Chapter 7

Computability

Computability was pioneered by Alan Turing and Kurt Gödel. Turing probably committed suicide by eating an apple he poisoned with cyanide. Gödel on the other hand had an obsessive fear of being poisoned with food; when his wife was hospitalized, he refused to eat, and eventually starved to death. Now it's your turn to study this intoxicating subject.

7.1 Undecidability

In the previous chapters, we have analyzed various computational tasks. While some functions were easy to compute efficiently, others were difficult; in these cases, we have focused on approximations or heuristics. However, given enough time and resources, could we always find the solution to a problem?

Problem 7.1 (Halting problem)**.** *Given a program P and an input x to P, does $P(x)$ halt (stop running) after a finite amount of time?*

Remarks:

- Can we write a Python program that solves Problem 7.1? Somewhat surprisingly, the input of our program is also a program (plus an input parameter of this program).
- Naturally, we must somehow encode the input program. There are various ways to do this. We can, for example, consider the whole code of the program as a long string of text, encoding each character in this string with a byte.
- The halting problem is sometimes easy to solve, for example, in case of the simple programs in Algorithm 7.2. But is it always?

```
def P_1(x):
    print("Hello, world!")
    return

```

```
def P_2(x):
    x = 1
    while x > 0:
        x += 1
    return
```

Algorithm 7.2: Example programs; P_1 is halting, P_2 is not.

Definition 7.3 (Undecidable)**.** *We say that a problem is undecidable if no algorithm solves the problem in finite time for every possible finite input.*

Remarks:

- This should be surprising! Definition 7.3 does not say that the runtime increase will be exponential or even double exponential in the input size, but that the problem really cannot be solved in *any* finite amount of time.

Theorem 7.4. *The halting problem is undecidable.*

Proof. Assume for the sake of contradiction that there exists a program $P_H(P, x)$ that successfully solves the halting problem in finite time for any input. Test program $P_T(x)$ then takes a bitstring as an input and calls P_H as a subroutine:

```
1 def P_T(x):
2     if P_H(x,x) == True:
3         while True: pass    #loop forever
4     else: return
```

Algorithm 7.5: Test program $P_T(x)$.

P_T interprets its input x as a program encoding, and calls the halting solution P_H on program x with input x. Since we assumed that P_H can always solve this problem in finite time, Line 2 is evaluated in finite time for any x.

Since P_T is also a program, it has a bitstring encoding τ according to our encoding scheme. What happens when we call $P_T(\tau)$? Note that this means that we are calling $P_H(\tau, \tau)$ as a subroutine, i.e. querying whether the program described by τ (that is, P_T) halts on input τ.

- If $P_H(\tau, \tau)$ is true, then P_T goes in to an infinite loop according to our code, and never halts. Hence $P_H(\tau, \tau)$ should be false instead!
- If $P_H(\tau, \tau)$ is false, then P_T immediately terminates according to our code, so $P_H(\tau, \tau)$ should be true!

We get a contradiction in both cases, so the halting problem is undecidable. □

Remarks:

- We assumed that the encodings of a program and its input are simply bitstrings. This is also close to practice. What if a bitstring is an invalid program, not respecting the syntax of programs? We could argue that in this case, the program simply halts (with an error).
- Of course Theorem 7.4 only holds in general: we cannot solve the halting problem correctly in finite time for *every* algorithm-input pair. Some specific algorithm-input pairs, e.g. the simple examples in Algorithm 7.2, can be decided easily.
- Also, a program P that actually halts is easy as well: we just run/simulate P, which will eventually halt. The halting problem is only undecidable because of programs P that do not halt. In this case it is difficult to distinguish if P is still running because it did not reach the halting point yet, or because it is never going to halt.
- There is a name for this weaker kind of decidability that is only required to work in one of the two cases (unlike our original concept of decidability in Definition 2.3).

Definition 7.6 (Semi-decidable)**.** *We say that a problem is semi-decidable if there exists an algorithm $\mathcal{A}$ such that*

- *if the answer is* `True`*, then $\mathcal{A}$ outputs* `True` *in finite time,*
- *if the answer is* `False`*, then $\mathcal{A}$ either outputs* `False` *in finite time or keeps running indefinitely.*

Theorem 7.7. *The halting problem is semi-decidable.*

Remarks:

- Given the undecidability of halting, is there an easy way to show that some other problems are also undecidable? Yes, we can use reductions again (Definition 2.6). Given a problem Π, we can show that if Π was decidable, then the halting problem would also be decidable. This implies that Π is also undecidable. One slight difference from Definition 2.6, however, is that for this argument, we do not need the reductions to run in polynomial time.
- Here is an example for such a reduction.

Problem 7.8 (Mortality problem)**.** *Given a program P, is it true that $P(x)$ halts for any possible input x?*

Remarks:

- This is different from the halting problem, but only so much: instead of a specific input, we now want to know if P halts on *every* input.

Theorem 7.9. *The mortality problem is undecidable.*

Proof. Assume that we have a program P_M that takes a program description as an input, and solves the mortality problem in finite time. Then given a specific input for the halting problem (a program P and an input x), consider the following program:

```
def P_T(y):
    if y == x:
        run program P(x)
    else:
        return
```

Algorithm 7.10: Another testing program.

Now let us run our mortality solution P_M on the encoding of P_T. We know that P_T certainly terminates in finite time for any input different from x. This implies that $P_M(P_T)$ is true if and only if $P(x)$ halts; hence a solution for the mortality problem allows us to solve the halting problem on P and x. Since halting is undecidable, such a solution cannot exist, so we have a contradiction. □

Remarks:

- Well, that was not so surprising; after all, mortality is a close relative of the halting problem.
- How about problems that are a far cry from halting? We need a clean and simple theoretical definition of what we mean by a program, algorithm or function. Let us make a brief detour into abstract machine models and computation theory.

7.2 The Turing Machine

What kind of building blocks do we need to obtain a simple theoretical model of a machine that can, intuitively speaking, do the same computations as a real computer?

Remarks:

- First we need some abstract *states* that represent the current state of our program, and we need to describe the transitions between these states. Such a set of states with predefined transition rules is known as a finite automaton.
- We also need some memory to store data. We usually assume that a memory consists of cells; our program can read data from these cells, write data into these cells, and move between these cells to be able to access any of them. We will now consider a *tape* of cells that is infinite in both directions, i.e. the cells can be enumerated by integers $\ldots, -2, -1, 0, 1, 2, \ldots$ (from $-\infty$ to ∞).

- We also have a *tape pointer* that points to a specific tape cell at each point in time, indicating that this is the tape cell that we can currently read/write. Initially the tape pointer points to cell 0.
- What kind of data can we write onto this tape? We assume that we have an *alphabet* Σ of possible symbols, and we can write exactly one symbol into each cell. In the simplest case, this alphabet can be binary, i.e. $\Sigma = \{0, 1\}$. We usually use some extra symbol, e.g. $\perp$ for the cells that we consider empty, and we use $\Sigma := \Sigma \cup \{\perp\}$.
- These building blocks define a famous theoretical model of computation.

Definition 7.11 (Turing Machine or TM)**.** *A Turing Machine has a finite set of states S, and a two-way infinite tape. Initially the machine is in a specified starting state $s_0 \in S$, the tape has some symbols on it (the input), and the tape pointer points to cell 0 of the tape.*

In each discrete time step, depending on the current state s and the current tape cell content σ, the machine executes the following steps:

- *change to another state $s' \in S$,*
- *write the tape, i.e. change the content σ of the current tape cell to any symbol $\sigma' \in \Sigma$,*
- *possibly move the tape pointer one step to the left or one step to the right.*

Formally a TM is defined by a function $(S, \Sigma) \to (S, \Sigma, m)$, where $m \in \{left, right, stay\}$ indicates the movement of the tape pointer.

With a TM we usually also select a specific halting (accepting) state $s_h \in S$. We say that the TM accepts an input if, when executed on this input, the TM eventually enters state s_h.

Remarks:

- We assume that computation is over whenever the halting state s_h is reached: the machine does not do anything (i.e. never changes the state/tape content/tape pointer) after this point, and the current tape content is considered to be the *output* of the computation.
- While a TM acts as a model of computation, we can also interpret it as a function: it converts an input (the initial content of the tape) to an output (final content of the tape).
- However, the function is not complete: for some inputs, the output may be undefined, since just like a Python program, a TM can easily run forever and never halt.
- How can we do actual computations in this abstract setting? Let us see an example for a simple operation: incrementing an integer.

Example 7.12 (Incrementation with a TM)**.** *Given a positive integer input x, our task is to increment x by 1. We assume that the input x is given in a binary representation, starting with least significant bit (LSB) first at cell 0, and going until cell $\lfloor \log_2 x \rfloor$. The tape pointer starts at cell 0, and empty cells of the tape are marked with a $\perp$.*

Lemma 7.13. *Example 7.12 can be solved on a simple TM with 2 states.*

Proof. The formal process of incrementing a binary number is as follows. We start going from the LSB to the most significant bit (MSB) until we encounter a 0, and we change every 1 to a 0 during the process. When we first find a 0, we change it to a 1 (or if we have left the MSB, we add an extra 1 to the front), and the incrementation is done. We have to translate this process to states and transitions.

This can be done with two states s_0 and s_h. The starting state is s_0, this is where the execution begins; s_h is a halting state where none of the transitions do anything. The transitions from s_0 are defined as follows:

Transitions from state s_0

Read		Write	Pointer	Next state
1	$\longrightarrow$	0	right	s_0
0 or $\perp$	$\longrightarrow$	1	stay	s_h

This ensures that the machine enters the halting state exactly when the incrementation is finished, i.e. when the tape contains $x + 1$ in the same binary representation. □

Remarks:

- Describing a TM for more complex computations can be some work, since it usually requires a higher number of states and transitions. Even in case of our incrementation example, if the number is in a reversed representation (i.e. starting with MSB), we already need an extra state to first move to the end of the input and then start processing the input from the other direction.
- The definition of the halting problem on TMs is as follows: given a description of a TM and an input (initial tape content), decide if this TM ever goes into the halting state. Note that this is only a reformulation of our original halting problem, so the same proof shows that this problem is undecidable.
- There also various other versions of TMs, e.g. a TM that has multiple tapes, and it can read/write these tapes simultaneously in every step. One can show that this is equivalent to the single-tape setting in terms of computability.
- There is one important concept in computation that this basic machine model cannot capture: randomization. In order to model that, we need to slightly extend the machine model.

Definition 7.14 (Randomized TM or RTM)**.** *In a Randomized Turing Machine, each transition is replaced by a set of available transitions, and a probability distribution over these transitions. In each step, a transition is chosen at random according to this probability distribution.*

Remarks:

- For example, a program on an RTM might have a state where it moves left on the tape with 50% probability, and moves right with 50% probability.
- Another formulation of RTMs is to take a deterministic TM, and add an extra tape of infinite random bits to the machine. The TM then reads bits from this extra tape, and (possibly) executes different transitions based on the next random bit.
- Randomness is a useful tool. However, strictly speaking, randomness does not increase the power of the machine in terms of computability. If we can solve a decision problem Π on an RTM in finite time, then we can also solve Π on a regular TM, by enumerating and simulating all the possible randomized outcomes. Note that this might increase the running time drastically.
- Our argument uses an important assumption: that we can use a TM to simulate the execution of another TM. We also need this property when expressing the proof of Theorem 7.4 in a TM-based context.
- Luckily, this is possible:

Theorem 7.15 (Universal TM)**.** *There exists a Universal Turing Machine which receives the encoding of another TM T (i.e., a program encoded as a string) and an input x to T on its tape, and simulates the behavior of T on x.*

Remarks:

- This is somewhat similar to a real-world Von Neumann computer architecture, where source code, constants and inputs of a computation are stored in the same memory.
- So how close are TMs to real computers? The fact that our program moves between a finite number of states is pretty realistic. What is unusual, however, is that we can only move in memory one step at a time.
- More realistic machine models do exist:

Definition 7.16 (RAM Machine)**.** *A RAM Machine is a model of computation that has explicit registers (instead of only cells) which can store integer values. The machine is capable of addressing these registers indirectly through pointers (instead of moving only sequentially between them).*

Remarks:

- While RAM machines seem more expressive, they are in fact equivalent to TMs: any program on a RAM machine can also be simulated on a TM.
- Since TMs are simpler, we usually stick to TMs. We say that a problem Π is computable if a TM can compute Π. This means that we can essentially use TMs to define the general notion of an algorithm.

Theorem 7.17 (Church-Turing Thesis)**.** *Any real-world algorithm or computation can be translated into an equivalent computation on a TM.*

Remarks:

- Intuitively, we can imagine an algorithm as a computation we can do with pen and paper, using a finite set of rules. This describes both our notion of real-world programs and the set of computations that are doable on a TM.
- The term Church-Turing thesis is often used differently in different contexts. The version shown above is a slightly informal phrasing, relating TMs to real-world computations. Sometimes the Church-Turing thesis is not considered as a theorem, but rather as a definition of the term algorithm.
- The Church-Turing Thesis allows us to classify models of computation: we consider a model "complete" if it can be used to run any algorithm according to this definition.

Definition 7.18 (Turing-complete)**.** *We say that a model of computation is Turing-complete if it can simulate any TM.*

Remarks:

- Naturally, TMs are Turing-complete. And so are RAM-machines, (Definition 6.25), and all popular programming languages, e.g. Python, C++ or Java.
- While a futuristic quantum computer surely is an impressive vision, it only allows us speed up the computation of problems. As such, quantum computers are also not "more" than Turing-complete: they can solve the same computational problems as a (Randomized) TM. In fact, there is no known computational model that can compute more than a TM!
- It is also not so easy to think of reasonable computational models that can execute a smaller subclass of computations than TMs. Some examples for such models are finite automata (essentially TMs without tape) or regular expressions.
- How about very different models that do not look anything like computers or programming languages? Can we also use them to do computations?

7.3 Computing on Grids

Definition 7.19 (Tile)**.** *A tile is a 1×1 square. Each side of the tile (left, top, right, and bottom) has a specific color. We assume that we are not allowed to rotate or flip tiles.*

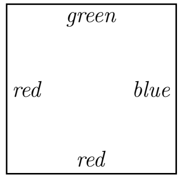

Figure 7.20: Example of a tile

Definition 7.21 (Correct Tiling)**.** *Two tiles can be placed next to each other if their touching side has the same color. E.g. if tile t_1 is red on its top and tile t_2 is red on its bottom, then we can place t_2 directly above t_1.*

Remarks:

- Our goal is to tile a given shape (e.g. a 5×4 rectangle) with a set of tiles such that the tiling is correct.

Problem 7.22 (Tiling)**.** *Assume we are given a set of n tiles, and we can take an arbitrary number of copies of each of these tiles. Does there exist a correct tiling of the entire infinite plane with our given set of tiles?*

Remarks:

- At first glance, this problem seems to have no connection to TMs or the halting problem.
- The most simple correct tiling one can imagine is a periodic tiling, when the same pattern of tiles keep repeating.

Definition 7.23 (Periodic Tiling)**.** *We say that a tiling of the plane is periodic if there exist positive integers w, h such that for every pair of coordinates $i, j \in \mathbb{Z}$, grid square (i, j) has the same tile as grid squares $(i + w, j)$ and $(i, j + h)$.*

Remarks:

- In a periodic tiling, we tile a $w \times h$ rectangle R such that the tiling within rectangle R is correct, and it is also correct to place two such rectangles R next to each other top/bottom or left/right. Then we can cover the entire plane with copies of R.

Lemma 7.24. *If we know that a periodic tiling exists, we can find it in finite time.*

Proof. We take every possible rectangle size $w \times h$, in an increasing ordering according to the sum $w + h$: first 1×1, then 1×2 and 2×1, then 1×3, 2×2 and 3×1, and so on. For each such size, we can try all possible tilings in each of these rectangles, and check their correctness.

If there exists a periodic tiling with a rectangle of size $w \times h$, then we try at most $(w + h)$ total sizes, and thus at most $(w + h)^2$ rectangle shapes before reaching $w \times h$. Each such shape has at most $n^{(w+h)^2}$ possible tilings. Since $(w + h)^2 \cdot n^{(w+h)^2}$ is a finite number, the algorithm indeed terminates in finite time. □

Remarks:

- Unfortunately, this does not answer the question whether the tiling problem is decidable in general. There are sets of tiles where a tiling of the entire plane is possible, but only in a fashion that is not periodic.
- To settle the question of decidability, we show that these tilings are in fact a surprisingly expressive model: we can use them to simulate any TM. This property will allow a reduction to the halting problem.

Theorem 7.25. *Some tile sets are Turing-complete: tilings can simulate the run of any TM on any input x.*

Proof. To outline the main idea of the proof, we will assume a slightly simpler setting: that we only need to tile the bottom half of the plane, i.e. below the origin. This is only for convenience; with further tricks, the same proof method can be extended to the entire plane.

The main idea of the proof is that each row describes the complete state of the tape of a TM in a specific time step, with the top row corresponding to time step 0, the row immediately below corresponding to time step 1, and so on. We can design the tiles carefully such that given a specific row (i.e. current configuration of the TM), the only possible tiling of the row directly below is the next configuration of the TM.

For example, for each tape symbol σ, we can create a tile that has color σ on the top and bottom, and a special default color (say, white) on the left and right. This already allows us to automatically copy the content of the tape into the row below. Furthermore, we use special tiles to keep track of the tape pointer and the current state: the tape cell with the pointer will also be marked with the current state. If, for example, we have a transition from state q_1 to q_2 that also replaces a 0 by a 1 on the tape and moves the tape pointer one step to the right, then we can describe this behavior with the tiles shown in Figure 7.26.

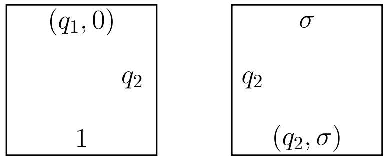

Figure 7.26: Example tiles to simulate a transition of the TM.

The number of tiles we require altogether is only a function of the tape alphabet size and the number of states and transitions in the TM, and thus it is a finite number. By defining some special tiles for the first row, one can also ensure that the only possible tiling of the first row is to have the input x on the tape, and the tape pointer and TM state in their desired initial position.

With such a set of tiles, if the TM halts on x in k steps, then this allows us to tile the first k rows of the plane, and then no tiling will be possible for the $(k+1)^{\text{th}}$ row (since there is no following configuration of the TM). On the other hand, if the TM runs forever, then there is always a next configuration, so the tile set allows us to tile the entire lower half of the plane. □

Remarks:

- So we can do actual computations with a set of tiles!

Theorem 7.27. *The tiling problem is undecidable.*

Proof. Consider an instance of the halting problem with TM T and input x. As we have seen in Theorem 7.25, we can create a set of tiles that correspond to running T on x, and a tiling with this set is possible if and only if T halts on x. Thus solving this tiling problem allows us to decide whether T halts on x; however, the halting problem is undecidable, so the tiling problem must be undecidable, too. □

Remarks:

- There are other models of computation that work on grids. A popular example is Game of Life.

Definition 7.28 (Game of Life or GoL)**.** *In Game of Life, each cell has two states, black (alive) and white (dead), and the update rule is as follows:*

- *If the cell is black: if it has exactly 2 or 3 black neighbors among its 8 neighboring cells, it remains black, otherwise it becomes white.*
- *If the cell is white: if it has exactly 3 black neighbors among its 8 neighboring cells, it becomes black, otherwise it remains white.*

Remarks:

- These simple rules create a surprisingly wide range of patterns. There are stable configurations which keep their shape without changing; there are oscillators that keep repeating a few specific configurations periodically; there are “gliders” that exhibit a similar periodicity but also slowly move through the grid in the meantime. There are more complex patterns that repeatedly create smaller oscillators or gliders.
- These constructions can then be used to form gadgets on a higher abstraction level: we can create logical AND and OR gates, and ultimately a finite automaton. These tools then allow us to simulate the behavior of a TM in GoL, similarly to the tiles before.

Theorem 7.29. *Game of Life is Turing-complete.*

Remarks:

- As a result, we can also formulate some undecidable problems in this model.

Problem 7.30 (GoL Reachability)**.** *Given an initial configuration c, the task is to decide if another configuration c' will ever occur.*

Theorem 7.31. *GoL Reachability is undecidable.*

Remarks:

- GoL is a in fact a special case of a widespread model of computation on grids called Cellular Automaton.

Definition 7.32 (Cellular Automaton or CA)**.** *A Cellular Automaton consists of a (two-dimensional) grid of cells, where each cell is in a specific state. In each iteration, every cell (concurrently and independently) changes its state based on the current states of the cells in its immediate neighborhood.*

Remarks:

- If we denote the set of states by S, then a CA is essentially described by a function $f : S \times S^N \to S$ (with N denoting the size of the neighborhood). Each cell executes this function in each round to obtain the state in the next round.
- Since GoL is a special case of Cellular Automata, CAs in general are also Turing-complete.
- CAs can model various processes in natural sciences, ranging from Physics to Biology, with the cells of the automaton representing anything from chemical molecules to actual (biological) cells.
- There are many other areas (beyond halting and grids) where we can find undecidable problems. To mention another surprising example: Given a couple of $k \times k$ matrices with integer entries, it is undecidable if they can be multiplied in some order, possibly with repetitions, such that we obtain the zero matrix as a result.

7.4 Post Correspondence Problem

Finally, we discuss some variants of the so-called Post Correspondence Problem. This problem is an interesting conclusion to our whole lecture: it demonstrates that seemingly similar problems can easily have a completely different complexity.

Problem 7.33 (PCP)**.** *We have a set of dominoes, where each domino (α, β) has two words written on the domino: one word α on the top, and one word β on the bottom. Can we make a sequence of these dominoes, possibly with repetitions, such that the concatenation of words on top is the same as the concatenation of words on the bottom?*

Remarks:

- Given a finite alphabet of symbols Σ, a *word* is a finite string formed from these letters, possibly with repetitions. A concatenation of words $\alpha_1, \alpha_2, \ldots$ is the word obtained by writing these words after each other in this order.

Theorem 7.35. *PCP is undecidable.*

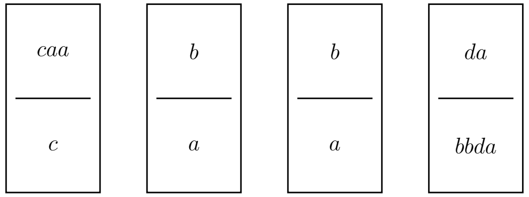

Figure 7.34: Example solution of a PCP: both the top and the bottom string is *caabbda*. The sequence consists of 4 dominoes, using one of the dominoes twice.

Proof. The proof is quite technical, so we only outline the main idea. Similarly to tiles, we can use dominoes to simulate the running of a TM. The concatenated string will describe the history of the run of the TM as a list of subsequent configurations. The bottom string is always "one step ahead" the top string in this computation; thus by defining an appropriate domino for each possible transition in the TM, we can ensure that the next configuration is always a valid follow-up to the current configuration. If the TM reaches a terminal state, then some extra dominoes ensure that the top string can catch up to the bottom string; this way the two strings become identical, and thus we have a valid sequence of dominoes that solves the PCP problem.

Since such a PCP solution exists if and only if the TM halts. Since the halting problem is undecidable, the PCP problem is also undecidable. □

Remarks:

- However, there is a simple algorithm that terminates in finite time if the answer is yes.

Theorem 7.36. *PCP is semi-decidable.*

Proof. We can enumerate all possible domino sequences based on their length in increasing order. Then if there exists a solution with a domino sequence of length k for some finite number k, then until reaching this sequence, we check at most

$$n + n^2 + ... + n^k \leq k \cdot n^k$$

possible sequences. Since each such check takes at most $O(k)$ time, we can find the solution in finite time. □

Remarks:

- The PCP problem is often used in reductions when analyzing problems related to formal languages.
- In terms of the number of dominoes used, the best known method to simulate a TM requires 5 different dominoes. This shows that if we have the correct 5 dominoes, then the problem is undecidable. However, what happens if we restrict the problem to less than 5 dominoes?

Theorem 7.37. *With only 1 domino, the PCP problem is decidable in polynomial time.*

Proof. In this case, any sequence consists of a specific number of repetitions of our single domino. In this case, the top and bottom strings are only identical if our single domino has the same word on the top and bottom side (and in this case, a single instance of the domino already provides a solution). We can easily check this in linear time: we only need to read the two words and compare them. □

Theorem 7.38. *With only 2 dominoes, the PCP problem is decidable.*

Proof. The proof of this claim is quite involved, so we do not discuss it here. □

Remarks:

- With 3 or 4 dominoes, it is still an open question whether PCP is undecidable or not.
- Another possible modification is to restrict the size of the alphabet. More specifically we need an alphabet of at least $|\Sigma| \geq 2$ letters for undecidability. If Σ only consists of a single character, then the problem becomes decidable.

Theorem 7.39. *PCP with* $|\Sigma| = 1$ *is solvable in polynomial time.*

Proof. In this case, we only need to make sure that the top and bottom words have the same length, i.e. the same number of occurrences of our single character. This means that for each available domino, we only need to consider the difference of length between the top and bottom words, which gives us a (not necessarily positive) integer. The task then reduces to analyzing this set of integers, and selecting a subset of them (with possible repetitions) that sums up to 0.

Solving this is rather easy in polynomial time. If one of the integers is 0, then this already forms a valid sequence on its own. If not, then we need to check if there is at least one positive number $x_i > 0$ and at least one negative number $x_j < 0$ among our integers: then a sequence consisting of x_i copies of the number x_j and $|x_j|$ copies of the number x_i also sums up to 0. Otherwise, all the numbers are positive (or negative); in this case, the sum of any sequence is also positive (or negative, respectively). □

Remarks:

- For another variant, we can also restrict the size of the allowed domino sequence.

Problem 7.40 (Bounded PCP)**.** *In Bounded PCP, the input also contains an integer k, and we only accept domino sequences that have length at most k.*

Theorem 7.41. *Bounded PCP is decidable but NP-hard.*

Proof. With n dominoes, we only have n^k possible domino sequences. By enumerating and checking all these possibilities, the problem is clearly decidable in finite time.

The proof of NP-hardness can be shown through a reduction from the longest common substring problem; we do not discuss it here. □

Chapter Notes

The Turing Machine was developed by Alan Turing in 1936, long before the invention of modern day computers [15]. Turing has specifically defined the model in order to study the halting problem, and prove its incomputability. The halting problem (and its different variants) has kept its central place in the area; the majority of known incomputability results are shown through a reduction that comes either directly or indirectly from this problem.

A very similar line of thought and proof technique to the halting problem's incomputability has also appeared in the work of Kurt Gödel, who was studying incompleteness theorems and the axiomatization of natural numbers at about the same time [13]. The general message of these two results has caused a large surprise (even shock) in the scientific community, where the general belief (based on Hilbert's conjectures) was that, intuitively speaking, every well-defined question can be answered. The results have shown that this is not the case, which has far-reaching philosophical consequences.

The Turing Machine has also remained the fundamental model to study computations ever since. The closely related concepts (e.g. Universal Turing Machine, Turing-completeness, or different formulations of the Church-Turing thesis) have been gradually developed and refined in the following decades. This rapidly developing area was studied by some of the most important mathematicians of the 20th century, including John von Neumann, Alonzo Church or Stephen Kleene.

The tiling problem was first discussed by Wang in 1961 [16]. However, in his first work, Wang conjectured that whenever a tiling exists, a periodic tiling also exists. A few years later his student Berger showed that some tile sets only allow an aperiodic tiling, and that the problem is undecidable due to its connection to the halting problem [3].

Game of Life was devised by John Conway in 1970 [8], and has been analyzed in numerous papers and books since then [2]. There are many simulators online where you can create different patterns and follow their development through the rounds [1].

As for Cellular Automata in general, there is an immense literature discussing different aspects of the topic. Different variants of automata have been used in a very wide range of applications, e.g. generating pseudo-random numbers in computer science [14], modeling the crystallization of snowflakes [4], modeling the geometric patterns on seashells [6] or modeling the flow of traffic on the freeway [10].

The PCP problem was introduced by Emil Post in 1946 [12]. A long line of works have followed that tried to reduce the number of dominoes required for undecidability, going down to set of 7 dominoes in 1996 [9], and then finally 5 tiles in the work of Neary in 2015 [11]. The decidability for 2 dominoes was proven by Ehrenfeucht, Karhumäki and Rozenberg [7], while the NP-hardness of Bounded PCP was first discussed in [5].

This chapter was written in collaboration with Pál András Papp.

Bibliography

[1] John Conway's Game of Life Online. `https://playgameoflife.com`.

[2] Andrew Adamatzky. *Game of life cellular automata*, volume 1. Springer, 2010.

[3] Robert Berger. *The undecidability of the domino problem*. Number 66. American Mathematical Soc., 1966.

[4] Charles D Brummitt, Hannah Delventhal, and Michael Retzlaff. Packard snowflakes on the von neumann neighborhood. *Journal of Cellular Automata*, 3(1), 2008.

[5] Robert L Constable, Harry B Hunt III, and Sartaj Sahni. On the computational complexity of scheme equivalence. Technical report, Cornell University, 1974.

[6] Stephen Coombes. The geometry and pigmentation of seashells. *Nottingham: Department of Mathematical Sciences, University of Nottingham*, 2009.

[7] Andrzej Ehrenfeucht, Juhani Karhumäki, and Grzegorz Rozenberg. The (generalized) post correspondence problem with lists consisting of two words is decidable. *Theoretical Computer Science*, 21(2):119–144, 1982.

[8] Martin Gardner. Mathematical games: The fantastic combinations of john conway's new solitaire game "life". *Scientific American*, 223(4):120–123, 1970.

[9] Yuri Matiyasevich and Geraud Senizergues. Decision problems for semi-thue systems with a few rules. In *Proceedings 11th Annual IEEE Symposium on Logic in Computer Science*, pages 523–531. IEEE, 1996.

[10] Kai Nagel and Michael Schreckenberg. A cellular automaton model for freeway traffic. *Journal de physique I*, 2(12):2221–2229, 1992.

[11] Turlough Neary. Undecidability in Binary Tag Systems and the Post Correspondence Problem for Five Pairs of Words. In *32nd International Symposium on Theoretical Aspects of Computer Science (STACS)*, volume 30 of *LIPIcs*, pages 649–661, Dagstuhl, Germany, 2015.

[12] Emil L Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946.

[13] Panu Raatikainen. Gödel's Incompleteness Theorems. In *The Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, winter 2020 edition, 2020.

[14] Marco Tomassini, Moshe Sipper, and Mathieu Perrenoud. On the generation of high-quality random numbers by two-dimensional cellular automata. *IEEE Transactions on computers*, 49(10):1146–1151, 2000.

[15] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.

[16] Hao Wang. Proving theorems by pattern recognition—ii. *Bell system technical journal*, 40(1):1–41, 1961.