Languages and Algorithms for Artificial Intelligence

Third Module

Exercise Book

Ugo Dal Lago Francesco Gavazzo

1 Mathematical Preliminaries

Exercise 1.1. What is the cardinality of the set S^n, where S is any finite alphabet? Prove your claim.

Exercise 1.2. What is the cardinality of the subset of $\{0,1\}^{2n}$ consisting of all and only the *palindrome* words? Prove your claim.

Exercise 1.3. What is the cardinality of the subset of $\{0,1\}^n$ consisting of all and only the words which have even *parity* (and the parity of a binary string is the number of occurrences of the symbol 1 inside it)? Prove your claim.

Exercise 1.4. Relate the following pair of functions (f_i, g_i) by way of $O(\cdot)$, $\Omega(\cdot)$ or $\Theta(\cdot)$ notation:

$$
\begin{aligned}
f_1(n) &= n^2 & g_1(n) &= 4n^1 + 100\log(n)\\
f_2(n) &= n\log(n) & g_2(n) &= 10n\log(\log(n))\\
f_3(n) &= 2^n n^2 & g_3(n) &= 3^n\\
f_4(n) &= 100n & g_4(n) &= \frac{1}{100}n\log(n)\\
f_5(n) &= \log^3(n) & g_5(n) &= n^{\frac{2}{3}}\\
f_6(n) &= 10^{-3}n^3 & g_6(n) &= 10^4n^3 + 10^5n^2\log^3(n)
\end{aligned}
$$

Exercise 1.5. Define appropriate encodings of the following countable sets into the set of $\{0,1\}^*$ binary strings:

- The set $\mathbb{Q}$ of all rational numbers.
- The disjoint union $\mathbb{N} \uplus \mathbb{Z}$ of the set $\mathbb{N}$ of the natural numbers and of the set $\mathbb{Z}$ of the integer numbers.
- The class of all finite, directed graphs, namely pairs of the form (V, E) where V is a finite set, and E is a subset of $V \times V$.

2 The Computational Model

Exercise 2.1. Define an efficient 1-tape Turing Machine computing the function $\mathit{inverse} : \{0,1\}^* \to \{0,1\}^*$ such that $\mathit{inverse}(x)$ is the binary string obtained by flipping all bits in x, e.g. $\mathit{inverse}(01011)$ is 10100. Give the Turing Machine explicitly as a triple in the form (Γ, Q, δ).

Exercise 2.2. Define an efficient 1-tape Turing Machine computing the successor function on the natural numbers, when natural numbers are encoded in in pure binary. In order to make your task simpler, you can safely suppose that the binary string encoding a natural number has least signifcant bits on the left and most significant bits on the right, e.g. 12 is encoded as 0011 rather than as 1100. Give the Turing Machine explicitly as a triple in the form (Γ, Q, δ).

Exercise 2.3. Do Exercise 2.2, but assume, now, that natural numbers are encoded as usual, e.g., 12 is encoded as 1100.

Exercise 2.4. A pair (x, y) of binary strings *of equal length* can be easily encoded into a single binary string in many ways. Pick one, and write $\llcorner(x, y)\lrcorner$ for the encoding of the pair. Define an efficient 3-tape Turing Machine computing the function $xor : \{0,1\}^* \to \{0,1\}^*$ such that $xor(\llcorner(x, y)\lrcorner) = x \oplus y$, where $\oplus$ is the bitwise exclusive or operator, i.e. $100 \oplus 101$ is 001. Try to give the Turing Machine explicitly as a triple in the form (Γ, Q, δ). What happens if we just allow two tapes rather than three?

Exercise 2.5. Show that the function $T(n) = 3n + 2n + 1$ is time-constructible.

Exercise 2.6. Give an example of a function T which is *not* time-constructible, and prove your claim.

3 Polynomial Time Computable Problems

Exercise 3.1. Show that the following problem is computable in polynomial time.

> Given a list $A = [a_1, \ldots, a_n]$ of natural numbers and a number $v \in \mathbb{N}$, return an index $i \in \{1, \ldots, n\}$ such that[1] $A[i] = v$, if any, and return -1 otherwise.

Exercise 3.2. Show that the following problem is computable in polynomial time.

> Sort a list $A = [a_1, \ldots, a_n]$ of natural numbers.

Hint. You do not need to be efficient: a naive sorting algorithm works fine for solving this exercise.

Exercise 3.3. Determine whether the following algorithms run in polynomial time, where for strings s_1, s_2 we denote by $s_1 :: s_2$ their concatenation. Motivate your answer.

Data: A string $s \in \{0,1\}^*$
$p \leftarrow s$;
$\ell \leftarrow |s|$;
$i \leftarrow 1$;
while $i < \ell$ **do**
  $p = p :: p$
end
return p

Data: A string $s \in \{0,1\}^*$
$p \leftarrow s$;
$\ell \leftarrow |s|$;
$i \leftarrow 1$;
while $i < \ell$ **do**
  $p = p :: s$
end
return p

Exercise 3.4. Recall that a *directed* graph is a pair $G = (V, E)$ where V is a set of vertexes and $E \subseteq V \times V$ is the 'edge' relation between vertexes. Notice that we do not require E to be symmetric. We represent graphs using the so-called adjacency matrices. Formally, we regard graphs as pairs (V, A) where $V = \{v_1, \ldots, v_n\}$ is a set of vertexes and A is an $n \times n$-matrix over $\{0, 1\}$. Intuitively, A encodes the edge relation according to the convention that $A_{i,j} = 1$ if and only if there is an edge from v_i to v_j. A universal sink is a vertex v_i such that for all $j, k \leq n$ with $j \neq i$ we have

$$A_{i,k} = 0 \quad A_{j,i} = 1$$

Notice that if a graph has a universal sink, then the latter is unique. Show that determining whether a graph has a universal sink is computable in polynomial time.

4 Between the Feasible and the Unfeasible

Exercise 4.1. Suppose that $\mathcal{L}_1, \mathcal{L}_2 \in \mathbf{NP}$, i.e., that the two languages $\mathcal{L}_1$ and $\mathcal{L}_2$ are both in the class $\mathbf{NP}$. Prove that $\mathcal{L}_1 \cap \mathcal{L}_2$ is itself in $\mathbf{NP}$. What can we say about $\mathcal{L}_1 \cup \mathcal{L}_2$?

[1]We denote by $A[i]$ the i-th element of A.

Solutions to Selected Exercises

Exercise 1.3. For every n, let E_n and O_n be the subsets of $\{0,1\}^*$ consisting of the strings of length n having even and odd parity, respectively. As an example:

$$E_3 = \{000, 011, 110, 101\};$$
$$O_3 = \{001, 010, 100, 111\}.$$

It seems that half of the strings of $\{0,1\}^*$ are in E_3 and half are in O_3. Is this a general rule. The answer is positive, and indeed, we will now prove that $|E_n| = |O_n| = 2^{n-1}$ for every $n \geq 1$. The first thing we prove is that for every such n, it holds that

$$E_n = \{x \in \{0,1\}^* \mid x = 0 \cdot y \wedge y \in E_{n-1}\} \cup \{x \in \{0,1\}^* \mid x = 1 \cdot y \wedge y \in O_{n-1}\}$$
$$O_n = \{x \in \{0,1\}^* \mid x = 0 \cdot y \wedge y \in O_{n-1}\} \cup \{x \in \{0,1\}^* \mid x = 1 \cdot y \wedge y \in E_{n-1}\}$$

Every string in E_n, if $n \geq 1$ either starts with a 0 or with a 1. In the former case, the rest of the string is itself in E_n, in the latter case, it is of course in O_n. Viceversa, any string in the form $0 \cdot y$ where $y \in E_{n-1}$ and any string in the form $1 \cdot y$, where $y \in O_{n-1}$ are in E_n. Similarly for strings in the form $0 \cdot y$ where $y \in O_{n-1}$ and any string in the form $1 \cdot y$, where $y \in E_{n-1}$ which are in O_n by definition. As a consequence, we can safely conclude that

$$|E_n| = |E_{n-1}| + |O_{n-1}|; \qquad |O_n| = |O_{n-1}| + |E_{n-1}|. \tag{1}$$

The fact that $|E_n| = |O_n| = 2^{n-1}$ for every $n \geq 1$ can thus be proved by induction on n:
- If $n = 1$, then $E_n = \{0\}$ and $O_n = \{1\}$ and the thesis holds.
- Suppose the thesis holds for n, and let us prove that if must hold for $n+1$:

$$|E_{n+1}| = |E_n| + |O_n| = 2^{n-1} + 2^{n-1} = 2 \cdot 2^{n-1} = 2^n;$$
$$|O_{n+1}| = |O_n| + |E_n| = 2^{n-1} + 2^{n-1} = 2 \cdot 2^{n-1} = 2^n.$$

In both cases, we make use of Equation (1), followed by the induction hypotheses.
Please notice how we had to prove a *stronger* result rather than the one required by the exercises, namely that *both* E_n and O_n have a given cardinality. □

Exercise 1.4. Let us consider, e.g., the functions f_4 and g_4. Recall that they are defined as follows:

$$f_4(n) = 100n \qquad g_4(n) = \frac{1}{100} n \log(n)$$

Intuitively, we expect g_4 to grow asymptotically strictly faster than f_4. Let us thus prove that f_4 is $O(g_4)$:

$$f_4(n) \leq c \cdot g_4(n) \Leftrightarrow 100n \leq \frac{c}{100} n \log(n)$$
$$\Leftrightarrow 100 \leq \frac{c}{100} \log(n) \Leftrightarrow \log(n) \geq \frac{10000}{c}$$

As a consequence, *for every* $c > 0$, any n such that $\log(n) \geq \frac{10000}{c}$ would make the inequality $f_4(n) \leq c \cdot g_4(n)$ true, which thus holds for sufficiently large n. Similarly, one can prove that $g_4 \in \Omega(f_4)$:

$$g_4(n) \geq c \cdot f_4(n) \Leftrightarrow \frac{1}{100} nlog(n) \geq 100cn \Leftrightarrow \log(n) \geq 10000c$$

As a consequence, for every c, the inequality $g_4(n) \geq c \cdot f_4(n)$ holds whenever $\log(n) \geq 10000c$, thus for sufficiently large n. □

Exercise 2.1. The alphabet Γ can be defined as $\{\triangleright, \Box, 0, 1\}$, while the set of states Q is $\{q_{\mathtt{init}}, q_s, q_r\}$. The transition function is specified as follows:

$$\begin{aligned}
(q_{\mathtt{init}}, \triangleright) &\longmapsto (q_s, \triangleright, \mathtt{S}) \\
(q_s, \triangleright) &\longmapsto (q_2, \triangleright, \mathtt{R}) \\
(q_s, 0) &\longmapsto (q_s, 1, \mathtt{R}) \\
(q_s, 1) &\longmapsto (q_s, 0, \mathtt{R}) \\
(q_s, \Box) &\longmapsto (q_r, \Box, \mathtt{S}) \\
(q_r, \Box) &\longmapsto (q_r, \Box, \mathtt{L}) \\
(q_r, 0) &\longmapsto (q_r, 0, \mathtt{L}) \\
(q_r, 1) &\longmapsto (q_r, 1, \mathtt{L}) \\
(q_r, \triangleright) &\longmapsto (q_{\mathtt{halt}}, \triangleright, \mathtt{S})
\end{aligned}$$

In all the other cases (e.g. when the state is $q_{\mathtt{init}}$ and the symbol is not $\triangleright$, the behavior of the machine is not relevant, i.e., δ can be defined arbitrarily defined. $\Box$

Exercise 3.1. First, we design an algorithm solving the desired task.

```
Data: A = [a_1, ..., a_n], v
Result: An index i ∈ {1, ..., n} s.t.
        A[i] = v, if any, −1, otherwise
i ← 1;
while i ≤ n do
    if A[i] = v then
        return i
    else
        i ← i + 1
    end
end
return −1
```

How to prove that this algorithm works in polynomial time? As seen in class, we do that in four steps.

1. We encode the input as a binary string. Our analysis of the complexity of the algorithm will be given with respect to the length (call it ℓ) of such a string.
2. We prove that the number of instructions of the algorithm is bounded by a polynomial in ℓ.
3. We argue that each instruction can be simulated by a Turing machine in polynomial time.
4. We show that all 'intermediate' data and results of the algorithm are bounded by a polynomial in ℓ.

We begin with step 1. In order to encode $A = [a_1, \ldots, a_n]$ as a binary string, we first need to encode its components a_i. For that, we can choose one standard encoding of the natural numbers in $\{0,1\}^*$. Let us write $\llcorner a_i \lrcorner$ for the encoding of a_i in binary. Since using n bits we can encode $2^n - 1$ (natural) numbers, the encoding of a number $a \in \mathbb{N}$ requires $\log a + 1$ bits[2]. Therefore, we have $|\llcorner a_i \lrcorner| = \log a_i + 1$. The next step is to understand how to encode the whole A. For that, we regard a list of elements $[b_1, \ldots, b_n]$ as a 'pair of pairs' of the form $(((b_1, b_2), b_3), \ldots b_n)$. Recall that given a pair (b_1, b_2) of bitstrings, we can define the string $\llcorner(b_1, b_2)\lrcorner \in \{0,1\}^*$ by first translating (b_1, b_2) as the string $b_1\#b_2 \in \{0,1,\#\}$ and then encoding $b_1\#b_2$ as a string $\llcorner(b_1, b_2)\lrcorner \in \{0,1\}^*$. For the latter point, we simply map 0 to 00, 1 to 11, and # to 01. As a consequence, we see that $|\llcorner(b_1, b_2)\lrcorner| = 2|b_1| + 2|b_2| + 2$. Now, given a list $[b_1, \ldots, b_n]$ of bitstring by regarding it as a pair

[2]Actually, we should take the floor of $\log a$.

$(((b_1, b_2), b_3), \ldots b_n)$, we see that we obtain an encoding $\llcorner[b_1, \ldots, b_n]\lrcorner$ of length $\sum_{i=1}^n 2|b_i|+2(n-1)$. Applying these general considerations to A (and recalling that $|\llcorner a_i \lrcorner| = \log a_i + 1$), we obtain:

$$\llcorner A \lrcorner = \llcorner[\llcorner a_1 \lrcorner, \ldots, \llcorner a_n \lrcorner]\lrcorner \qquad |\llcorner A \lrcorner| = \sum_{i=1}^{n} 2(\log a_i + 1) + 2(n-1)$$

Finally, we pair A with v (recall that both A and v are input of our algorithm), so $\llcorner(\llcorner A \lrcorner, \llcorner v \lrcorner)\lrcorner$ gives an encoding of the input of our algorithm in binary notation. Notice that

$$\ell = |\llcorner(\llcorner A \lrcorner, \llcorner v \lrcorner)\lrcorner| = 2(\sum_{i=1}^{n} 2(\log a_i + 1) + 2(n-1)) + 2(\log v + 1) + 2.$$

We now move to step 2. The latter is straightforward. Our algorithm consists of:

- 1 assignment ($i \leftarrow 1$).
- n iteration of:
  - An inequality check ($i \leq n$).
  - A conditional branching performing:
    - An equality check ($A[i] = v$),
    - Either a return instruction or an assignment ($i \leftarrow i + 1$).
- 1 return instruction

Therefore, the number of the instruction is of the form $b + c \cdot n$, for suitable constants b, c, and thus it is bounded by a polynomial in ℓ. In order to prove step 3, we have to argue that all the aforementioned instructions can be simulated by a TM in polynomial time. For instance, an equality check can be simulated as follows. Say we have two values a and b stored in different portions of a tape of a TM. In order to check whether a is equal to b, the machine simply moves back and forth between a and b checking whether they are bitwise equal. This can be done in polynomial time with respect to the length of a and b, provided that the 'distance' between a and b in the tape is itself bounded by a polynomial in the length of a and b. This will be indeed ensured by step 4. Similar arguments can be used to show that all other instructions can be simulated efficiently by a TM.

Finally, in order to prove step 4 we simply observe that the only intermediate value computed by our algorithm is i, which can be at most n (and therefore it is bounded by a polynomial in ℓ). □

Exercise 3.3. In order to prove that the two algorithms run in polynomial time we have to follow the four steps of previous exercise. These are mostly straightforward. In fact, step 1 is trivial, as the input s is already in binary. For step 2 we simply observe that we have ℓ iterations, and that overall the number of instructions is of the form $b + c \cdot \ell$, for suitable constants b, c, and thus polynomial in ℓ. Moreover, it is not hard to see that all the instructions can be easily simulated by a TM. What goes wrong is step 4. In fact, in the first algorithm at each iteration we concatenate p with itself. That means that if before entering into the while-loop $|p| = \ell$, then after one iteration we will have

$$|p| \text{ after iteration } 1 = 2(|p| \text{ at iteration } 0) = 2\ell$$

Similarly, we obtain

$$\begin{aligned} |p| \text{ after iteration } 2 &= 2|p|(\text{ at iteration } 1) = 4\ell \\ |p| \text{ after iteration } 3 &= 2|(p| \text{ at iteration } 2) = 8\ell \\ &\vdots \end{aligned}$$

and thus

$$|p| \text{ after iteration } n = 2(|p| \text{ at iteration } n-1) = 2^n \ell$$

That means that the length of p is exponential in ℓ, and therefore the first algorithm cannot run in polynomial time. Notice that this is not true for the second algorithm, where we have $|p| \approx \ell^2$. □

Exercise 3.4. Let $G = (V, A)$ be a graph with $V = \{v_1, \ldots, v_n\}$. We design an algorithm for determining whether G has a (necessarily unique) universal sink. The key observation is noticing that if v_i is a universal sink, then the i-th row in A contains only 0s, whereas the i-th column contains all 1s (except in the entry $A_{i,i}$). Graphically:

$$\begin{array}{cc} & \begin{array}{cccccc} & & & i & & \end{array} \\ \begin{array}{c} \\ \\ \\ i \\ \\ \\ \end{array} & \begin{pmatrix} & & & 1 & & \\ & & & 1 & & \\ & & & 1 & & \\ 0 & 0 & 0 & 0 & 0 & 0 \\ & & & 1 & & \\ & & & 1 & & \end{pmatrix} \end{array}$$

Let us write $US(v)$ for the property "v is a universal sink". Then, we see that for all vertexes $v_j, v_k \in V$ we have that

$$A_{j,k} = 1 \implies \neg US(v_j);$$
$$A_{j,k} = 0 \implies \neg US(v_k).$$

We can thus proceed as follows. Let POS be a list of potential universal sinks. Obviously, we begin assuming $POS = V$. We sequentially pick pairs of vertexes (v_i, v_j) in POS and look at $A_{j,k}$. If $A_{j,k} = 1$, then we know that v_j cannot be universal sink, and thus we remove it from POS. Otherwise, $A_{j,k} = 0$ meaning that v_k cannot be universal sink, and thus we remove it from POS. Proceeding this way, we will end up with POS containing a single vertex v_i. We then check whether v_i is universal sink by checking whether the i-th row of A contains 0s only, and whether the i-th column of A contains all 1s (except for $A_{i,i}$). If we succeed then we v_i is a universal sink. Otherwise, there is no universal sink.

```
Data: V = [v_1, ..., v_n], A
Result: An index i ∈ {1, ..., n} s.t.
        US(v_i), if any, −1, otherwise
POS ← [1, ..., n];
// We use only labels of vertexes
i ← 1;
j ← 2;
while j ≤ n do
    if A_{i,j} = 1 then
        POS = POS.remove(i);
        i ← j;
        j ← j + 1;
    else
        POS = POS.remove(j);
        j ← j + 1;
    end
end
i = POS.fst;
// We have POS = [i] for some i
// Check row
j ← 1;
while j ≤ n do
    if A_{i,j} = 0 then
        j ← j + 1;
    else
        return −1
    end
end
// Check column
j ← 1;
while j ≤ n do
    if A_{j,i} = 1 or j = i then
        j ← j + 1;
    else
        return −1
    end
end
return i
```

Notice that we might have been more efficient in checking columns and rows. However, our implementation is closer to what we would do when programming a TM and makes our analysis easier.

We now show that the above algorithm works in polynomial time. First, the encoding of the input is standard. The input, in fact, consists of a natural number n (the number of vertexes[3]) and of an $n \times n$-matrix of bits. We encode the latter as a list made of n elements each of which consisting of n bits. Therefore, such an encoding has length $2n^2 + n - 1$. Pairing the latter with the encoding of n (which has length $\log n$), we obtain an input of length $2(2n^2 + n - 1) + 2\log n + 1$.

How many instructions our algorithm has? The first loop consists of $n - 1$ iterations, each of which essentially consists of assignments, checking values of the matrix, and removing elements from a list. After that, we have loops for checking row and columns, which consist of n iteration each (and thus a total of $2n$ iterations). Summing up, we have $3n - 1$ iterations, plus a fixed number of equality checking, assignments (plus arithmetic operations), and basic operations on lists. The algorithm thus has running time $O(n)$, and thus it is polynomial in $2(2n^2 + n - 1) + 2\log n + 1$.

[3] Recall that we actually work with labels $1, \ldots, n$ of vertexes, rather than with vertexes themselves.

The last two points to prove in order to conclude that our algorithm runs in polynomial time are showing that each instructions can be simulated by a TM in polynomial time, and that all intermediate values/results have length polynomially bounded by the length of the input. The latter point is straightforward, whereas for the former we essentially proceed as in previous exercises. $\square$

Exercise 4.1. By the fact that $\mathcal{L}_1 \in \mathbf{NP}$ and $\mathcal{L}_2 \in \mathbf{NP}$, we know that there are polynomials p_1, p_2 and polytime TMs $\mathcal{M}_1, \mathcal{M}_2$ such that:

$$\mathcal{L}_1 = \{x \in \{0,1\}^* \mid \exists y_1 \in \{0,1\}^{p_1(|x|)}.\mathcal{M}_1(x, y_1) = 1\};$$
$$\mathcal{L}_2 = \{x \in \{0,1\}^* \mid \exists y_2 \in \{0,1\}^{p_2(|x|)}.\mathcal{M}_2(x, y_2) = 1\}.$$

Now, let $\mathcal{P}$ be the TM which, on input (x, y_1, y_2), simulates $\mathcal{M}_1$ on input (x, y_1) and $\mathcal{M}_2$ on input (x, y_2), and returns 1 iff *both* return 1. Clearly, $\mathcal{P}$ works in polynomial time. Now:

$$\begin{aligned}
\mathcal{L}_1 \cap \mathcal{L}_2 &= \{x \in \{0,1\}^* \mid (\exists y_1 \in \{0,1\}^{p_1(|x|)}.\mathcal{M}_1(x, y_1) = 1) \wedge (\exists y_2 \in \{0,1\}^{p_2(|x|)}.\mathcal{M}_2(x, y_2) = 1)\};\\
&= \{x \in \{0,1\}^* \mid \exists y_1 \cdot y_2 \in \{0,1\}^{p_1(|x|)+p_2(|x|)}.(\mathcal{M}_1(x, y_1) = 1) \wedge (\mathcal{M}_2(x, y_2) = 1)\};\\
&= \{x \in \{0,1\}^* \mid \exists y_1 \cdot y_2 \in \{0,1\}^{p_1(|x|)+p_2(|x|)}.(\mathcal{P}(x, y_1, y_2) = 1)\}.
\end{aligned}$$

In other words, the class **NP** is closed by intersections. Closure by unions can proved similarly, but not exactly in the same way. In particular, from $\mathcal{M}_1$ and $\mathcal{M}_2$, one can form another machine $\mathcal{Q}$, different from $\mathcal{P}$, which on input (x, y) (where $y \in \{0,1\}^{\max\{p_1(|x|), p_2(|x|)\}}$), simulates $\mathcal{M}_1(x, y_1)$ and $\mathcal{M}_2(x, y_2)$, (where y_1, y_2 are prefixes of y of lengths $p_1(|x|)$ and $p_2(|x|)$, respectively) and returns 1 iff *either one or the other* return 1. Clearly, $\mathcal{Q}$ works in polynomial time. Now:

$$\begin{aligned}
\mathcal{L}_1 \cup \mathcal{L}_2 &= \{x \in \{0,1\}^* \mid (\exists y_1 \in \{0,1\}^{p_1(|x|)}.\mathcal{M}_1(x, y_1) = 1) \vee (\exists y_2 \in \{0,1\}^{p_2(|x|)}.\mathcal{M}_2(x, y_2) = 1)\};\\
&= \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{\max\{p_1(|x|), p_2(|x|)\}}.(\mathcal{M}_1(x, y_1) = 1) \vee (\mathcal{M}_2(x, y_2) = 1)\};\\
&= \{x \in \{0,1\}^* \mid \exists y \in \{0,1\}^{\max\{p_1(|x|), p_2(|x|)\}}.(\mathcal{Q}(x, y) = 1)\}.
\end{aligned}$$

$\square$