, such as the one on Figure 1.3. A diagram consists of an infinite grid, where line number n represents the n^{th} configuration of the computation. Each cell of the grid represents one cell of the memory, and the head is drawn inside the cell on which it is.

1.3 Turing machines compute functions on integers

1.3.1. Now we have formal definitions of a *program*, a *computation*, an *input* and an *output*. However, our inputs and outputs only talk about sequences of elements of \mathbb{I} . In many cases, it will be much more convenient to talk about natural (nonnegative) integers; we mean *all* the integers, not just up to $2^{32} - 1$. We now show how to encode natural integers (i.e., any element of \mathbb{N}) into memory.

Since each memory cell contains an element of $\mathbb{I} = \{0, \dots, 2^{32} - 1\}$, it is natural to write integers in base 2^{32} , so that each integer is a sequence of cells. However, the memory of a Turing machine is infinite, while the decomposition of an integer in a base (whatever the base) is always finite. We need to fill the rest of the memory with something; in computer science, it is common to fill unused memory with zeroes. Therefore, we could write that the representation of an integer is the decomposition of this integer in base 2^{32} , followed by infinitely many zeroes.

Here we run into a big problem: we have no way to detect when the representation of an integer is finished! Consider 10000000000000000001 (written in base 2^{32}), for instance. We cannot tell whether

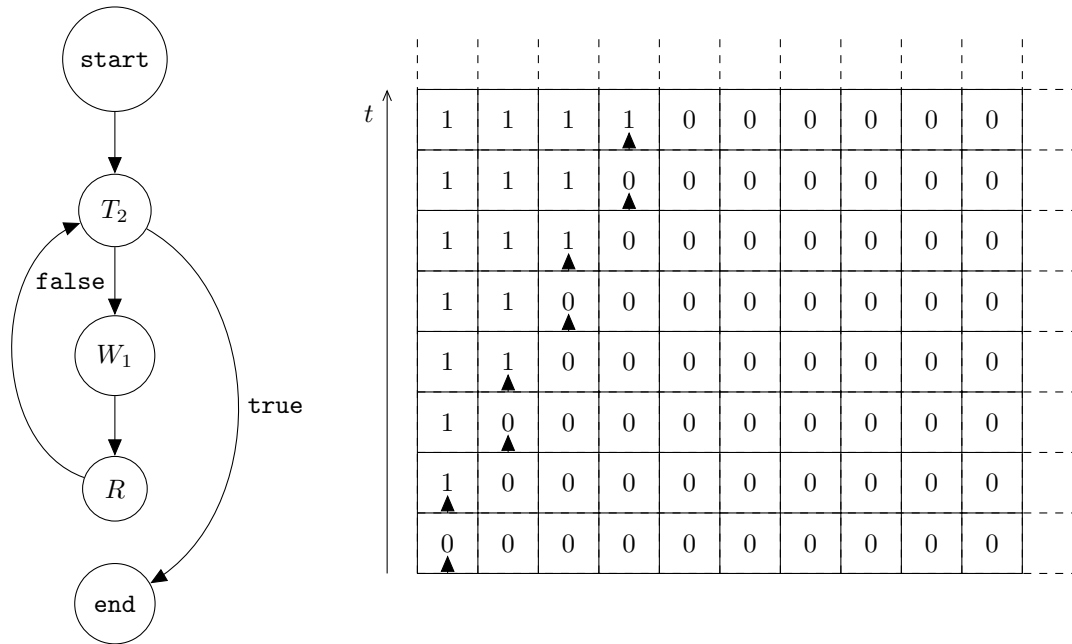
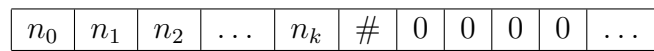


Figure 1.3: A space-time diagram of a Turing machine.

a long sequence of zeroes is part of an integer, or if it is just the infinite filling that comes after the representation. We need a way to mark the end of an integer in memory.

1.3.2. The easiest solution is to write integers in base 2^{31} , and to use the digit 2^{31} itself (which never appears in decompositions in base 2^{31}) as a marker of ending. For convenience, we will write the symbol $\#$ for the digit 2^{31} . Moreover, we will write integers with the lowest-significant digit first, which is the reverse of the usual convention. If n is an integer and $n_k \dots n_1 n_0$ is its decomposition in base 2^{31} , then the memory containing just n looks like this:



If n is an integer, we note \bar{n} its representation.

1.3.3. For any integer $n \in \mathbb{N}$, there is a Turing machine that, starting from a memory full of zeroes, writes \bar{n} on the memory. This machine is displayed on Figure 1.4. Note that, at the end of the execution, the head is positioned after $\#$.

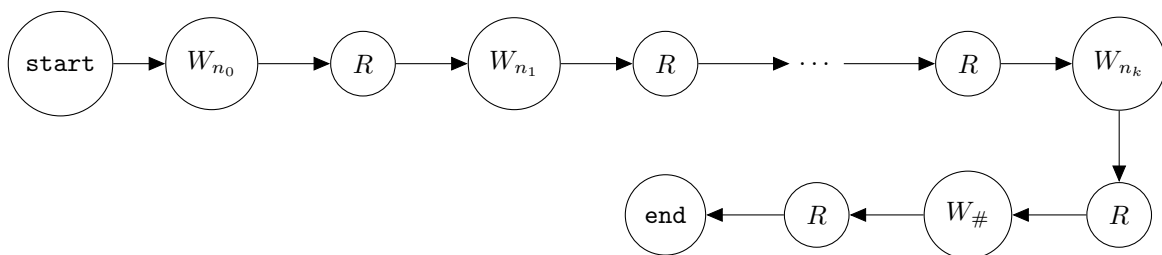


Figure 1.4: Turing machine writing the integer $n = n_k \dots n_1 n_0$ in the memory.

1.3.4. Let T denote a Turing machine and f a function defined on a subset of \mathbb{N} . We say that P computes f if and only if:

- f is defined on an integer n if and only if P terminates on input \bar{n} ;
- $f(n) = P(\bar{n})$ for each n where f is defined.

We say that a function f is *computable* if and only if there exists a program P such that P computes f . If f only returns booleans (0 or 1), we sometimes say that f is *decidable* instead of computable. The following question is fundamental in computer science:

Question. Is there a function f , defined over a subset of the integers, which is **not** computable?

We will see the answer later in this lecture. If you are impatient, you can try a counting argument. (How many functions are there? How many Turing machines are there?)

1.3.5. The composition of two computable functions is computable.

1.4 Turing machines can do arithmetic

Exercise 1.4.1. Let \mathcal{A} denote the set $\{+, -, \times, \div, <, =\}$. We suppose that the operators $<$ and $=$ return the integer 1 instead of **True**, and the integer 0 instead of **False**. Let m, n denote two arbitrary integers. For each \dagger in \mathcal{A} , design a Turing machine that, on input $\bar{m}\#\bar{n}\#$:

m_0	m_1	...	m_k	#	n_0	n_1	...	$n_{k'}$	#	0	0	0	...
-------	-------	-----	-------	---	-------	-------	-----	----------	---	---	---	---	-----

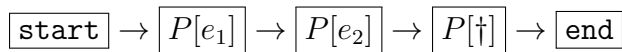
terminates and outputs $\bar{\ell}\#$, where $\ell = m \dagger n$. Thus, the memory looks like this after the execution:

ℓ_0	ℓ_1	...	$\ell_{k''}$	#	0	0	0	...
----------	----------	-----	--------------	---	---	---	---	-----

and the head is positioned after the #.

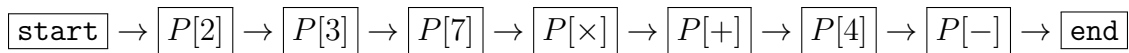
1.4.2. Using the Turing machines from §1.3.1 and Exercise 1.4.1, we can translate any arithmetic expression that contains integer constants and operators from \mathcal{A} . Given an expression, we define P as follows:

- $P[i]$, where i is an integer constant, is just the machine defined in §1.3.1;
- $P[e_1 \dagger e_2]$ is defined as:



where $P[\dagger]$ is a Turing machine from Exercise 1.4.1.

Let us look at an example. The expression $2 + 3 \times 7 - 4$ will be translated as (mind the priority of \times over $+$ and $-$):



To understand how this works, let us observe the contents of the memory after each box is executed (we do not write the infinite part of the memory full of zeroes at the end):

- after **start**: nothing (only zeroes)
- after $P[2]$: $\bar{2}\#$
- after $P[3]$: $\bar{2}\#\bar{3}\#$
- after $P[7]$: $\bar{2}\#\bar{3}\#\bar{7}\#$
- after $P[\times]$: $\bar{2}\#\bar{21}\#$
- after $P[+]$: $\bar{23}\#$
- after $P[4]$: $\bar{23}\#\bar{4}\#$
- after $P[-]$: $\bar{19}\#$

As you can see, we can visualize the memory as a stack of integers, where the top of the stack is on the right. A block $P[i]$, for an integer i , pushes the integer i on the top of the stack. A block $P[\dagger]$, for an operation \dagger , pops two integers from the stack, computes the result of the operation on these two numbers, and pushes the result back on the stack. (We should remember to move the head to the left before each operation.)

It turns out that any arithmetic expression can be evaluated on this stack form. The **Forth** programming language, for example, is based on stacks [3].

Exercise 1.4.3 (Optional). Write a Python class that models an arithmetic expression, i.e., a binary tree where internal nodes have labels in \mathcal{A} and leaves have labels in \mathbb{N} . Using the class `Node` from §1.5.4 below, write a translator from arithmetic expressions to Turing machines.

1.5 A Turing machine interpreter in Python

We saw that Turing machines are able to perform the usual arithmetic operations over the integers. However, this is not enough to prove that this model is as powerful as usual programming languages, such as Python. So, the purpose of this section is to prove the following result.

Theorem 1.5.1. *If a function is computable by a Turing machine, then there exists a Python program which computes the same function.*

1.5.2. To be fully formal, we would need a mathematical definition of Python. However, we saw in Section 1.2 that it was already hard to define Turing machines, and Python is orders of magnitude more complicated than that! Fortunately we can still give convincing arguments which use our intuitive understanding of Python.

By the way, the ideas of this section apply as well to almost any programming language. We chose Python because it is rather popular, but any language (C++, Common Lisp, Forth, Erlang, Smalltalk...) will do. Some programming languages even have mathematically precise definitions, most notably Scheme [7] and Haskell [1], so all the arguments of this section could be turned to a fully formal proof for Scheme or Haskell. Unfortunately, this would require a *huge* amount of work, and we have better things to do for today.

The easiest way to prove Theorem 1.5.1 is to write a *Turing machine interpreter* in Python. In other words, we want to write a function in Python that takes a Turing machine P and an integer n as an input and that returns $P(\bar{n})$, if it exists. Let us call this function `compute`. How does it help to prove the theorem? Suppose that we have `compute` in Python. Then, for any computable function f , there exists a Turing machine P which computes f ; so we can just write a Python program that calls `compute` on that Turing machine (we can hardcode the machine in our program).

Exercise 1.5.3. Write a Turing machine interpreter (the function `compute`) in Python.

The rest of this section gives the solution to that exercise. You may skip it if you are confident that Turing machine can indeed be run by Python programs. If not, please try to write a Python interpreter by yourself before reading on!

1.5.4. Let us start to write some code. The next listing shows the definition of a class that handles Turing machines. It follows closely §1.2.3.

```
class Node():
    def __init__(self, label, next=None, i=None):
```

```

# Label can only be one of these:
if not label in ["L", "R", "W", "T", "start", "end"]: raise(ValueError)
# We need an integer i for W_i and T_i
if label in ["W", "T"]:
    if not (type(i) is int): raise(ValueError)
# But we don't want i for any other label
else:
    if not (i is None): raise(ValueError)
# Finally, we can initialize our node
self.label = label
self.i = i
self.next = next          # This represents next_if_true for T_i
self.next_if_false = None # Only used for T_i

# The next 4 lines allow to write v.next_if_true for v.next
@property
def next_if_true(self): return self.next
@next_if_true.setter
def next_if_true(self,value): self.next = value

# A few convenient functions to build nodes
def start(next=None): return Node("start", next)
def end(): return Node("end")
def L(next=None): return Node("L", next)
def R(next=None): return Node("R", next)
def W(i, next=None): return Node("W", next, i)
def T(i, next_if_true=None, next_if_false=None):
    v = Node("T", next_if_true, i)
    v.next_if_false = next_if_false
    return v

```

1.5.5. The next listing shows the definition of a class that models configurations and that can compute successors, following §1.2.4. The only difference between the code and the formal definition is that the `Configuration` class doesn't keep a pointer to the Turing machine P , but only a pointer to the current node v . Besides that, the code is a straightforward translation of the math.

```

class Configuration():
    def __init__(self, node, memory=[0], head=0):
        if head >= len(memory): raise(ValueError)
        if not type(node) is Node: raise(TypeError)

        self.node = node
        self.memory = memory
        self.head = head

    def successor(self):
        if self.node.label == "start":
            self.node = self.node.next

```

```

elif self.node.label == "L":
    self.head = max(self.head-1, 0)
    self.node = self.node.next

elif self.node.label == "R":
    self.head = self.head+1
    # Since the memory is infinite, we extend it as needed
    if len(self.memory) == self.head: self.memory.append(0)
    self.node = self.node.next

elif self.node.label == "W":
    self.memory[self.head] = self.node.i
    self.node = self.node.next

elif self.node.label == "T":
    if self.memory[self.head] <= self.node.i:
        self.node = self.node.next_if_true
    else:
        self.node = self.node.next_if_false

elif self.node.label == "end": pass # Nothing to do!

return self

```

1.5.6. Finally, the next listing shows three functions. The first two are functions to convert an arbitrary integer to its memory representation (defined in §1.3.1) and back. The third function is `compute`: it takes a Turing machine P and an integer n as parameters, and computes $P(\bar{n})$. Note that if P enters an infinite loop, then the `compute` function will also loop!

```

def integer_to_representation(n, base=2**31):
    result = []
    while True:
        q,r = n//base, n%base
        result.append(r)
        if q == 0:
            result.append(base) # The end marker is 2**31
            return result
        else:
            n = q

def representation_to_integer(representation, base=2**31):
    result = 0
    n = 1
    for i in representation:
        if i >= base: break # Stop if we found the end marker
        result += i * n
        n *= base

```

```

return result

def compute(program, n):
    # We suppose that `program` points to the `start` node
    if not program.label == "start": raise(ValueError)

    M = integer_to_representation(n)
    c = Configuration(program, M, 0)
    while c.node.label != "end":
        c = c.successor()
    return representation_to_integer(c.memory)

```

1.5.7. Now, it is not hard to translate any Turing Machine into a Python program. The next listing shows how to write the Turing machine from Figure 1.2 in Python. The idea is to first create all the nodes using the functions `start`, `L`, `R`, `W`, `T` and `end`. Then, for each node v , set all the edges by changing $v.next$ (or $v.next_if_true$ and $v.next_if_false$ if it's a T node). Once the graph is created, simply call `compute` on it and on the desired integer n to compute $P(\bar{n})$.

```

def machine_of_fig1(n):
    ## Nodes
    tm = [
        # Left branch
        start(),
        W(1),
        R(),
        T(1),
        T(3),
        # Upper branch
        R(),
        T(3),
        # Lower branch
        W(2),
        R(),
        W(0),
        # End
        end(),
    ]

    ## Vertices
    # Left branch
    tm[0].next = tm[1]
    tm[1].next = tm[2]
    tm[2].next = tm[3]
    tm[3].next_if_true, tm[3].next_if_false = tm[4], tm[1]
    tm[4].next_if_true, tm[4].next_if_false = tm[5], tm[7]
    # Upper branch
    tm[5].next = tm[6]
    tm[6].next_if_true, tm[6].next_if_false = tm[2], tm[10]

```

```

# Lower branch
tm[7].next = tm[8]
tm[8].next = tm[9]
tm[9].next = tm[10]

## Finally, we can just call compute
return compute(tm, n)

```

The whole code given above makes an implementation of the Turing machine from Figure 1.2 in Python. This code can trivially be adapted to any other Turing machine. By our remarks from §1.5.2, this code is enough to prove Theorem 1.5.1. \square

1.6 A Turing machine that interprets Python

Still because we want to argue that Turing machines are a good model of programming languages, we would like to prove the converse of Theorem 1.5.1.

Theorem 1.6.1. *For each Python function f defined over a subset of \mathbb{N} , there exists a Turing Machine P that computes f .*

The remarks in §1.5.2 also apply here: almost any programming language could be used instead of Python, and ideally we would need a mathematical definition of Python (or whatever language we use). Ultimately, though, our main task is to translate an arbitrary Python program into a Turing machine. Since this is a *very* difficult task, we will split it into two steps, and we will skip the details of the first step.

Claim 1.6.2. *Let Simple Python denote the restriction of the Python programming language that only allows integers and arrays, arithmetic ($+$, $-$, \times , \div , $=$ and $<$), *if/else* statements, and *while* loops. Any Python program can be translated in Simple Python. Moreover, **the translator itself can be written in Simple Python.***

Indeed, all features of normal Python that are not in Simple Python (such as strings, dictionaries, objects, *for* loops, exceptions, etc.) can ultimately be translated into arrays of integers. The Python interpreter actually does this in order to execute Python code, but we will not explain the details. If you are interested, you can refer to a recent textbook about compilers.

It might be hard to believe that the Python \rightarrow Simple Python translator can be written in Simple Python. However, it seems perfectly reasonable that such a translator may be written in normal Python. For instance, Pypy [5] is a Python interpreter which is written itself in Python; it does a translation to Simple Python as a part of its interpretation process. So, suppose that we have a translator written in normal Python. Then, we can run the translator *on its own source code* in order to get a translator written in Simple Python.

Exercise 1.6.3 (Optional). Rewrite the Turing machine interpreter of Section 1.5 in Simple Python.

1.6.4. Our task is now to write a (mathematical) function T that translates any Simple Python function to a Turing machine that computes the same function. We already have Turing machines for arithmetic expressions: this was §1.4.2. We still need to handle variables, arrays, *if/else* instructions and *while* loops.

Exercise 1.6.5. (a) For each integer i in \mathbb{N} , design a Turing machine that, on input:

v_0	#	v_1	#	...	v_{k-1}	#	#	0	0	0	...
-------	---	-------	---	-----	-----------	---	---	---	---	---	-----

writes a copy of v_i after the ##. Note that v_0, v_1, \dots, v_{k-1} are *blocks* of several cells, not just single cells.

Hint: you need to temporarily change the original v_i in order to write a copy, but you can restore it later.

(b) Conversely, for each integer i in \mathbb{N} , design a Turing machine that, on input:

v_0	#	v_1	#	...	v_{k-1}	#	#	v_k	#	0	0	0	...
-------	---	-------	---	-----	-----------	---	---	-------	---	---	---	---	-----

erases v_i and writes a copy of v_k instead. Note that v_k might be shorter or longer than v_i ! Besides, we might have $i \geq k$; in this case, cells containing just 0 should be inserted to expand the array up to the right size.

(Instead of writing this Turing machine by hand, you might want to write a program that produces this Turing machine. You can use the Python code developed in the previous section as a starting point.)

1.6.6. A 2-memory Turing machine a Turing machine with two memories. In other terms, in a 2-memory configuration, each memory cell contains a couple of numbers instead of a single number. Call (i, j) the contents of the memory cell just under the head. Instructions **start**, **end**, L and R work the same as with usual Turing machines. There are two instructions for testing: T and T' . The first one tests against i , while the second one tests against j . Similarly, there are two writing instructions: W and W' ; the first rewrites i while the second rewrites j .

Exercise 1.6.7. Prove that a function f is computable by a Turing machine if and only if it is computable by a 2-memory Turing machine.

1.6.8. Thanks to Exercise 1.6.7, we can use two memories instead of one. Therefore, we can use one memory to evaluate arithmetic expressions (as in the previous section) and one to store the values of variables, using the same schema as in Exercise 1.6.5. Arrays are handled similarly, since they are just variables that occupy more than one cell of memory.

Exercise 1.6.9. Suppose we have two Turing machines T_1, T_2 , and T_3 for Python programs P_1, P_2 , and P_3 respectively. Design Turing machines that do:

- `if(P_1): P_2 else: P_3`
- `while(P_1): P_2`

1.6.10. We have implemented all the features of Simple Python with Turing machines, therefore, Theorem 1.6.1 is proved. \square

1.7 The Universality theorem

Here is a direct consequence of Theorems 1.5.1 and 1.6.1.

Proposition 1.7.1. *Turing machines are equivalent to Python, in the sense that a function defined over a subset of \mathbb{N} is computable by a Turing machine if and only if it is computable in Python.*

It helps to establish the following result.

Theorem 1.7.2 (The Universality Theorem). *There is a sequence of Turing machines $(P_n)_{n \in \mathbb{N}}$ and a Turing machine U such that:*

- *all Turing machines appear in (P_n) : for each machine Q , there is an integer i such that $Q = P_i$;*
- *if U is run on input $\bar{m} \# \bar{n} \#$ for two integers m and n , then it computes $P_m(n)$.*

The second point means that if P_m terminates on input \bar{n} with output \bar{r} , then U terminates on input $\bar{m} \# \bar{n} \#$ with output \bar{r} . Conversely, if P_m does not terminate on input \bar{n} , then U does not terminate on input $\bar{m} \# \bar{n} \#$.

Proof. By Theorem 1.5.1, any Turing machine can be translated to a Python program. So for the sequence (P_n) just set P_n to be the Python program represented by the integer n (viewed as a sequence of bits). Then, the machine U is just a Python interpreter. There exists a Turing machine that interprets Python, because there exists a Python program that interpret Python (Pypy [5]), and because any Python program can be translated into a Turing machine by Theorem 1.6.1. \square

1.7.3. Theorem 1.7.2 says that, if we have a program written somewhere in the memory, then it is possible to run that program as if it were part of the source code. Or, in other terms: the source code may be in memory instead of being fixed in advance. This allows to do metaprogramming: programs that write programs (and then, run them).

1.8 The Halting theorem

1.8.1. For this section, we start with a practical question called the **Halting problem**:

Given a Turing machine T and an input x , does T terminates on input x ?

Thanks to Proposition 1.7.1, it is equivalent to ask this question about Turing machines and about Python programs. Thus the task is as follows: we are given (the source code of) a Python function P and a number x , and we need to compute whether $P(x)$ is going to meet an infinite loop, or it will return something. Since infinite loop are usually not a desirable feature, it is interesting to detect them.

Theorem 1.8.2. *The halting problem is undecidable.*

By this we mean that no Turing machine H (or no Python program) exist that takes another Turing machine P (or another Python program) and a memory M as an argument, such that $H(P, M) = 1$ if $P(M)$ terminates and $H(P, M) = 0$ if $P(M)$ does not terminate. In other terms, it is not possible to algorithmically test whether a given program loops or not.

Proof. Suppose a function `halt()` exists in Python somehow, and consider the following program:

```
# Suppose that halt(P,x) returns 0 if P(x) loops and 1 otherwise
def problem(s):
    if halt(s, s):
        while True: pass
    else:
        return 0

# What does this print?
print (problem(problem))
```


If this program terminates, then by definition this means that ‘problem(problem)’ has an infinite loop. But if ‘problem(problem)’ has an infinite loop, then the last line of the code also has an infinite loop, and this program does not terminate: we have a contradiction.

Conversely, if this program does not terminate, then this means that ‘problem(problem)’ terminates. But if ‘problem(problem)’ terminates, then the last line will terminate, so the whole program will terminate for sure. We have again a contradiction.

So, in any case, supposing that `halt()` exists in Python gives a contradiction. By Proposition 1.7.1, this is equivalent to say that the corresponding Turing machine cannot exist either (it would also yield a contradiction). \square

An important consequence is that computers are not all-powerful: there is at least one problem (namely, the halting problem) that they cannot solve. Besides, the Halting theorem is also useful to prove that other problems are undecidable, thanks to a technique called *reduction*.

Lemma 1.8.3 (Reduction). *Let U and (P_n) be as in Theorem 1.7.2. Call $h(n, x)$ the function such that $h(n, x) = 1$ if $P_n(x)$ terminates, and $h(n, x) = 0$ otherwise. If f is a function and a an algorithm such that $f(a(n, x)) = h(n, x)$ for each n and each x , then f is uncomputable.*




Proof. Indeed, if f were computable, then the function $n, x \mapsto f(a(n, x))$ would be computable (the composition of two computable functions is computable). But then h would be computable, which is a contradiction with Theorem 1.8.2. \square

This lemma just says that, if we want to prove that a function f is uncomputable, all we have to do is to find an algorithm a such that $f(a(x, n)) = h(x, n)$ for all x and all n . We will see examples of application of this lemma in the exercises, and in next lectures.

1.e Exercises for Lecture 1

Here are all the exercises from the lecture notes, reorganized and renumbered to make an exercise sheet. There are also a few new questions, to keep things interesting. Solve only what you like.

Exercise 1.e.1. In this exercise, we deal with numbers that might be greater than 2^{31} (but still nonnegative). Recall that integers are represented as sequences $n_0, n_1, \dots, n_k, \#$, where $\# = 2^{31}$ and $\bar{n} = n_k \dots n_1 n_0$ in base 2^{31} . Differently said: the first digit to appear is the lowest-significant one.


- (a) Design a Turing machine that computes the function $n \mapsto n + 1$.
- (b) Let $z = 2^{31} - 1$; draw the space-time diagram of the Turing machine you designed for (a) on the input $z, z, z, \#$.
- (c) Design a Turing machine that computes the function $m, n \mapsto m + n$.
- (d) Design a Turing machine that computes the function $m, n \mapsto 0$ if $m = n$ and 1 otherwise.
-  (e) Design a Turing machine that computes the function $m, n \mapsto 0$ if $m \leq n$ and 1 otherwise.
-  (f) Design a Turing machine that compute multiplication of integers.
-  (g) Design Turing machines that compute substraction and division of integers. *Hint:* reuse your addition/multiplication machines, and use a brute-force algorithm.

Exercise 1.e.2.

- (a) Write a Turing machine interpreter in Python (or any language you like).
- (b) Improve your interpreter so that it shows the space-time diagram of the computation that it is running.

Exercise 1.e.3. Design a Turing machine that takes a sequence of integers between 0 and $2^{31} - 1$ (included), terminated by $\#$, and sorts that sequence in nondecreasing order.


Exercise 1.e.4.


- (a) Prove that a function f is computable by a Turing machine if and only if it is computable by a 2-memory Turing machine (cf. §1.6.6).
- (b) Write the definition of a k -memory Turing machine, and prove that k -memory and ℓ -memory Turing machines are equivalent (in the sense of the previous question) for all $k, \ell \geq 1$.
- (c) Prove that the set of computable functions is the same for any finite set \mathbb{I} with at least two elements. In other terms, whether $\mathbb{I} = \{0, \dots, 2^{32} - 1\}$ or $\mathbb{I} = \{0, 1, \dots, 9\}$ or $\mathbb{I} = \{0, 1\}$ does not change which functions are computable.
-  (d) Prove that a Turing machine where the memory is bi-infinite, i.e., a function $\mathbb{Z} \rightarrow \mathbb{I}$ instead of $\mathbb{N} \rightarrow \mathbb{I}$, can compute the same functions as a normal Turing machine.

Exercise 1.e.5. Find, on the internet:

- (a) An undecidable problem other than the Halting problem. Try to understand its proof.
- (b) A compiler from any programming language to Turing machines. Try it on simple programs and look at the resulting machines.

 (c) Try to read a bit of the source code of the compiler you found.

Exercise 1.e.6 (). Write a Python class that models an arithmetic expression, i.e., a binary tree where internal nodes are labeled by arithmetic operations ($+$, $-$, \times , \div , $=$, $<$) and leaves have labeled by integers. Then, write a translator from arithmetic expressions to Turing machines.

Exercise 1.e.7 (). Feel free to change \mathbb{I} in order to have more additional symbols: for instance, you can set $\mathbb{I} = \{2^{33} - 1\}$.

- (a) For each integer i in \mathbb{N} , design a Turing machine that, on input:


v_0	#	v_1	#	...	v_{k-1}	#	#	0	0	0	...
-------	---	-------	---	-----	-----------	---	---	---	---	---	-----

writes a copy of v_i after the $\#\#$. Note that v_0, v_1, \dots, v_{k-1} are *blocks* of several cells, not just single cells. *Hint*: you need to temporarily change the original v_i in order to write a copy, but you can restore it later.




- (b) Conversely, for each integer i in \mathbb{N} , design a Turing machine that, on input:

v_0	#	v_1	#	...	v_{k-1}	#	#	v_k	#	0	0	0	...
-------	---	-------	---	-----	-----------	---	---	-------	---	---	---	---	-----

erases v_i and writes a copy of v_k instead. Note that v_k might be shorter or longer than v_i ! Besides, we might have $i \geq k$; in this case, cells containing just 0 should be inserted to expand the array up to the right size.

Exercise 1.e.8 (). Suppose we have three Turing machines T_1 , T_2 , and T_3 for Python programs P_1 , P_2 , and P_3 respectively. Design Turing machines that do:

- `if(P_1): P_2 else: P_3`
- `while(P_1): P_2`

Exercise 1.e.9 (  ). Write a Simple Python \rightarrow Turing machine compiler. (The definition of Simple Python is in Claim 1.6.2). Reuse the constructions you designed in the previous exercises!

Contacts

- Daria Pchelina (dpchelina@clipper.ens.fr)
- Guilhem Gamard (guilhem.gamard@normale.fr)

Lecture 2

Wang tiles and the Domino Problem

2.1 Introduction

In the previous lecture, we learned about the notion of *computability*. A problem is computable if and only if there exists an algorithm solving it; thus, it can possibly be done by a computer. On the other hand, we saw that some problems were *not* computable. As mathematicians, one thing that we would like to do is to make our computers prove theorems for us, automatically. Hence the natural question: is theorem-proving computable? Let us look at an example.

2.1.1. Suppose that we have a logic formula like:

$$\forall x \in \mathbb{N}, \exists y \in \mathbb{N}, \forall z \in \mathbb{N}, \phi(x, y, z)$$

where ϕ is a formula without quantifiers (i.e., without \forall and \exists) and without free variables (i.e., all variables are either x , y or z). However, ϕ might contain function names, predicate names, and constant names. One important question is: does this formula have a *model*? In other terms, can we find an actual function for each function name appearing in ϕ , an actual predicate (i.e. a function returning a boolean) for each predicate name appearing in ϕ , and an actual integer for each constant name appearing in ϕ , so that the whole formula $\forall x \exists y \forall z \phi(x, y, z)$ is true?

For instance, if we want to find a model for the formula:

$$\forall x \in \mathbb{N}, \exists y \in \mathbb{N}, \forall z \in \mathbb{N}, G(y, x) \wedge [G(y, z) \implies (\neg D(z, y) \vee G(t, z))] \quad (2.1)$$

then we need to find two predicates G and D , as well as a constant t , which make this formula true. Here is an example of such a model:

- $G(m, n)$ is “ $m > n$ ”;
- $D(m, n)$ is “ m is a divisor of n ”;
- t is the constant 2.

With these interpretations for G , D , and t , the formula means “there are infinitely many prime numbers”, which is obviously true. (The formula reads: for each integer x , there is an integer $y > x$ such that, for each integer $z < y$, either z is not a divisor of y or $z < 2$.)


Be careful, though: the formula (2.1) does not necessarily mean “there are infinitely many primes”. It might have another, very different model where G , D , and t are completely unrelated to prime numbers. In other words, we just gave one *possible interpretation* (i.e., one *model*) of the formula that makes it true, and this interpretation means that there are infinitely many primes (which is true). Other interpretations making the formula true (models) are possible. Of course, there also are interpretations that make the formula false, but they are usually not very interesting to explore.

2.2 Tilings (at last!)

2.2.1. In 1961, Hao Wang tried to write a program that automatically checks whether an arbitrary $\forall \in \mathbb{N} \exists \in \mathbb{N} \forall \in \mathbb{N}$ formula has a model. He found a way to *translate* the problem of finding a model to a much simpler combinatorial problem, which he describes as follows:

Assume we are given a finite set of square plates of the same size with edges colored, each in a different manner. Suppose further there are infinitely many copies of each plate (plate type). We are not permitted to rotate or reflect a plate. The question is to find an effective procedure by which we can decide, for each given finite set of plates, whether we can cover up the whole plane (or, equivalently, an infinite quadrant thereof) with copies of the plates subject to the restriction that adjoining edges must have the same color.

— Wang, 1961 [9]

In other words, a *tile* is a square of unit size with colored edges, like this: . A set of tiles is called a *tileset*; each tile in a tileset can be copied infinitely many times. Now the question is: given a tileset A , can we cover the whole plane with copies of tiles from A , such that two adjacent borders always have the same color? Such a covering is called a *valid tiling* (or just a *tiling*). Figure 2.1 shows a tileset and a piece of a tiling of the plane.

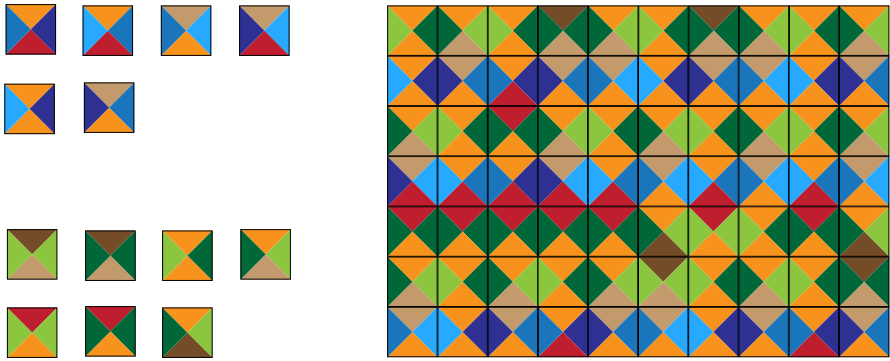


Figure 2.1: A tileset and a (part of) a valid tiling by this tileset.

It changes nothing if we consider that the plane is equipped with a grid made of unit squares, and that our task is to fill each square with a copy of a tile (respecting the color rule). Thus, each cell of the grid is uniquely determined by the coordinates of its bottom left-hand corner. A *valid tiling* is, formally, a function from \mathbb{Z}^2 to A , which satisfies the previously stated color constraints. Therefore, if τ is a tiling, we will write $\tau(0,0)$ for the tile which lies in the cell of coordinates $(0,0)$. More generally, if D is a subset \mathbb{Z}^2 , a *valid tiling of D* (by the tileset A) is a function from D to A which satisfies the color constraints.

Notation. If a is a tile, then we denote by $\text{north}(a)$, $\text{west}(a)$, $\text{south}(a)$ and $\text{east}(a)$ respectively the color on top, on the left, on bottom or on the right of a .

Capital letters $A, B, C...$ usually denote tilesets, while small letters a, b, c usually denote tiles. Capital letters $T, U, V...$ denote tilings.

2.2.2. Some tilesets have valid tilings of the plane; for instance, the set $\left\{ \begin{matrix} \text{orange} & \text{blue} \\ \text{blue} & \text{orange} \end{matrix} \right\}$ trivially has tiling (see Figure 2.2). Conversely, some tilesets do not have any valid tiling; for instance $\left\{ \begin{matrix} \text{orange} & \text{blue} \\ \text{red} & \text{green} \end{matrix} \right\}$,

because there is red on the right of the first tile, but no tile has red on the left side; and there is green on the bottom of the second tile, but no tile has green on the top side. If a tileset has a valid tiling of the plane, then we say that this tileset is *solvable*. Conversely, a tileset without any tiling of the plane is called *unsolvable*.

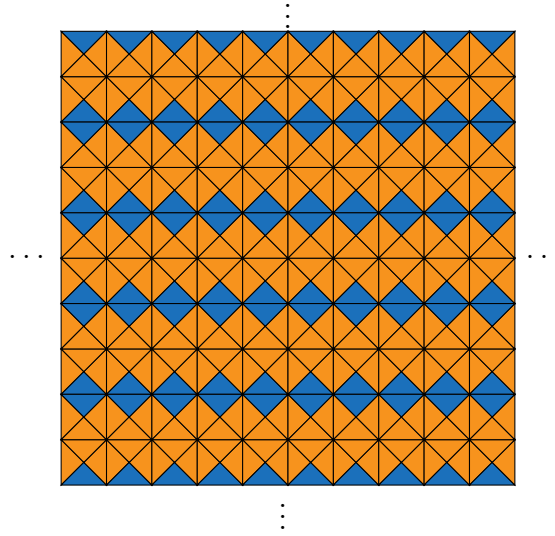


Figure 2.2: A simple tiling.

Question 2.2.3 (The Domino Problem). Given a tileset A , is A solvable?

As mentioned above, Wang could translate the problem of finding a model for any $\forall \in \mathbb{N} \exists \in \mathbb{N} \forall \in \mathbb{N}$ formula into the Domino problem. More precisely, Wang wrote a program that takes a formula ϕ as input and returns a tileset A as output, such that ϕ has a model if and only if A has a tiling of the plane. Since this algorithm is very complicated, we will not give the details here.

In order to do automatic model-finding, the only remaining step is to find an algorithm for the domino problem. First, the next theorem is very useful when we try to tile the plane.

Theorem 2.2.4 (Compactness). *If a tileset A can tile (in a valid way) a square of any finite size, then it can tile the whole plane.*

Exercise. Prove the Compactness theorem. (Solution below.)

Proof of Theorem 2.2.4 (Compactness). Let $T(n)$ denote, for each integer n , a valid tiling of an $n \times n$ square by A . Consider $(x_n)_{n \in \mathbb{N}}$ a sequence of elements of \mathbb{Z}^2 that covers \mathbb{Z}^2 , for instance a spiral (see Figure 2.3). Since there are finitely many tiles in A and infinitely many $T(n)$'s, the pigeonhole principle implies that there are infinitely many $T(n)$'s that agree on the tile at position x_0 . In other terms, there is a tile a_0 and an infinite subsequence of $T(n)$, let us call it $T(\alpha_0(n))$, such that

$$\forall n \in \mathbb{N}, \quad T(\alpha_0(n))(x_0) = a_0.$$

Now there are infinitely many tilings in $T(\alpha_0(n))$ and finitely many tiles in A . Thus there exists a tile a_1 and an infinite subsequence of $T(\alpha_0(n))$, let us call it $T(\alpha_1(n))$, such that

$$\forall n \in \mathbb{N}, \quad T(\alpha_1(n))(x_0) = a_0 \quad \text{and} \quad T(\alpha_1(n))(x_1) = a_1.$$

By recurrence, for each integer k , there exists an infinite subsequence of $T(n)$, call it $T(\alpha_k(n))$, such that:

$$\forall i \in \{0, \dots, k\}, \forall n \in \mathbb{N}, \quad T(\alpha_k(n))(x_i) = a_i.$$

Now define U to as follows: for each n in \mathbb{N} , let $U(x_n) = T(\alpha_n(0))(x_n)$ ¹. Since each point in \mathbb{Z}^2 is somewhere in the x_n 's, U is a function $\mathbb{Z}^2 \rightarrow A$; but it might not be a *valid* tiling, because there might be two touching square borders with different colors. Suppose this is the case, for instance that there exist positions (i, j) and $(i + 1, j)$ such that $\text{east}(U(i, j)) \neq \text{west}(U(i + 1, j))$. By construction of U , there are infinitely many tilings in $T(n)$ such that $T(n)(i, j) = U(i, j)$ and $T(n)(i + 1, j) = U(i + 1, j)$. More precisely, if $(i, j) = x_k$ and $(i + 1, j) = x_{k'}$, then set $m = \max(k, k')$ and any tiling in $T(\alpha_n(n))$ has this property (for any n). But all the tilings in $T(n)$ are valid, so all the touching borders have the same color: a contradiction. As a conclusion, U is a valid tiling of the plane. \square

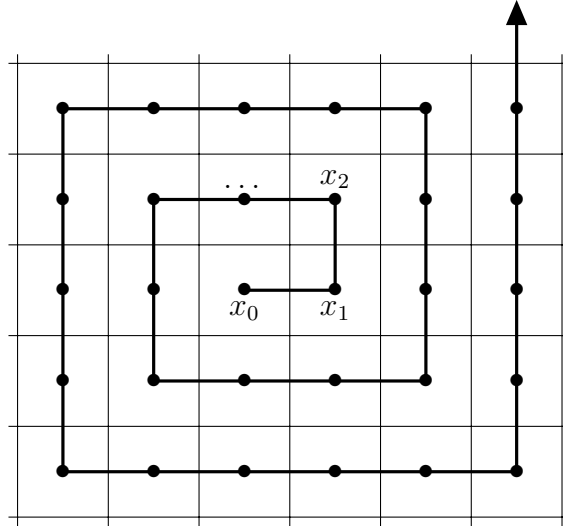


Figure 2.3: A sequence of elements of \mathbb{Z}^2 that covers \mathbb{Z}^2 .

2.3 Wang's algorithm

2.3.1. A tiling T is called *periodic* if there exists a nonzero vector \vec{v} in \mathbb{Z}^2 such that, for any x in \mathbb{Z}^2 , we have $T(x) = T(x + \vec{v})$. In other terms, T is invariant under translation by \vec{v} . In this case, we call \vec{v} a *period* of the tiling. We insist that the zero vector $(0, 0)$ is never a period.

Moreover, if \vec{v} and \vec{v}' are two periodicity vectors of T and k, k' two nonzero integers, then $k\vec{v} + k'\vec{v}'$ is also a periodicity vector of T .

Exercise 2.3.2 (Optional). Let T denote a solvable tileset. Prove that if T have a valid tiling with one period \vec{v} , then it has a tiling with two periods \vec{v}' and \vec{v}'' which are not colinear (proportional).

2.3.3. A tiling of a finite square is called *repeatable* if it is valid and if four copies of this square arranged in a 2×2 fashion (like on Figure 2.4) are also a valid tiling. More formally, consider a tileset A , a discrete square $\{0, \dots, n - 1\}^2$ (this is a cartesian product), and a tiling $T : \{0, \dots, n - 1\}^2 \rightarrow A$ of this square; then T is repeatable if and only if, for all $i \in \{0, \dots, n - 1\}$,

- the bottom side of $T(0, i)$ has the same color as the top side of $T(n - 1, i)$; and
- the right side of $T(i, n - 1)$ has the same color of the left side of $T(i, 0)$.

¹Formally we need some form of axiom of choice to justify that U actually exists, for instance Koning's lemma.

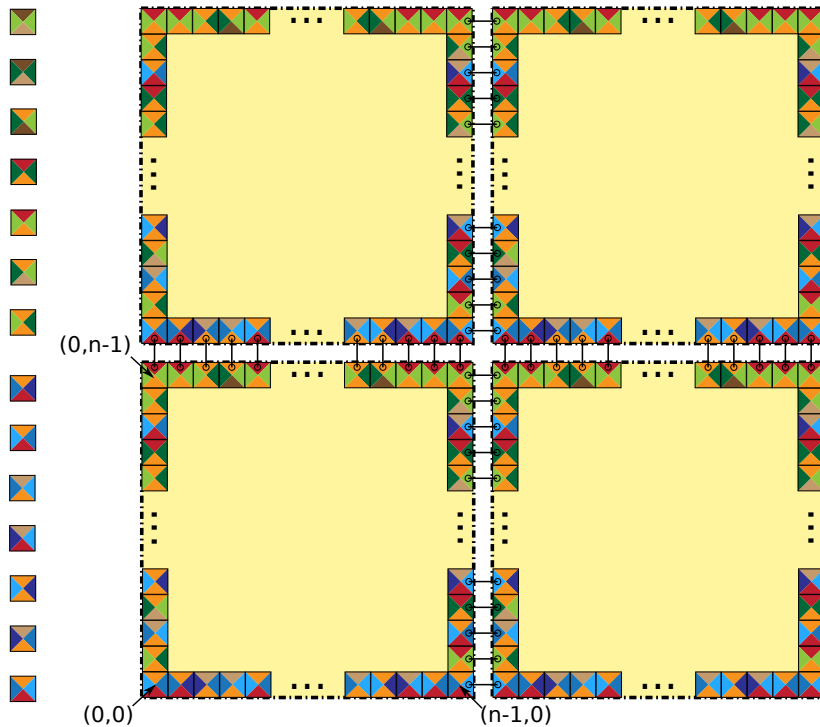


Figure 2.4: A repeatable square.

Lemma 2.3.4. *Let A denote a solvable tileset. There exists a periodic tiling by A if and only if there exists a repeatable square for A .*

Proof. If a repeatable square exists, we can immediately build a periodic tiling by just repeating the square over and over all the plane.

Conversely, suppose that a periodic tiling T exists. Then, by Exercise 2.3.2, then there exists a tiling T' with two periodicity vectors which are not colinear. Call them $\vec{v}_1 = (x_1, y_1)$ and $\vec{v}_2 = (x_2, y_2)$. Observe that $x_2\vec{v}_1 - x_1\vec{v}_2$ is a vertical vector, while $y_2\vec{v}_1 - y_1\vec{v}_2$ is an horizontal vector. Since T' has both a horizontal periodicity vector $(x_1x_2, 0)$ and a vertical periodicity vector $(0, y_1y_2)$, it contains a repeatable square of size $x_1x_2 \times y_1y_2$. \square

Theorem 2.2.4 and Lemma 2.3.4 together suggest the following algorithm to solve the domino problem.

Algorithm 2.3.5 (Wang, 1961 [9]). Input: a tileset A . Output: whether A is solvable or not.

1. $n \leftarrow 1$
2. Try all possible tilings of a square of size $n \times n$ by A ;
3. If there is no valid tiling, return **false**;
4. Else, if there is a repeatable square among valid tilings, return **true**;
5. Else, do $n \leftarrow n + 1$ and go back to Step 2.

The proof of correctness of this algorithm relies on the following idea.

Conjecture 2.3.6 (Wang, 1961 [9]). *A tileset is solvable if and only if it has a periodic tiling.*

Wang’s conjecture implies that the algorithm is correct. Indeed, if the given tileset A is solvable, then by Conjecture 2.3.6 and Lemma 2.3.4, A has a repeatable square. Since the algorithm tries all possible squares in search for a repeatable square, it will necessarily find it and conclude that A is solvable. On the other hand, when A is unsolvable, the algorithm returns `false`. If it did not, then it would mean that any square (of arbitrary size) is by tilable by A , so Theorem 2.2.4 would imply that A is solvable: a contradiction.

It only remains to prove Wang’s conjecture. Unfortunately, it turns out that

Theorem 2.3.7 (Berger, 1964 [2]). *The Domino problem is undecidable.*

which means that there is no algorithm that solves the Domino problem. This was shown in 1966 by a student of Wang, Robert Berger [2]. The proof works by encoding Turing machines in tilesets and translating the Domino problem to the Halting Problem (Section 1.8), which is undecidable.

2.3.8. Since the Domino problem is undecidable, Algorithm 2.3.5 must be incorrect. However, we are sure that if this algorithm returns `true`, then the given tileset was solvable (because the algorithm found a repeatable square); conversely, if the algorithm returns `false`, then the given tileset was unsolvable (because the algorithm found a size of square n which cannot be tiled). The only remaining possibility for this algorithm to be incorrect, is that it does not return anything at all. In other terms, there are some tilesets A on which the algorithm loops infinitely, without ever returning.

2.3.9. Consider a tileset A that makes Algorithm 2.3.5 loop. From the remarks above, A solvable, but it has no periodic tiling of the plane. A solvable tileset this property is called an *aperiodic tileset*. Conjecture 2.3.6 stated that there is no aperiodic tileset, and therefore

Conjecture 2.3.6 is false.

Berger provided such an aperiodic tileset in his work, as a part of his translation from the Domino problem to the Halting problem. Since then, many aperiodic tilesets were discovered, with fewer and fewer tiles. Understanding aperiodic tilesets is still an active area of research today [4].

2.4 Encoding Turing machines in tiles

Before proving Theorem 2.3.7, which is quite difficult, we prove a simpler result. Namely, we consider the *Domino Problem with fixed origin tile*: we are given a pair (A, a) where A is a tileset and a an element of A , and we are asked whether there exists a tiling of the plane T by A such that $T(0, 0) = a$. In other terms, we have the same problem as Domino, except that one specific tile have been placed in advance at the origin. We prove the following theorem.

Theorem 2.4.1 (Wang et al, 1963 [8]). *The Domino problem with fixed origin is undecidable.*

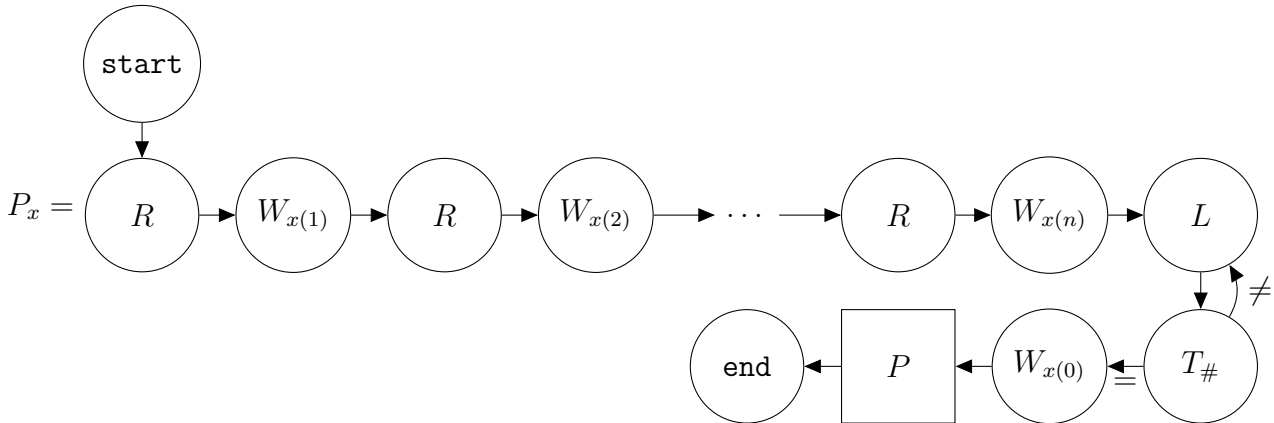
The plan of the proof is as follows: we start with a Turing machine P and an input x for P , and we design a tileset T such that each tiling (with a specific tile at the origin) by T must “draw” a space-time diagram of $P(x)$. If $P(x)$ does not terminate, then its space-time diagram is infinite, so T can tile infinitely many lines and thus it is solvable (with fixed origin). On the other hand, if $P(x)$ terminates, then its space-time diagram is finite, so T can tile only finitely many lines and thus it is not solvable (with fixed origin).

Moreover, **the translation of P and x into a tileset T will be an algorithm**. Let us call this algorithm f , so we have $f(P, x) = T$. Now, suppose we have an algorithm d to solve the Domino

problem with fixed origin. Then by the remarks above, for any Turing machine P and any input x , we have $d(f(P, x)) = 1$ if $P(x)$ terminates, and 0 otherwise. In other terms, $d(f(P, x))$ is an algorithm for the Halting problem; such an algorithm does not exist, so we have a contradiction. Consequently, there is no algorithm d that solves the Domino problem with fixed origin.

2.4.2. For the remainder of this section, fix a Turing machine P and an input x for P .

2.4.3. If Q is any Turing machine, write $Q(\#)$ for the computation of $\#$ on the input full of symbols $\#$. Instead of making a tiling for the computation $P(x)$, it will be more convenient to make a tiling for the computation $P'(\#)$, for some other machine P' . Let $x = x(0) \dots x(n)\#$ and define P_x to be the machine that first writes x on the memory, then moves the head back at the beginning, and runs P :

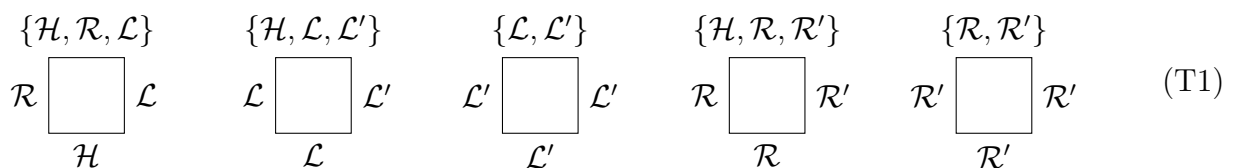


Lemma 2.4.4. *The computation $P(x)$ terminates if and only if $P_x(\#)$ does. If these computations terminate, they both have the same output. Moreover, the transformation of (P, x) into P_x is computable.*

A slightly more general version of this lemma is known as the s_n^m theorem and was initially proved by Stephen Kleene. It is a fundamental result of computability theory. For this lecture, however, we will stick with Lemma 2.4.4.

2.4.5. Now it remains to build a tiling T such that, any tiling by T (with a specific tile at the origin) somehow “draws” the space-time diagram of $P_x(\#)$. (We will make the meaning of “draw” precise later, in §2.4.10 and Lemma 2.4.11.) To make this whole construction more readable, we will build several tilesets, $(T1)$, $(T2)$, $(T3)$, etc., each slightly more complicated than the previous one. The last one in this sequence will be the desired T . We will not explicitly give Python code for that algorithm, but our explanations should make it easy to write it.

2.4.6. For $(T1)$, the set of colors is $\mathcal{C} = \{\mathcal{H}, \mathcal{L}, \mathcal{L}', \mathcal{R}, \mathcal{R}'\}$ (their meanings will be explained soon) and the set of tiles is:



The notation $\{\mathcal{H}, \mathcal{R}, \mathcal{L}\}$ for a color means that we make three copies of the tile: one with the color \mathcal{H} , one with \mathcal{R} , one with \mathcal{L} . Thus this tileset has 13 tiles in total. The meaning of the colors is as follows:

- \mathcal{H} means: the head is here;
- \mathcal{L} means: the head is immediately on my left;
- \mathcal{L}' means: the head is on my left, but not just next to me;
- \mathcal{R} means: the head is immediately on my right;
- \mathcal{R}' means: the head is on my right, but not just next to me.

Figure 2.5 shows a line tiled by (T1). What (T1) does is to guarantee that there is no more than one head on each line. There are three kinds of tiles: the head, tiles on the left of the head, and tiles on the right of the head. Besides, the two tiles that touch the head are different from all the others; in a sense, they *know* that they touch the head. This extra information will become handy later; it is the reason why we have to distinguish between \mathcal{R} and \mathcal{R}' (also between \mathcal{L} and \mathcal{L}').



Figure 2.5: A line tiled by (T1). The symbol $*$ means “any color”.

2.4.7. For (T2), the set of colors is $\mathcal{C} \times \mathbb{I}$, a cartesian product, where \mathcal{C} is the set of colors of (T1) and \mathbb{I} is the set of values that can appear in a memory cell of P (the Turing machine we are trying to encode into tiles). Then, for each t in (T1), each x in \mathbb{I} and each y in \mathbb{I} , make the following tile:

$$\begin{array}{ccc}
 \begin{array}{c} (-, y) \\ (-, x) \begin{array}{|c|} \hline \square \\ \hline \end{array} (-, x) \\ (H, x) \end{array} & \text{if } \text{south}(t) = H; & \begin{array}{c} (-, y) \\ (-, x) \begin{array}{|c|} \hline \square \\ \hline \end{array} (-, x) \\ (-, y) \end{array} & \text{otherwise} & (T2)
 \end{array}$$

where “—” means “the same color as in the tile t ”. We now have $13 \times |\mathbb{I}|^2$ tiles, where $|\mathbb{I}|$ is the number of elements of \mathbb{I} .

This new tileset (T2) inherits the features of (T1): on each line, at most one tile is the “head”, and all other tiles “know” if they are to the left or to the right of the head. Moreover, the two tiles that touch the head “know” that they touch the head. But (T2) does something more: each tile now has an arbitrary element of \mathbb{I} on its bottom color, and this element is propagated as-is on its top color. The only exception is the head tile: it can have any element of \mathbb{I} on its top side, no matter what its bottom element is. Finally, the tiles also propagate one element of \mathbb{I} on their left and right sides; because of the color constraints we designed, this element is always the one found on the bottom side of the head.

Figure 2.6 shows a line tiled by (T2).

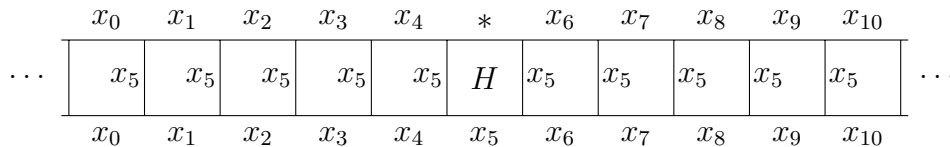


Figure 2.6: A line tiled by (T2). The x_i and $*$ are elements of \mathbb{I} . The first components of the colors (\mathcal{H} , \mathcal{R} , \mathcal{L} , etc.) are not shown.

2.4.8. For (T3), the set of colors is $\mathcal{C} \times \mathbb{I} \times P$, where \mathcal{C} is the set of colors of (T1), \mathbb{I} is the set of values that might appear in the memory cells of P , and finally P itself is the set of nodes of the

Turing machine we're encoding into tiles. For each tile t in (T2) and for each node v in P , add the following tile into (T3):

$$\begin{array}{ccc} & (-, -, v') & \\ (-, -, v) & \square & (-, -, v) \\ & (-, -, v) & \end{array} \quad (\text{T3})$$

where, once again, the “—” refer to the corresponding colors in the tile t from (T2). If the label of v is anything except T_i , then there is only one edge going out from v , and we call v' the target of this edge. If the label of v is T_i , however, there are two edges going out from v . Fortunately, each tile “knows” what is the contents of the memory cell under the head, because this information is propagated in the second component of the left and right sides of each tile of (T2). Thus we have all the required information to uniquely determine v' .

Tiling a line with (T3) is pretty similar to tiling a line with (T2), except that the whole line “knows” one node in the graph of P , and guarantees that the next line has the next node in the execution of the program.

2.4.9. Now comes the interesting part. We build (T4) from (T3) by *removing* some tiles.

First step. Remove all tiles matching the following pattern:

$$\begin{array}{ccc} & (*, y, v') & \\ (\mathcal{R}, x, v) & \square & (\mathcal{L}, x, v) \\ & (\mathcal{H}, x, v) & \end{array} \quad \text{where } v \in T, x \in \mathbb{I} \text{ and } \begin{cases} y \neq i & \text{if } v \text{ has label } W_i, \\ y \neq x & \text{otherwise.} \end{cases}$$

Indeed, remember that, in (T2) (and thus in (T3)), the head is free to rewrite its memory cell with whatever value (up to the tiler's choice). Removing those tiles guarantees that the head does not change the value of its cell, unless the label of the current node is W_i ; in this case, it guarantees that the newly written value is i .

Second step. For each v other than L and R , remove all the tiles matching the following pattern:

$$\begin{array}{ccc} & (y, *, *) & \\ (*, *, v) & \square & (*, *, v) \\ & (x, *, v) & \end{array} \quad \text{whenever } x \neq y.$$

This guarantees that, if the label of v is not R nor L , then the head doesn't move.

Third step. For $v = R$, then remove all tiles that match one of the following patterns:

$$\begin{array}{ccc}
 \begin{array}{c} (x, *, *) \\ (\mathcal{R}, *, v) \square (\mathcal{L}, *, v) \\ (\mathcal{H}, *, v) \end{array} & \text{for } x \neq \mathcal{R}; & \begin{array}{c} (x, *, *) \\ (\mathcal{L}, *, v) \square (\mathcal{L}', *, v) \\ (\mathcal{L}, *, v) \end{array} & \text{for } x \neq \mathcal{H}; \\
 \\
 \begin{array}{c} (x, *, *) \\ (\mathcal{R}', *, v) \square (\mathcal{R}, *, v) \\ (\mathcal{R}, *, v) \end{array} & \text{for } x \neq \mathcal{R}' & &
 \end{array}$$

These removals guarantee that:

- the tile above \mathcal{H} will have \mathcal{R} on the next line (so the head becomes “the head is to my right”);
- the tile above \mathcal{L} will have \mathcal{H} on the next line (so it becomes the new head);
- the tile above \mathcal{R} will have \mathcal{R}' on the next line (so it is further from the head) on the next line.

The construction of (T3) then guarantees that the rest of the next line is going to be consistent. Figure 2.7 shows two lines tiled by (T4), supposing $v = R$; we only show the parts of the colors related to the position of the head.

\mathcal{R}'	\mathcal{R}'	\mathcal{R}'	\mathcal{R}'	\mathcal{R}'	\mathcal{R}	\mathcal{H}	\mathcal{L}	\mathcal{L}'	\mathcal{L}'	\mathcal{L}'
\mathcal{R}'	\mathcal{R}'	\mathcal{R}'	\mathcal{R}'	$\mathcal{R}\mathcal{R}$	$\mathcal{L}\mathcal{L}$	\mathcal{L}'	\mathcal{L}'	\mathcal{L}'	\mathcal{L}'	\mathcal{L}'
\mathcal{R}'	\mathcal{R}'	\mathcal{R}'	\mathcal{R}'	\mathcal{R}	\mathcal{H}	\mathcal{L}	\mathcal{L}'	\mathcal{L}'	\mathcal{L}'	\mathcal{L}'

Figure 2.7: Two consecutive lines tiled by (T4) for $v = R$. Only the $\mathcal{H}/\mathcal{R}/\mathcal{L}$ part of the colors is shown.

Fourth step. Finally for $v = L^2$, remove the tiles matching the following pattern:

$$\begin{array}{ccc}
 \begin{array}{c} (x, *, *) \\ (\mathcal{R}, *, v) \square (\mathcal{L}, *, v) \\ (\mathcal{H}, *, v) \end{array} & \text{for } x \neq \mathcal{L}; & \begin{array}{c} (x, *, *) \\ (\mathcal{R}', *, v) \square (\mathcal{R}, *, v) \\ (\mathcal{R}, *, v) \end{array} & \text{for } x \neq \mathcal{H}; \\
 \\
 \begin{array}{c} (x, *, *) \\ (\mathcal{L}, *, v) \square (\mathcal{L}', *, v) \\ (\mathcal{L}, *, v) \end{array} & \text{for } x \neq \mathcal{L}' & &
 \end{array}$$

These removals guarantee that:

- the tile above \mathcal{H} will have \mathcal{L} on the next line (so the head becomes “the head is to my left”);

²Experts will note that this has nothing to do with the $V = L$ axiom of set theory

- the tile above \mathcal{R} will have \mathcal{H} on the next line (so it becomes the new head);
- the tile above \mathcal{L} will have \mathcal{L}' on the next line (so it is further from the head) on the next line.

The construction of (T3) then guarantees that the rest of the next line is going to be consistent. Figure 2.8 shows two lines tiled by (T4), supposing $v = L$; we only show the parts of the colors related to the position of the head.

\mathcal{R}'	\mathcal{R}'	\mathcal{R}'	\mathcal{R}	\mathcal{H}	\mathcal{L}	\mathcal{L}'	\mathcal{L}'	\mathcal{L}'	\mathcal{L}'	\mathcal{L}'
\mathcal{R}'	\mathcal{R}'	\mathcal{R}'	\mathcal{R}'	$\mathcal{R}\mathcal{R}$	$\mathcal{L}\mathcal{L}$	\mathcal{L}'	\mathcal{L}'	\mathcal{L}'	\mathcal{L}'	\mathcal{L}'
\mathcal{R}'	\mathcal{R}'	\mathcal{R}'	\mathcal{R}'	\mathcal{R}	\mathcal{H}	\mathcal{L}	\mathcal{L}'	\mathcal{L}'	\mathcal{L}'	\mathcal{L}'

Figure 2.8: Two consecutive lines tiled by (T4) for $v = L$. Only the $\mathcal{H}/\mathcal{R}/\mathcal{L}$ part of the colors is shown.

2.4.10. Let $c = (P, v, M, h)$ denote a configuration and ℓ a line tiled by (T4). We write ℓ_n for the n^{th} tile in ℓ and $\text{south}(\ell_n) = (p_n, m_n, v_n)$. Then we say that ℓ encodes c if and only if:

- $v = v_n$ for each integer n ;
- $M(n) = m_n$ for each integer n ;
- $p_h = \mathcal{H}$.

Lemma 2.4.11. *If a line ℓ tiled by (T4) has at least one cell with \mathcal{H} on its south color, then ℓ encodes a configuration c .*

Moreover, if ℓ, ℓ' are two lines tiled by (T4), both representing configurations, such that ℓ' can be put on top of ℓ to form a valid tiling, then the configuration encoded by ℓ' is the successor of the configuration encoded by ℓ .

Proof. The first statement is immediate: the position of the head is the position of the (unique) cell having \mathcal{H} on its south color; the current node v is the node appearing on all south colors; and the contents of the memory are also read on the south colors.

The second statement is true by construction of (T4), so it results from the remarks made in §2.4.6, §2.4.7, §2.4.8 and §2.4.9. □

Exercise 2.4.12. Why is the condition “has at least one cell with \mathcal{H} on its south color” needed in this lemma?

Exercise 2.4.13. Let us modify the definition of a Turing machine and of a configuration (§1.2.4) so that the memory is *biinfinite*, i.e., a function $\mathbb{Z} \rightarrow \mathbb{I}$ instead of $\mathbb{N} \rightarrow \mathbb{I}$. Prove that a function is computable with an \mathbb{N} -memory-Turing machine if and only if it is computable with a \mathbb{Z} -memory-Turing machine. Moreover, show that the transformation of an \mathbb{N} -memory-Turing machine to the “equivalent” \mathbb{Z} -memory-Turing machine is computable.

2.4.14. Now we make (T5) by *removing* tiles from (T4). Let s denote the vertex labeled **start** in P . Then, make T by removing each tile in (T4) that matches the following pattern:



Theorem 2.4.15. *The computation $P_x(\#)$ (and therefore $P(x)$) does **not** terminate if and only if there exists a tiling T of the upper half of the plane $(\mathbb{Z} \times \mathbb{N})$ such that:*

$$T(0,0) = \begin{array}{ccc} & (H, \#, s') & \\ (R, \#, s) & \square & (L, \#, s) \\ & (H, \#, s) & \end{array}$$

where s is the node of P labeled with **start**, and s' is its successor (the next node to execute).

Proof. The definition of $(T5)$ is the subset of $(T4)$ such that the input given to the machine consists only of $\#$ symbols. By Lemma 2.4.11, the placement of the tile in $(0,0)$ guarantees that the line 0 of any tiling by T encodes a configuration. By Lemma 2.4.11 again, the line number $n+1$ encodes the successor of the configuration represented by line n . By definition of a computation (§1.2.4), there are infinitely many lines if and only if the machine does not terminate. \square

2.4.16. We still have one problem to overcome before claiming a proof of Theorem 2.4.1. Namely, Theorem 2.4.15 only deals with tilings of *half* of the plane, but we need tilings of the full plane. The Compactness theorem (§2.2.4) does not help here, because of the fixed tile at the origin.

Fortunately, there is an easy solution: make a tileset $(T5')$, where each tile is just the vertical symmetric of a tile of $(T5)$. Moreover, in each tile of $(T5')$, replace each color (t, v, s) with the color (t, \bar{v}, s) , where \bar{v} is a copy of v (but *different from* v). The only exception is that $\bar{s} = s$, where s is the node of P labeled with **start**. See Figure 2.9 for an illustration of this transformation.

$$\left\{ \begin{array}{ccc} (t_4, x', v') & & (t_1, x, \bar{v}) \\ (t_2, y, v) & \square & (t_3, y, v) \end{array} \right\} \bar{t} = \left\{ \begin{array}{ccc} (t_2, y, \bar{v}) & \square & (t_3, y, \bar{v}) \\ (t_1, x, v) & & (t_4, x', \bar{v}') \end{array} \right.$$

Figure 2.9: Transformation of a tile of $(T5)$ into a tile of $(T5')$. We have $v = \bar{v}$ if v is labeled with **start**, and $v \neq \bar{v}$ otherwise.

By the design of $(T5')$, it is impossible to put one tile of $(T5)$ next to a tile of $(T5')$ (since the third component of tiles are disjoint between $(T5)$ and $(T5')$) *except* when the two tiles have the **start** node. Moreover, $(T5)$ makes the computation moving up, while $(T5')$ moves down. Consequently, any tiling by $(T6) = (T5) \cup (T5')$ must follow the schema of Figure 2.10.

Lemma 2.4.17. *The tileset $(T5)$ can tile $\mathbb{N} \times \mathbb{Z}$ (with the origin constraint of Theorem 2.4.15) if and only if $(T6)$ can tile $\mathbb{Z} \times \mathbb{Z}$ (with the same origin constraint).*

Consequently,

Conclusion. The tileset $(T6)$ is solvable (with the origin constraint from Theorem 2.4.15) if and only if $P(x)$ does not terminate. Moreover, the whole procedure transforming (P, x) into $(T6)$ is computable. Since the negation of a boolean is obviously computable, we have a program that translates the Halting problem into the Domino problem with fixed origin. If the latter were decidable, then by our translation the former would be, and we would have a contradiction. Therefore, the Domino problem with fixed origin is undecidable.

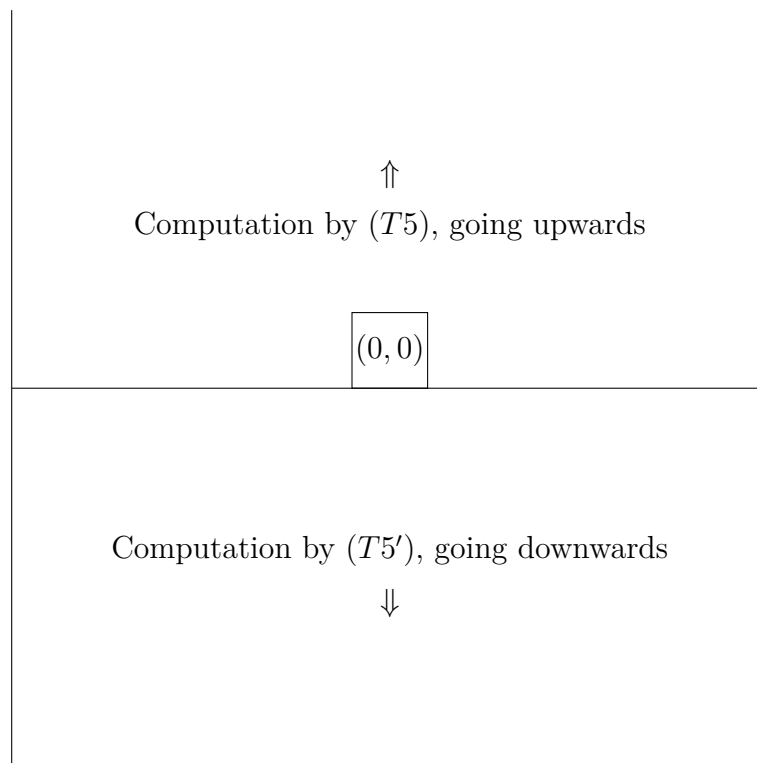



Figure 2.10: This symmetric tiling does the computation of $P(\#)$ twice: once upwards and once downwards.

2.e Exercises for Lecture 2

Here are all the exercises from the lecture notes, reorganized and renumbered to make an exercise sheet. There are also a few new questions, to keep things interesting. Solve only what you like.

Exercise 2.e.1. (Lineland) We consider the one-dimensional version of the Domino problem: a *Wang bar* is a unit segment with colored vertices, and a finite set of Wang bars is a *barset*. Given a barset B , a *one-dimensional tiling* of a set of points $S \subseteq \mathbb{Z}$ by B is a function $S \rightarrow B$. A tiling of the whole \mathbb{Z} is called a *tiling of the line*. Of course Wang bars also have color constraints: a tiling is *valid* if for each pair of neighbouring bars, the adjacent vertices have the same color. For example, these two bars match: 

Here is an example of a valid tiling of the line:




A barset is *solvable* if it has at least one valid tiling of the line.

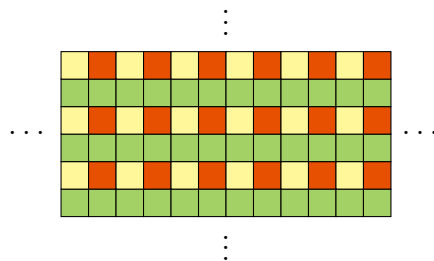
A tiling of the line T is *periodic* if there exists a nonzero integer p , called the *period*, such that $T(x + p) = T(x)$ for all x in \mathbb{Z} .

- (a) Prove the following statement (called the one-dimensional Compactness theorem). Let A denote a barset; if there is a valid tiling of the set $\{-n, \dots, +n\}$ by A for every integer n , then there is a valid tiling of \mathbb{Z} by A .
- (b) Show that Wang’s conjecture holds for one-dimensional tilings: if a barset is solvable, then it has at least one periodic tiling of the line.
- (c) Give an algorithm solving the one-dimensional Domino Problem.

Back to Flatland... Only Exercise 2.e.1 talks about Wang bars. All other exercises talk about two-dimensional tiles.

Exercise 2.e.2 (). Let A denote a solvable tileset. Prove that if A has a valid tiling T with one period \vec{v} , then it has a tiling (maybe different from T) with two periods \vec{v}_1 and \vec{v}_2 which are not colinear (proportional).

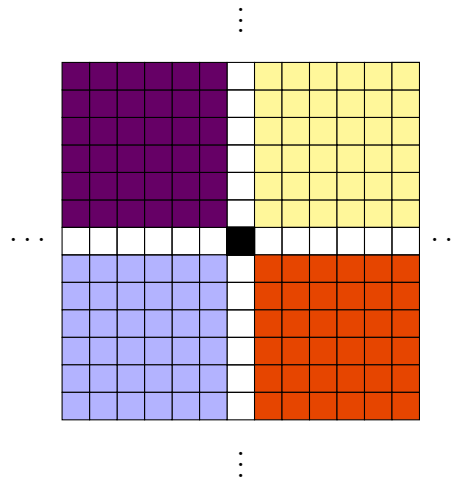
Exercise 2.e.3. Design a set of 3 Wang tiles A such that all tilings of the plane by A , up to translation, look like this:



where squares of different colors denote Wang tiles of different types.

Exercise 2.e.4.

(a) Design a set of 9 Wang tiles A such that one of its tilings of the plane looks like this:



where squares of different colors denote Wang tiles of different types. (One color can correspond to several tiles, but one tile corresponds to only one color on the picture).

(If necessary, start by designing a tiling with any number of tiles inside, and then try to reduce it.)

(b) Explain why the tileset from your solution of (a) also can assemble in tilings where all the tiles are “orange” (respectively “blue”, “violet”, “yellow”).

Exercise 2.e.5. Find an algorithm that, given a **directed** graph G , generates a set of Wang tiles A such that G has a cycle if and only if A is solvable. The algorithm should work with a constant memory. **The algorithm itself should not answer whether the graph has a cycle or not**, it only should translate to a tileset.

Exercise 2.e.6. Why is the condition “has at least one cell with \mathcal{H} on its south color” needed in Lemma 2.4.11 of the lecture notes?

Exercise 2.e.7 (\mathbb{Z}). Let us modify the definition of a Turing machine and of a configuration (§1.2.4) so that the memory is *biinfinite*, i.e., a function $\mathbb{Z} \rightarrow \mathbb{I}$ instead of $\mathbb{N} \rightarrow \mathbb{I}$. Prove that a function is computable with an \mathbb{N} -Turing machine if and only if it is computable with a \mathbb{Z} -Turing machine.

Exercise 2.e.8 (\mathbb{Z}). Write a Python program that converts a Turing machine into a set of tiles, as described in Section 2.4.

Contacts

- Daria Pchelina (dpchelina@clipper.ens.fr)
- Guilhem Gamard (guilhem.gamard@normale.fr)

Lecture 3

The Robinson tileset

3.1 Introduction

3.1.1. In the last lecture, we proved that the Domino Problem *with fixed origin* is undecidable. This is a good step, but we still have to get rid of the constraint on the origin to prove that the original Domino Problem is undecidable.

To do so, let's discuss the Robinson tileset, that we will denote by R . It was initially published by Raphael Robinson in 1971 [6]. The tileset R is aperiodic, meaning that it is solvable (it can tile the plane), but no valid tiling of the plane by R is periodic. Moreover, it contains relatively few tiles, which makes its description easy.

Once R is well-understood, we show how to exploit it to fully prove that the Domino Problem is undecidable. (That proof can be found in the paper by Robinson, [6].)

3.1.2. A representation of R is displayed on Figure 3.1, but it is not exactly a set of Wang tiles.

- First, by contrast with the tilesets we have seen so far, Robinson tiles *can be rotated*, so we have to make 4 copies of each Robinson tile (one copy per possible orientation).
- Second, Robinson tiles do not have colors on their borders, but rather colored arrows coming in and out. The rule is that two tiles can be neighbours if and only if each arrow going out of the first tile is prolonged by an arrow coming into the second tile. The prolonging arrow must have the same color as the initial one, and the endpoints should exactly match. Figure 3.2 shows an illustration of valid pairs of tiles.
- Finally, observe that one tile (four if we count rotations) has small yellow dots in its corners. The rule is that whenever four tiles share one common corner, *exactly one* of them should have those yellow dots. Figure 3.3 illustrates that rule.

Formally, that last rule is not a “color constraint” in the sense of Wang tiles. However, it is not hard to design a set of Wang tiles that enforces that rule (see Exercise 3.e.2). A piece of a Robinson tiling satisfying those constraints is shown in Figure 3.4.

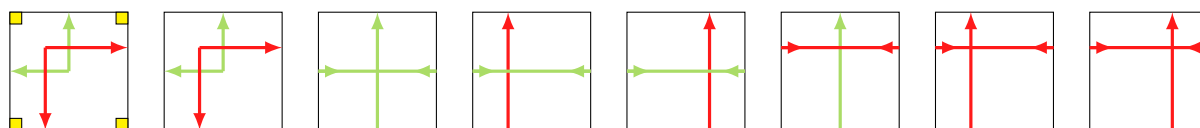


Figure 3.1: Robinson tiles, up to rotation.

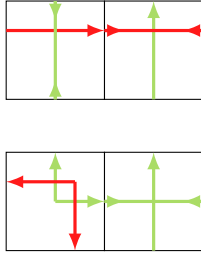


Figure 3.2: Examples of adjacent tiles.

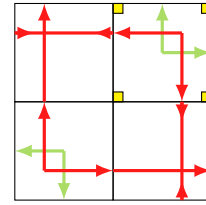


Figure 3.3: Four tiles sharing a corner.

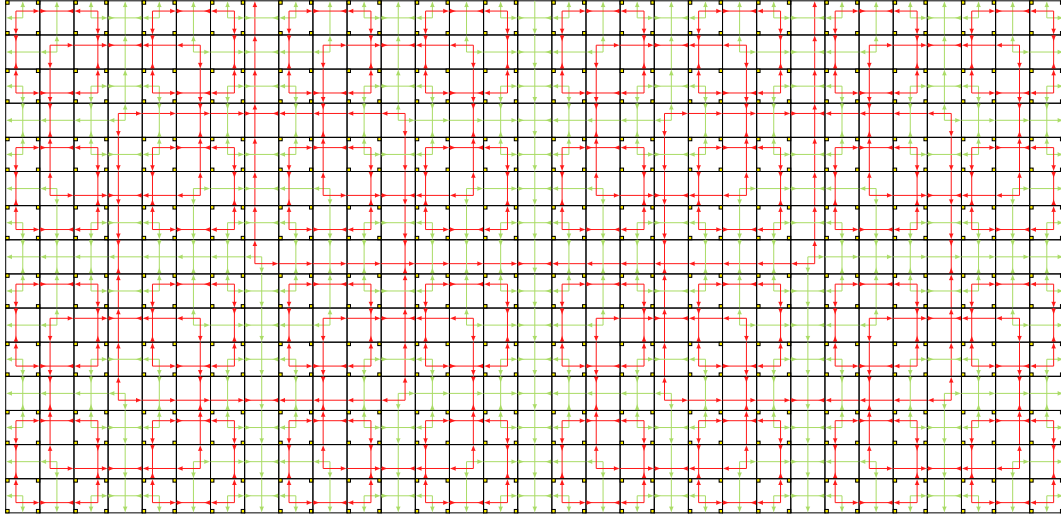


Figure 3.4: A piece of a Robinson tiling.

3.2 Tiling in the large: macrotiles

3.2.1. Let A and B denote two tilesets. A *projection* of A on B is simply a function $f : A \rightarrow B$ that preserves color constraints. In other terms, if a_1 and a_2 are allowed to be next to each other, then $f(a_1)$ and $f(a_2)$ also are. If T is a tiling by A , then $f(T)$ denotes the tiling by B obtained by applying f to each tile of T . The definition of a projection implies that if T is valid, then $f(T)$ is also valid.

A projection $f : A \rightarrow B$ is:

- *injective* if two different tiles are always given two different images: $\forall a_1, a_2 \in A, a_1 \neq a_2 \implies f(a_1) \neq f(a_2)$;
- *surjective* if any tile of B has at least one preimage by f : $\forall b \in B, \exists a \in A, f(a) = b$;
- *bijective* if it is both injective and surjective.

If $f : A \rightarrow B$ is an injective (respectively surjective, bijective) projection, then its extension to tilings $f : A^{\mathbb{Z}^2} \rightarrow B^{\mathbb{Z}^2}$ is also injective (respectively surjective, bijective). Therefore, for all our work, it does not hurt to treat a projection between tilesets and its extension to tilings as the same object.

Exercise 3.2.2. Design a set of Wang tiles A along with a surjective projection from A to R .

3.2.3. Let A denote an arbitrary tileset. A *macrotiler* over A is just a valid tiling of a square (say, of size $n \times n$) by A . If $M = \{m_1, \dots, m_k\}$ is a set of macrotilers over A , all of the same size, then we can consider M as a tileset on its own. Indeed, we can check as usual if two macrotilers are allowed

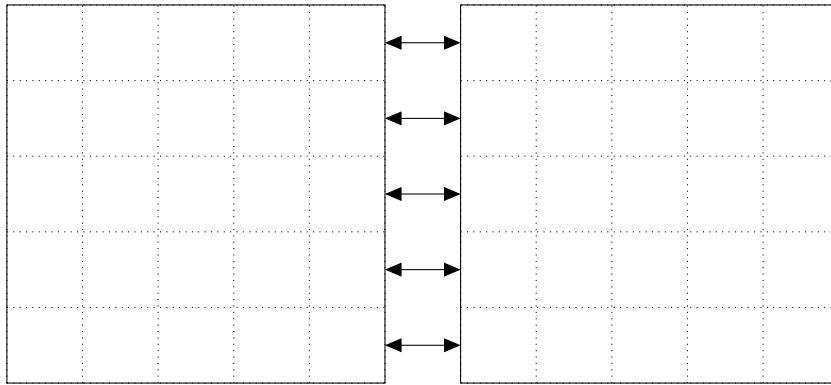


Figure 3.5: Checking if two macrotiles may be neighbours.

to be neighbours: all the colors along the edges in contact should match (see Figure 3.5). In a sense, the colors on the borders of macrotiles are cartesian products of colors on the base tiles.

3.2.4. Suppose we have a tileset A and a set M of macrotiles over A , all of the same size. We say that M is a *macrotiler* over A . Let f denote a bijective projection of A on M . This projection maps each tile a in A to a macrotile m in M , while preserving the validity of the color constraints. But since m is just a square tiled by A , we can again apply f to it: $f(m)$ is a macromacrotiler over A . Consider the set of all macromacrotilers:

$$f(M) = \{f(m) \mid m \in M\}.$$

Each macromacrotiler is still a (very large!) square tiled by A , and the colors constraints are again preserved. Thus nothing stops us from defining $f^2(M)$, and $f^3(M)$, and so on. The satisfaction of color constraints is preserved at every step. Figure 3.6 gives an illustration of this process.

Lemma 3.2.5. *Let A be a tileset and M be a macrotiler over A . If there exists a bijective projection of A to M , then A is solvable.*

Proof. Call f the bijective projection and k the integer such that the elements of M have size $k \times k$. Take a any tile in A , and observe that the macrotiler $f^n(a)$ is well defined for every n . In particular, $f^n(a)$ is a valid tiling of a $k^n \times k^n$ square. Then the Compactness Theorem (§2.2.4) implies that A is solvable. \square

3.3 Robinson's hierarchy of macrotilers

3.3.1. Now we prove that R is solvable. We will define four sequences of macrotilers, called $(a_n)_{n \in \mathbb{N}}$, $(b_n)_{n \in \mathbb{N}}$, $(c_n)_{n \in \mathbb{N}}$, and $(d_n)_{n \in \mathbb{N}}$. First, a_0 , b_0 , c_0 and d_0 are just the four tiles of R with yellow dots in the corners. Recall that those four tiles are identical up to rotation; Figure 3.7 shows how we assign the names to them. By the way, we define \bar{a}_0 , \bar{b}_0 , \bar{c}_0 and \bar{d}_0 to be the versions of a_0 , b_0 , c_0 and d_0 without the yellow dots in the corners (those tiles also belong to R , as you can check on Figure 3.1); we will use them a little later.

In what follows, b_n , c_n and d_n will always be rotations of a_n , with the same orientations as in Figure 3.7. Thus it remains to define a_n for $n \geq 2$.

3.3.2. Let n denote some integer and suppose that a_n , b_n , c_n and d_n are defined. Then we define α_n (that's a Greek α !) as follows: arrange one copy of a_n , one copy of b_n , one copy of c_n and one

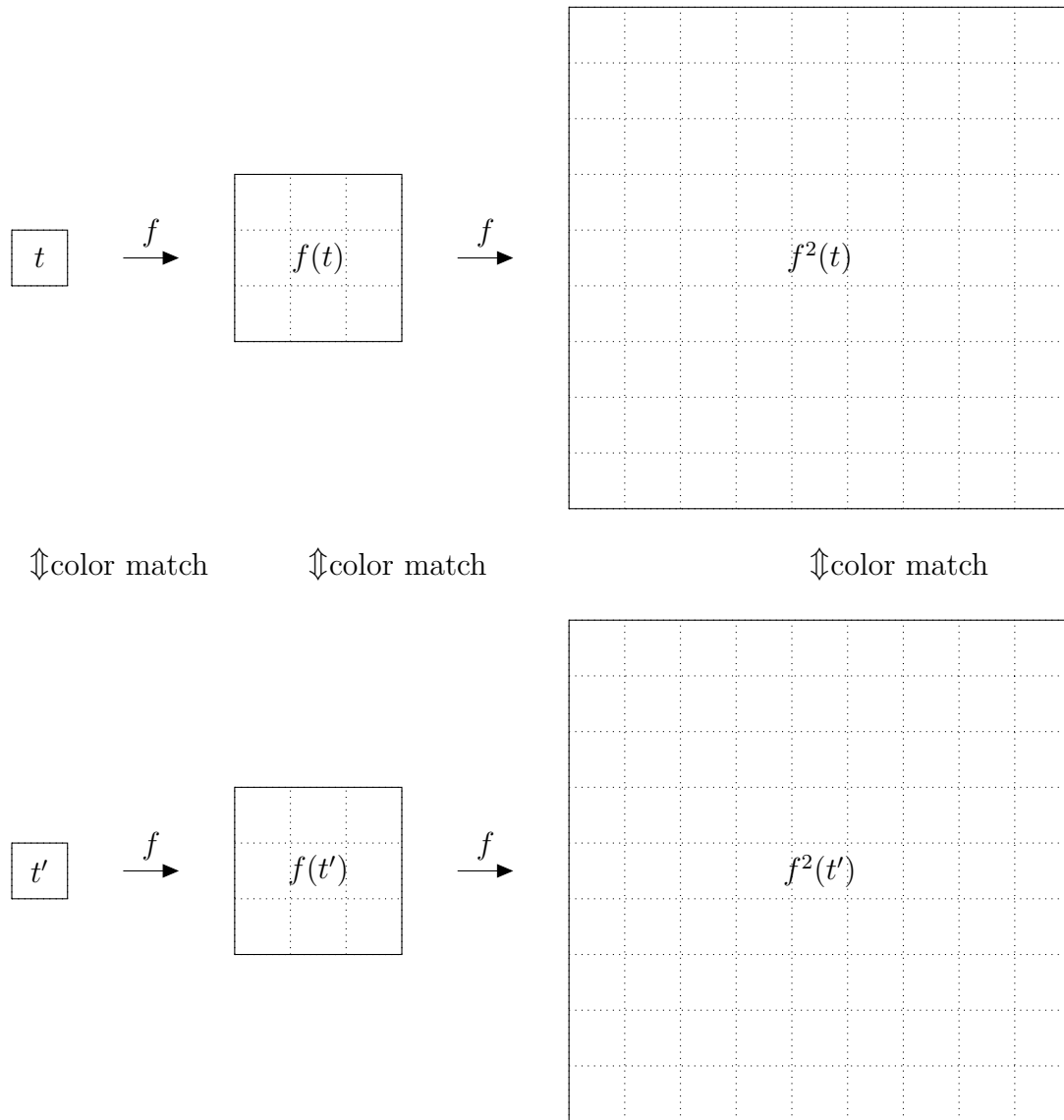


Figure 3.6: Iterating a projection of tiles on macrotiles

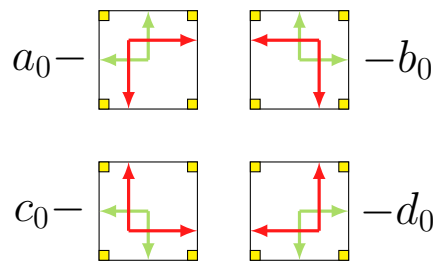


Figure 3.7: Robinson's macrotiles a_0 , b_0 , c_0 and d_0 .

copy of d_n in a square, leaving a gap of one cell between them, so that the gap has the shape of a cross. Then, add one copy of \bar{a}_0 in the center of that cross. (See Figure 3.8.) The pattern α_n is not quite a macrotile, because a macrotile has to have the shape of a square. Fortunately, there exists an unique way to complete α_n into a valid tiled square — we will prove this shortly. Then, a_n is just that unique completion α_n into a valid square. The macrotiles b_n , c_n and d_n are defined to be rotations of a_n ; since all tiles in R exist in all their possible rotations, they are also valid tilings of squares.

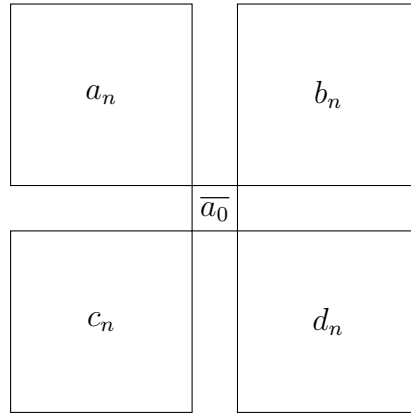


Figure 3.8: The pattern α_n : a partial macrotile for a_n . Its completion into a valid square is unique.

Lemma 3.3.3. *For each $n \geq 1$:*

- (i) *there is an unique completion of α_n into a valid square, which we call a_{n+1} ;*
- (ii) *a_n is of size $(2^{n+1} - 1) \times (2^{n+1} - 1)$;*
- (iii) *a_n has one red arrow going out on the middle of its right side and on the middle of its bottom side;*
- (iv) *everywhere else on the borders of a_n , green arrows are going out;*
- (v) *the four tiles in the corners of a_n have yellow dots.*

Moreover, b_n , c_n , and d_n also have properties (ii)–(v), except that the directions mentioned in (iii) should be adapted to the rotation that defined b_n , c_n and d_n .

Proof. Initialization. The base case, a_0 , happens to be a single tile, so all the statements of the lemma except the first one are tautological.

- (i) Figure 3.9 shows α_0 ; a direct observation (try all possibilities) shows that there is only one possible completion. Figure 3.10 shows the completed macrotile a_1 (on the right), as well as an alternative intuition about how this macrotile was constructed. Let b_1 , c_1 and d_1 be the rotated versions of a_1 .
- (ii) The tile a_0 is of size $1 \times 1 = (2^1 - 1) \times (2^1 - 1)$.
- (iii) The tile a_0 has red arrows pointing out of its right side and from its bottom side.
- (iv) The tile a_0 has green arrows pointing out from its left side and its top sides.
- (v) The tile a_0 has yellow dots.

It is immediate to check that the tiles b_0 , c_0 and d_0 have the same properties as a_0 , up to changing the directions of the red arrows according to the rotation.

Induction. Now, suppose that all the properties of the lemma hold for some integer n . Figure 3.11 shows α_n again, with some information about a_n that we know by recurrence hypothesis. We call *gaps* the four “corridors” that are bordered with green and red arrows pointing inwards.

- (i) The red and green arrows pointing inside the gaps (that come from the recurrence hypothesis on a_n , b_n , c_n and d_n), as well as the copy of \bar{a}_0 in the middle, leave only one possible completion. The right gap and the bottom gap have outgoing red arrows, that are “crossed” by the joining green and red arrows from b_n , d_n and c_n . The top gap and the left gap have outgoing green arrows, similarly “crossed” by joining green and red arrows from b_n , a_n and c_n .
- (ii) By induction hypothesis, a_n has border size $2^{n+1} - 1$, so a_{n+1} has, by construction, border size $2 \times (2^{n+1} - 1) + 1 = 2^{n+2} - 1$.
- (iii) The unique completion of α_n , which is a_{n+1} , has a red arrow pointing out at the right and at the bottom of its “gaps”.
- (iv) All the borders of a_n , b_n , c_n and d_n that face the outside of a_{n+1} have green arrows pointing out by recurrence hypothesis. Moreover, the only possible completion of α_n has green arrows pointing out of the top and left “gaps”.
- (v) The tile in top left-hand corner of a_n has yellow dots by recurrence hypothesis, and it is also the tile in top left-hand corner of a_{n+1} . Similarly, the bottom left-hand corner of b_n has yellow dots (by induction hypothesis) and it is also the bottom left-hand corner of a_{n+1} . A similar reasoning on c_n and d_n shows that the other corners of a_{n+1} also have yellow dots.

We defined b_{n+1} , c_{n+1} and d_{n+1} to be the rotations of a_{n+1} . Thus all the properties on a_{n+1} imply the same properties on b_{n+1} , c_{n+1} and d_{n+1} , except that the orientation of the red and green arrows are adapted according to the rotations. \square

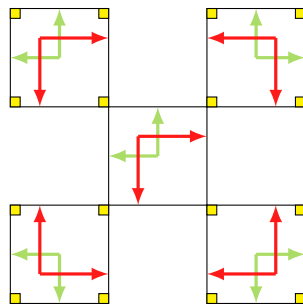


Figure 3.9: The pattern α_0 : a partial macrotile for a_1 .

3.3.4. For any integer n , the function $f : \{a_n, b_n, c_n, d_n\} \rightarrow R$ mapping a_n to a_1 , b_n to b_1 , c_n to c_1 and d_n to d_1 is an injective projection. However, it is not surjective: the tile \bar{a}_0 , for instance, has no preimage. This means that we can use this projection to “collapse” Robinson squares and somehow “move down” in the hierarchy, but we cannot use it to expand single tiles into Robinson squares and thus to “move up” in the hierarchy.

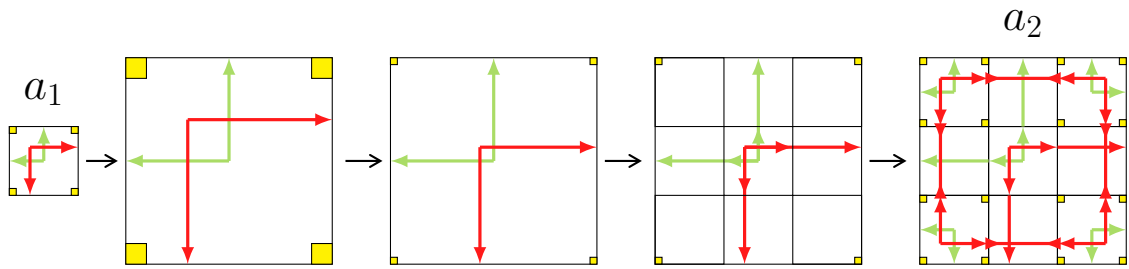


Figure 3.10: Detailed transformation of a macro-tile of level 0 in a macrotile of level 1.

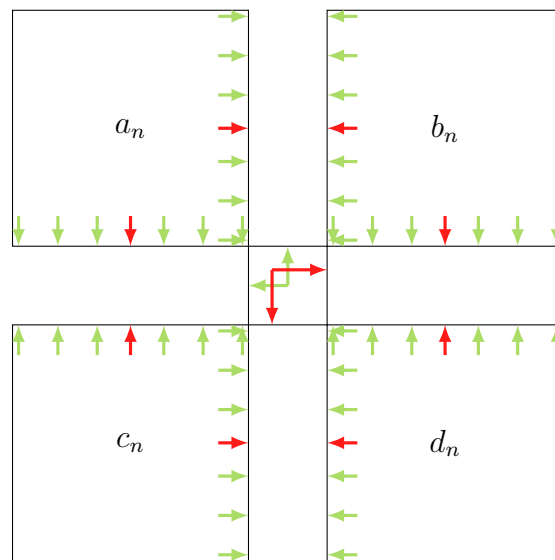


Figure 3.11: The pattern α_n , with additional information from Lemma 3.3.3.

3.4 Robinson's tileset is solvable

Now we prove that Robinson's tileset is solvable.

Theorem 3.4.1. *Robinson's tileset is solvable.*

Proof. Lemma 3.3.3 states that, for each n , the macrotile a_n is a valid tiling by R of a square of size $(2^n - 1) \times (2^n - 1)$. The result follows by the Compactness Theorem 2.2.4. \square

Theorem 3.4.2. *For any tiling T of the plane by R and for any integer n , one of the macrotiles a_n , b_n , c_n or d_n occurs somewhere in T .*

Proof. First, one of a_1 , b_1 , c_1 or d_1 should occur in any tiling because of the rule about yellow dots. By recurrence, we must show that any occurrence of a_n , b_n , c_n or d_n is part of an occurrence of a_{n+1} , b_{n+1} , c_{n+1} , or d_{n+1} . (Note that there is no correspondance here: an occurrence of a_n could be a part of an occurrence of c_{n+1} , for instance.) This proof is similar for a_n , b_n , c_n and d_n , so it can be done for a_n only.

The rest of the argument is just a long case analysis, without any surprise. It is thus left as an exercise in this version of the notes. The slides accompanying Lecture 3 (which could unfortunately not be displayed properly on the *D-Day!*) show where to look. \square

Exercise 3.4.3. Prove that Robinson's tileset is aperiodic: no tiling of the plane by R has a period.

Solution. Suppose there exists an R -tiling T with a period \vec{p} . Then, the translation of T by \vec{p} must send every occurrence of a_n (respectively, b_n , c_n , d_n) on an occurrence of a_n (respectively, b_n , c_n , d_n). We will prove in a moment that in all R -tilings, each tile belongs at most one occurrence of a_n or b_n or c_n or d_n . This means, for instance, that no tile can belong to an occurrence of a_n and an occurrence of b_n at the same time, nor to two different occurrences of a_n at the same time. In other terms, two occurrences of $a_n/b_n/c_n/d_n$ never intersect.

If we accept that this fact is true, then the translation of T by the periodicity vector \vec{p} should send each occurrence of a_n (respectively b_n , c_n , d_n) to a *disjoint* occurrence of a_n (respectively b_n , c_n , d_n), so \vec{p} should have length at least $2^{n+1} - 1$ (the size of a_n and his friends). But since we have occurrences of a_n for arbitrarily large n , the vector \vec{p} should have a length greater than $2^{n+1} - 1$ for arbitrarily large n , which is not possible: we have a contradiction.

Thus it remains to prove that:

Lemma 3.4.4. *In any R -tiling T , two macrotiles of the same level never intersect. In other terms, no tile of T belong at the same time to an occurrence of a_n and to an occurrence of b_n , c_n , d_n , nor to another occurrence of a_n .*

Proof. The proof is by recurrence. For $n = 1$, the objects a_1 , b_1 , c_1 and d_1 are single tiles, so they cannot intersect. Suppose that for some integer n , the macrotiles a_n , b_n , c_n and d_n do not intersect. Consider a_{n+1} , which is the completion of the pattern displayed on Figure 3.8. A simple geometrical argument shows that if an occurrence of b_{n+1} , c_{n+1} or d_{n+1} intersected with a_{n+1} , then we would necessarily have two occurrences of objects $\{a_n, b_n, c_n, d_n\}$ intersecting as well. By recurrence hypothesis, this doesn't happen. So occurrences of objects $\{a_{n+1}, b_{n+1}, c_{n+1}, d_{n+1}\}$ do not intersect either. \square

End of solution.

3.5 The Domino Problem is undecidable (at last!)

3.5.1. We have almost all the ingredients for the proof that the Domino problem is undecidable. We will proceed as in Section 2.4: start from (R) , and build gradually more complicated tilesets $(R1)$, $(R2)$, etc. At some point, we will take an arbitrary Turing machine and embed it in one of those tilesets, in the same way as we did in Section 2.4, but without requiring the constraint at the origin. But first, we need to modify (R) a little bit independently of any Turing machine, just to make things easier later.

3.5.2. Let us start to describe $(R1)$. Instead of having just red arrows, we want to have *even red arrows* (represented by dotted lines) and *odd red arrows* (represented by solid lines). They are defined as follows. Recall that, in a Robinson tiling, each red arrow is part of a red square, which is the border of a macrotile of size $(2^{n+1} - 1) \times (2^{n+1} - 1)$. If n is even, then the considered red arrow is an even red arrow, and if n is odd, then the considered red arrow is odd. See Figure 3.13 for an illustration what a tiling by $(R1)$ looks like.

The tileset $(R1)$, displayed on Figure 3.12, enforces the desired shape in its tilings. This set $(R1)$ have been built from (R) with the following algorithm. For each tile r in (R) :

- if r has *one* red arrow, then copy it twice: one with an even arrow and one with an odd arrow;
- if r has *two* crossing red arrows, then copy r twice: once with the horizontal arrow even and the vertical arrow odd; and once with the horizontal arrow odd and the vertical arrow even;
- if r has *no* red arrows at all, just keep it as-is in $(R1)$.

The idea is that whenever two red arrows are crossing, one of them must belong to a red square of size $2^n - 1$ while the other one must belong to a red square of size $2^{n+1} - 1$. Therefore, they cannot have the same parity. Thus the tiling $(R1)$ is built so that the parities of successive squares are alternating. Figure 3.13 shows four squares of size 3 (in 1-macrotils, thus, odd), one square of size 5 (in a 2-macrotile, thus, even) and a piece of a square of size 7 (in a 3-macrotile, thus, odd).

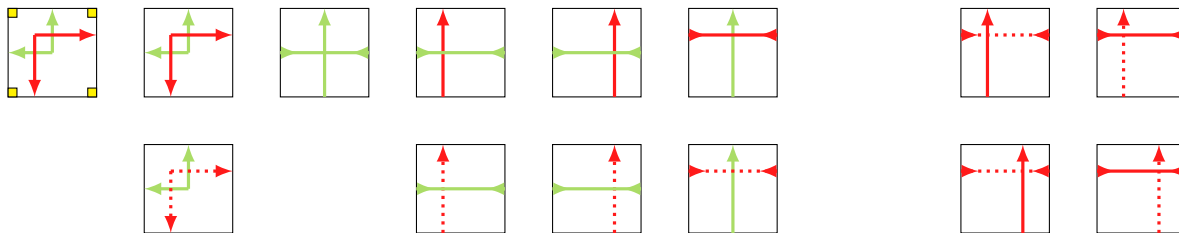


Figure 3.12: Extended Robinson tileset, $(R1)$, with even and odd red arrows. Rotated tiles are not shown.

3.5.3. From now on, even though they do exist, we will not draw yellow dots, green arrows, nor dotted red arrows anymore on our figures. We focus solely on red squares made of solid lines, understanding that those square must be organized as on Figure 3.16 because of the local constraints imposed by not-drawn arrows and dots.

3.5.4. We build $(R2)$ by slightly transforming each tile of $(R1)$: we move the solid red lines very close to their opposite border of the tile, in such a way that the red squares in tilings become very close to the border of the internal square of tiles they delimit. The resulting tileset, $(R2)$, is displayed

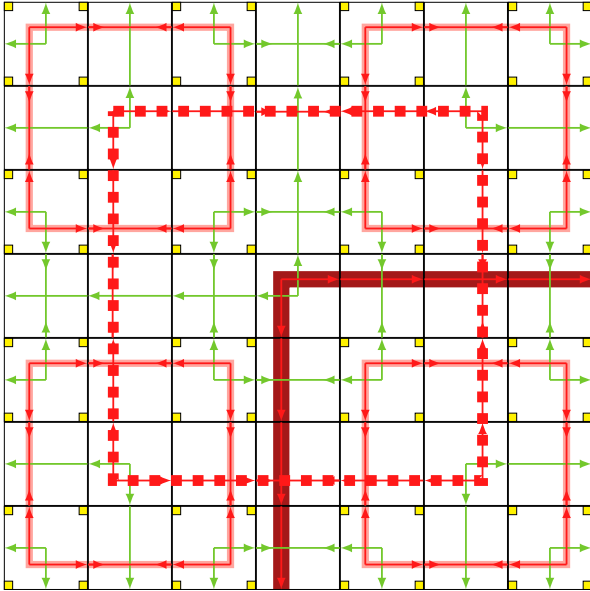


Figure 3.13: The macrotile a_4 in $(R1)$. Different shades of red are used to show the macrotiles of level 2, 3 and 4.

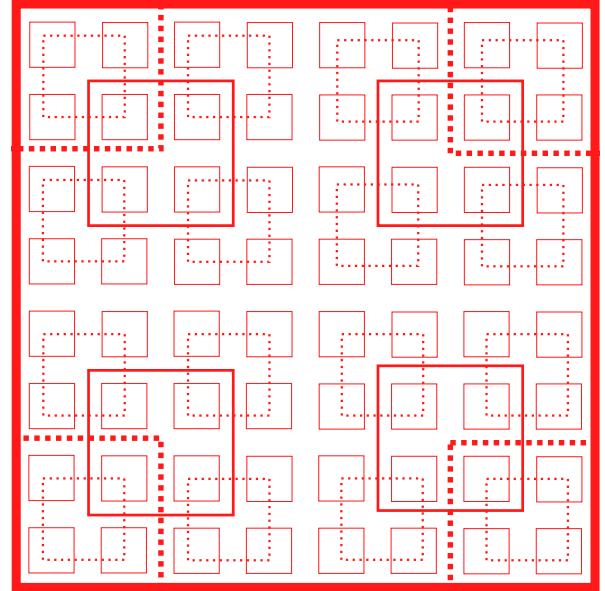


Figure 3.14: A tiling by $(R1)$: red squares with even (dotted) and odd (solid) red arrows. Green arrows and yellow dots are not displayed.

on Figure 3.15. Moreover, Figure 3.17) shows a tileset by $(R2)$ (without the green arrows, dotted red lines nor yellow dots) and Figure 3.18 shows how a single red square is tiled by $(R2)$.

Note that there is a bijective projection between $(R1)$ and $(R2)$; in other terms, the difference between $(R1)$ and $(R2)$ is purely graphical, it does not change the local rules at all. However, $(R2)$ will help to make our figures easier to read in the next paragraphs.

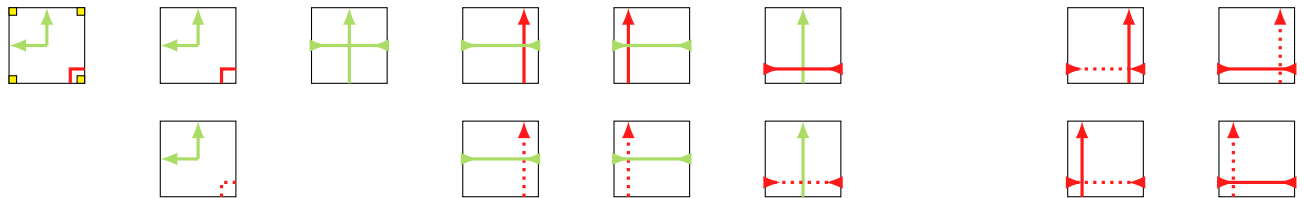


Figure 3.15: The tileset $(R2)$ is a modified version of $(R1)$, where the red arrows are closer to the border.

3.5.5. In $(R2)$, we call:

- *top tile* any tile having a red line near its top border;
- *bottom tile* any tile having a red line near its bottom border;
- *left tile* any tile having a red line near its left border;
- *right tile* any tile having a red line in its right border.

Be careful: by construction of $(R2)$, *the bottom border of a square is actually made of top tiles, and the top border of a square is made of bottom tiles!* Similarly, the right border of each square is made of left tiles, and the left border of each square is made of right tiles. In other words, the red squares tend to be “inside” the tiles that border them. Please see Figure 3.18 to make this picture clearer.

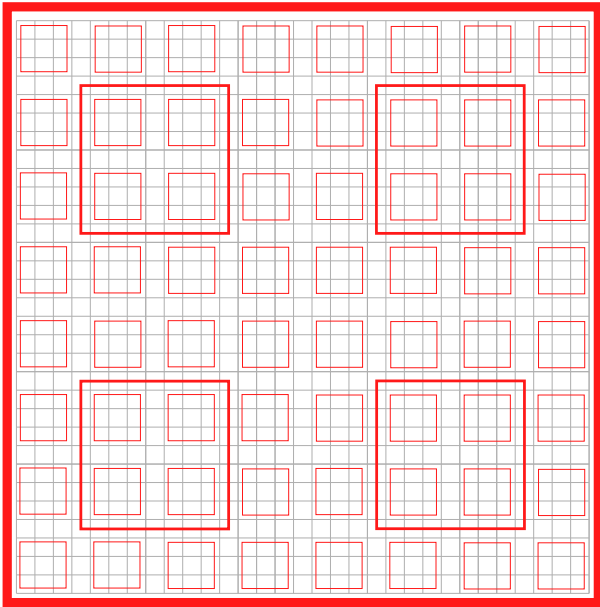


Figure 3.16: Red squares of odd level tiled by $(R1)$.

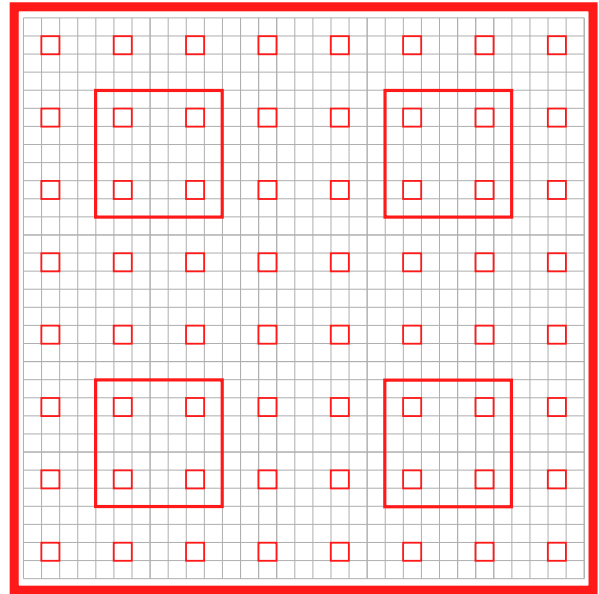


Figure 3.17: Red squares of odd level tiled by $(R2)$.

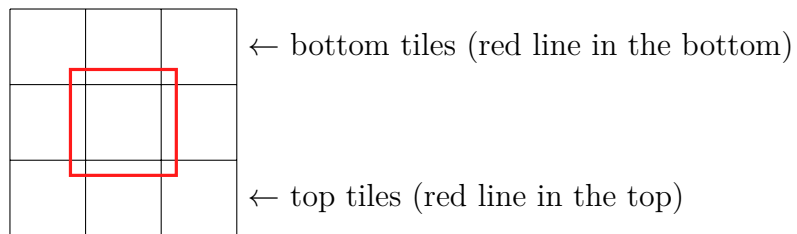


Figure 3.18: The top border of a square is made of bottom tiles, and vice-versa.

3.5.6. Consider again Figure 3.17. We say that each tile *belongs* to a red square, namely the smallest square that surrounds that tile. For instance, in Figure 3.18, only the central tile belongs to the visible red square. Or, in Figure 3.19 (don't look at all the details), all the dotted tiles belong to squares of size 1×1 , while yellow tiles belong to the square of size 7×7 . (The tiles with arrows also belong to this square, we will come to their meaning in a moment.)

Let C denote any red square; some tiles inside C belong to C , and some tiles do not: they belong to smaller squares that are contained in C . We say that a tile at position (x, y) is *pure* in C if it belongs to C , and if all tiles on the same line and on the same column as (x, y) in C also belong to C . For instance if C is a 7×7 -square of Figure 3.19, then the yellow tiles are pure in C , and the tiles marked with double arrows (\Leftrightarrow and \Updownarrow) belong to C but are not pure.

In a square C , the tiles that belong to C but that are not pure are called *unpure*. There are two kinds of unpure tiles: the *vertical unpure tiles* have a non-belonging tile on the same *column*, while the *horizontal unpure tiles* have a non-belonging tile on the same *row*. Each unpure tile is either vertical unpure, horizontal unpure, or both at the same time. On Figure 3.19, vertical unpure tiles are marked with horizontal arrows (\Leftrightarrow) while horizontal unpure tiles are marked with vertical arrows (\Updownarrow). The reason why the arrows seem reversed will become clear in a few paragraphs. Note that each tile belong to a square and is either pure, vertical unpure or horizontal unpure (or both kinds of unpure at the same time) in that square.

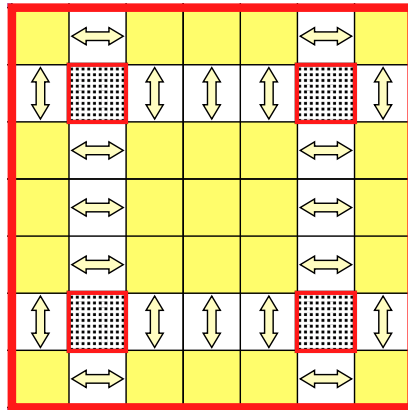


Figure 3.19: Yellow cells denote pure cells, arrows denote vertical unpure and horizontal unpure cells, dotted cells are inside small red squares.

3.5.7. The tileset $(R3)$ is a copy of $(R2)$ where each tile “knows” whether it is pure, vertical unpure or horizontal unpure. Formally, the set of colors of $(R3)$ is $\mathcal{C} = \mathcal{C}((R2)) \times \{\mathcal{P}, \mathcal{V}_1, \mathcal{V}_2, \mathcal{H}_1, \mathcal{H}_2\}$, where $\mathcal{C}((R2))$ is the set of colors of $(R2)$, and \mathcal{P} , $\mathcal{V}_{\{1,2\}}$ and $\mathcal{H}_{\{1,2\}}$ respectively mean “pure”, “vertical unpure” and “horizontal unpure”. To build $(R3)$, we follow these rules.

- For each tile with a red line near the left, make all copies with $\{\mathcal{H}_1, \mathcal{P}\}$ on the right and $\{\mathcal{H}_1, \mathcal{H}_2\}$ on the left.
- For each tile with a red line near the left, make all copies with $\{\mathcal{H}_2, \mathcal{H}_1\}$ on the right and $\{\mathcal{P}, \mathcal{H}_1\}$ on the left.
- For each other tile, make all copies of that tile with the same element of $\{\mathcal{P}, \mathcal{H}_1, \mathcal{H}_2\}$ on the left and on the right.

(The rules to treat $\mathcal{V}_{1,2}$ and vertical impurity are exactly symmetric.) To understand these rules, let us scan a big square (Figure 3.21) from left to right. The red line on the left is a right tile; the next

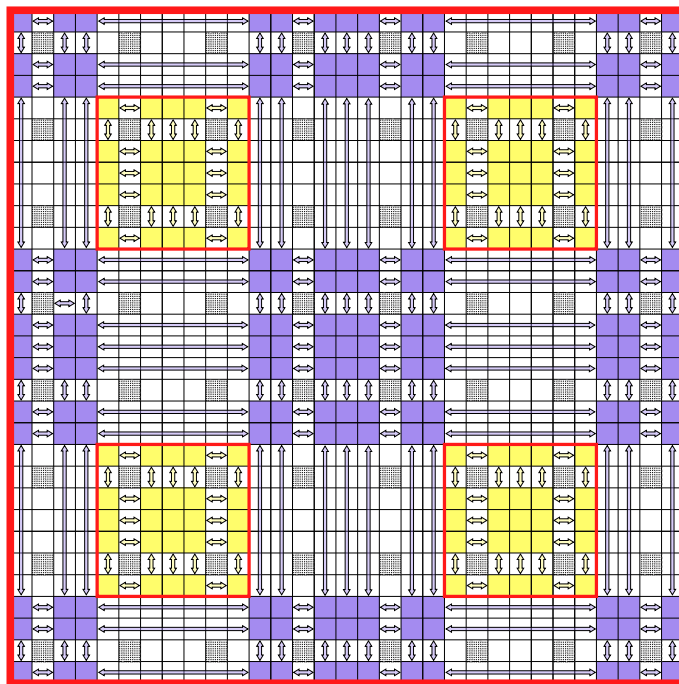


Figure 3.20: Two levels of hierarchy: yellow filling and arrows denote computational cells of the 1st level; violet filling and arrows—of the 2nd level.

red line that we find may either be a left line, which means that we found the end of the big square and thus all the tiles in between were pure. The next red line we find may also be a right line, which means that we found the beginning of a smaller square. In this case, all the tiles in between are horizontally impure. The situation is symmetrical if we start from the right of the square.

Finally, the tiles between two small squares are also vertically impure. Since the rule to detect them is a bit different, we use a separate color, \mathcal{H}_2 . The idea is that a tile to the right of a left tile is necessarily impure.

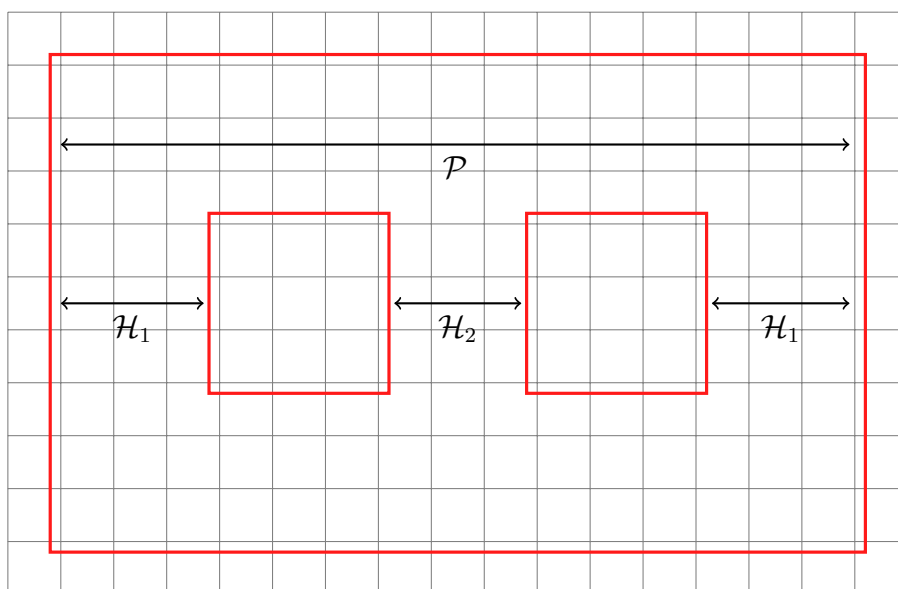


Figure 3.21: How to locate horizontal impure tiles.

The result is a tiling set (R_3) that has a surjective projection to (R_2), but which is partitioned in three subtiles: the subset of pure tiles (\mathcal{P} on every side), the subset of vertical impure tiles (\mathcal{V}_1 or

\mathcal{V}_2 on either horizontal side), and the subset of horizontal unpure tiles (\mathcal{H}_1 or \mathcal{H}_2 on either vertical side).

3.5.8. Let P denote a Turing machine and U the tileset that forces to draw a space-time diagram of $P(\#)$ (this is tileset $(T5)$ in the Section 2.4). All we have to do now is to merge U and $(R3)$ into a tileset V , such that:

- the computation of $P(\#)$ is only done on pure tiles;
- the vertical (respectively horizontal, both kinds) unpure tiles transmit the information about the computation horizontally (respectively vertically, both directions);
- the bottom, left-hand corner of each read square contains the head of a Turing machine.

It is not hard to take the cartesian product of U and $(R3)$, and then to remove the appropriate tiles to get the desired properties.

Consequently, in any tiling of the plane by V , each red square contains a space-time diagram of $P(\#)$, whose size depends on the size of the square. If the computation is not finished when the end of the square is reached, it is simply interrupted. This is not a problem because we have arbitrarily large squares, so another, bigger square will exist somewhere and give more time to the computation.

Theorem 3.5.9. *The tileset V tiles the plane if and only if $P(\#)$ does not halt. Moreover, the transformation from P to V is computable.*

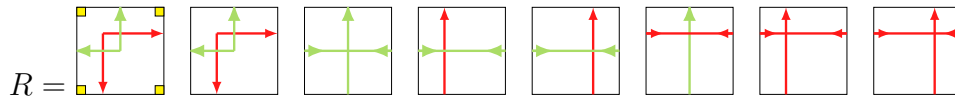
Proof. By Theorem 3.4.2, any tiling of the plane by $(R3)$, and thus by V , contains arbitrarily large red squares. A simple recurrence shows that the number of pure tiles inside a red square is strictly increasing with the size of the square. Therefore, any tiling of the plane by V has to contain arbitrarily large space-time diagrams of $P(\#)$. This is possible if and only if $P(\#)$ does not halt.

To see that the transformation between P and V is computable, observe that the tileset $(R3)$ is finite and known in advance. The tileset U can be produced from P using the algorithm from the previous chapter. Finally, V is a subset of the product of U and $(R3)$, and this subset is easily seen to be computable. □

3.e Exercises for Lecture 3

Here are all the exercises from the lecture notes, reorganized and renumbered to make an exercise sheet. There are also a few new questions, to keep things interesting. Solve only what you like.

Robinson tileset



Exercise 3.e.1. (a) Prove that there are infinitely many Robinson tilings of the plane.

(b) Prove that there are *uncountably* many Robinson tilings of the plane.

(c) Prove that no Robinson tiling has a period: thus, the Robinson tileset is aperiodic.

(d) Show that there exists a Robinson tiling with a right-infinite red arrow:



(e) Show that there exists a Robinson tiling with a bi-infinite red arrow:



Exercise 3.e.2. An *emulation* of R by Wang tiles is a pair (W, f) where W is a Wang tileset and f is a map $W \rightarrow R$ such that for any valid tiling T by W , the tiling $f(T)$ by R is also valid. (Apply f on each tile of T to get $f(T)$.) Design an emulation of R by Wang tiles, having:

(a) ≤ 64 tiles;

(b) 56 tiles.

Hierarchy

If X and Y are two rectangles of tiles with the same height, then we define \oplus as $X \oplus Y := \begin{matrix} X & Y \end{matrix}$.

If X and Y are two rectangles of tiles with the same width, then we define \ominus as: $X \ominus Y := \begin{matrix} X \\ Y \end{matrix}$.

Given a tileset A , an $(n \times m)$ -*substitution* is a function $s : A \rightarrow A^{n \times m}$ which maps each tile to a valid block of size $n \times m$, so that for all tiles x, y in A :

- $x \ominus y$ is valid if and only if $s(x) \ominus s(y)$ is valid;
- $x \oplus y$ is valid if and only if $s(x) \oplus s(y)$ is valid.

Exercise 3.e.3. Consider the tileset $T = \{ \begin{matrix} \text{orange} & \text{blue} \\ \text{blue} & \text{orange} \end{matrix}, \begin{matrix} \text{orange} & \text{orange} \\ \text{blue} & \text{orange} \end{matrix}, \begin{matrix} \text{orange} & \text{orange} \\ \text{orange} & \text{blue} \end{matrix}, \begin{matrix} \text{orange} & \text{orange} \\ \text{orange} & \text{orange} \end{matrix} \}$:

(a) find an (1×3) -substitution for T ;

(b) find an $(n \times 3)$ -substitution for T , for all $n \geq 2$.

Exercise 3.e.4. Let A be a tileset, s be a substitution $s : A \rightarrow A^{n \times m}$ and T be a tiling of the plane by A . We say that T is a *substitution tiling* for s if and only if, for any block C appearing in T , there exists an integer n and a tile a such that C appears in $s^n(a)$.

- (a) Design a substitution tiling for the (2×3) -substitution from Exercise 3.e.3.
- (b) Prove that there exist no substitution tiling for the (1×3) -substitution from Exercise 3.e.3.
- (c) Prove that, for any tileset A and any substitution $s : A \rightarrow A^{1 \times n}$, there exist no substitution tiling for A and s . Same question if s is $s : A \rightarrow A^{n \times 1}$.

Exercise 3.e.5. Let A denote a tileset, s denote a substitution and T denote a substitution tiling for A and s . We say that T has an *unique derivation* if there is *exactly one* tiling U such that $T = s(U)$.

- (a) Show that the tiling you found for Exercise 3.e.4(a) doesn't have unique derivation.
- (b) Show that there is no tiling with unique derivation for the (2×3) -substitution from Exercise 3.e.3.
- (c) Let T_0 denote a substitution tiling with substitution s . Prove that if there is an *unique* infinite sequence of tilings T_1, T_2, T_3, \dots such that:

$$T_n = s(T_{n+1})$$

for all n in \mathbb{N} (in particular, each T_n has an unique derivation), then T_0 is aperiodic.

Contacts

- Daria Pchelina (dpchelina@clipper.ens.fr)
- Guilhem Gamard (guilhem.gamard@normale.fr)

Appendix A

Bibliography

- [1] Simon Marlow et al. *The Haskell 2010 Language Report*. Self-published, 2010. URL: <https://www.haskell.org/onlinereport/haskell2010/>.
- [2] Robert Berger. “The undecidability of the domino problem”. In: *Memoirs of the American Mathematical Society* 66 (1966). His PhD thesis, with the same title, was defended in 1964 at Harvard university; it contains pretty much the same text.
- [3] Leo Brodie. *Starting forth*. Prentice-Hall, 1981. URL: <https://www.forth.com/starting-forth/>.
- [4] Emmanuel Jeandel and Michaël Rao. “An aperiodic tileset of 11 Wang tiles”. In: *Arxiv* 1506.06492 (2015). URL: <https://arxiv.org/abs/1506.06492>.
- [5] *Pypy*. URL: <https://pypy.org>.
- [6] Raphael M. Robinson. “Undecidability and Nonperiodicity for Tilings of the Plane”. In: *Inventiones Mathematicae* 12.3 (1971).
- [7] Michael Sperber et al. *Revised⁶ Report on the Algorithmic Language Scheme*. Cambridge University Press, 2007. URL: <http://www.r6rs.org/>.
- [8] Hao Wang. “Dominoes and the AEA case of the decision problem”. In: *Computation, Logic, Philosophy: A Collection of Essays*. Mathematics and its Applications (China Series). Springer, 1989.
- [9] Hao Wang. “Proving theorems by pattern recognition—II”. In: *Bell System Technical Journal* 40.1 (1961), pp. 1–41.

Appendix B

List of figures

1.1	The memory (filled with random numbers) and the head.	5
1.2	Example of a program (that doesn't do anything interesting).	6
1.3	A space-time diagram of a Turing machine.	8
1.4	Turing machine writing the integer $n = n_k \dots n_1 n_0$ in the memory.	8
2.1	A tileset and a (part of) a valid tiling by this tileset.	21
2.2	A simple tiling.	22
2.3	A sequence of elements of \mathbb{Z}^2 that covers \mathbb{Z}^2	23
2.4	A repeatable square.	24
2.5	A line tiled by (T1).	27
2.6	A line tiled by (T2).	27
2.7	Two consecutive lines tiled by (T4) for $v = R$	29
2.8	Two consecutive lines tiled by (T4) for $v = L$	30
2.9	Transformation of a tile of (T5) into a tile of (T5')	31
2.10	This symmetric tiling does the computation of $P(\#)$ twice.	32
3.1	Robinson tiles, up to rotation.	35
3.2	Examples of adjacent tiles.	36
3.3	Four tiles sharing a corner.	36
3.4	A piece of a Robinson tiling.	36
3.5	Checking if two macrotiles may be neighbours.	37
3.6	Iterating a projection of tiles on macrotiles	38
3.7	Robinson's macrotiles a_0, b_0, c_0 and d_0	38
3.8	The pattern α_n : partial macrotile for a_n	39
3.9	The pattern α_0 : a partial macrotile for a_1	40
3.10	Detailed transformation of a macro-tile of level 0 in a macrotile of level 1.	41
3.11	The pattern α_n , with additional information from Lemma 3.3.3.	41
3.12	Extended Robinson tileset, (R1), with even and odd red arrows. Rotated tiles are not shown.	43

3.13	The macrotile a_4 in $(R1)$. Different shades of red are used to show the macrotiles of level 2, 3 and 4.	44
3.14	A tiling by $(R1)$: red squares with even (dotted) and odd (solid) red arrows.	44
3.15	The tileset $(R2)$ is a modified version of $(R1)$, where the red arrows are closer to the border.	44
3.16	Red squares of odd level tiled by $(R1)$	45
3.17	Red squares of odd level tiled by $(R2)$	45
3.18	The top border of a square is made of bottom tiles, and vice-versa.	45
3.19	Yellow cells denote pure cells, arrows denote vertical unpure and horizontal unpure cells, dotted cells are inside small red squares.	46
3.20	Two levels of hierarchy: yellow filling and arrows denote computational cells of the 1st level; violet filling and arrows—of the 2nd level.	47
3.21	How to locate horizontal unpure tiles.	47