On Computability 1

Darius Lorek

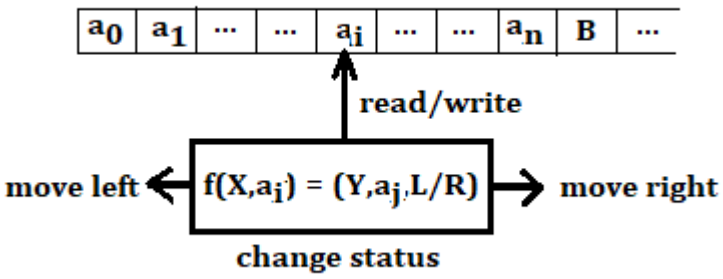
December 30, 2022

Abstract

In this first document on computability, we introduce the constraint satisfaction model of computation and show how powerful it is for both the theoretical considerations and the practical solutions of specific configuration and matching problems. First of all, we recall some of the elementary concepts of computer science. A basic level of understanding of these concepts is prerequisite and will be assumed. Detailed information can be obtained from internet or from the relevant literature - we will specify some references to it.

1 Turing Machines

A Turing Machine (TM) is a theoretical model of computation and can recognize the widest class of formal languages: the recursively enumerable languages.
The idea behind it is, that a TM, which is composed of a strip of tape divided into discrete cells, each of which can hold a single symbol drawn from a finite set of symbols of an alphabet, and a r/w head, positioned over these cells, which has a state selected from a finite set of states can read/write from/into these cells and then change the state acc. to a transition function, which determines what to do for each combination of the current state and the symbol that is read, and move then one cell to the right or to the left. Doing so a TM can recognize, if a word written on the strip at the beginning of the computation belongs to a defined formal language. We can implement every algorithm in this relatively simply computational model, compute every computable function. As a first step in this blog, we show that other equivalent models of computation can realise a more convenient handling of problems like pattern recognition. Once defined we choose one of these models to implement a concrete pattern recognizer for concrete human languages.



a simple TM (a_0, …, a_n alphabet symbols; B blank; X, Y status; L/R left/right)

The tape above is finite to the left and endless to the right of the head. It can be proven, that an unrestricted tape for both sides doesn't increase the computability class of a TM, like the class of recognizable formal languages. The same holds for the usage of more than one tape. All these extended TMs can be simulated by a TM described above. We give now a formal definition. A Turing Machine (TM) is a 7-tuple (Q, Σ, Γ, f, q_0, B, F) where

Q is a finite set of head states
Σ is a finite set of valid input symbols, a subset of Γ

Γ is a finite set of valid tape symbols
f is the transition function $Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\}$, can be undefined for some arguments
$\mathbf{q_0}$ from Q is the start state
B is the blank symbol, a member of Γ
F is a set of end states, $F \subseteq Q$

We need to mention, that time as resource doesn't matter in these preliminary considerations.

2 Equivalence between Type-0 formal grammars and Turing Machines

Typo-0 grammars include all formal grammars and thus is the widest class of formal grammars. They generate all languages that can be recognized by a TM. They are defined as a 4-tuple (V, T, P, S) where

V is a finite set of variables
T is a finite set of terminal symbols (V and T are disjunct)
P is a finite set of productions of the form $A \in V \rightarrow \alpha \in (V \cup T)^*$
S is a member of V, the start symbol

Example. With that definition we can define the set of arithmetic expressions with the operators {+, -, ∗, /} and with additional terminal symbols {(,), id (as a symbol for an operand)} and the productions:

$$P := \{E \rightarrow E + E, E \rightarrow E - E, E \rightarrow E * E, E \rightarrow E / E, E \rightarrow (E), E \rightarrow id\}$$

The last production of P states, that a single operand is an expression. This grammar can be then defined as ({E}, {+, -, ∗, /, (,), id}, P, E) and we would be able to generate arithmetic expressions like:

$$E \Longrightarrow E * E \Longrightarrow (E) * E \Longrightarrow (E) * id \Longrightarrow (E + E) * id \Longrightarrow (E + id) * id \Longrightarrow ((E) + id) * id$$
$$\Longrightarrow ((E - E) + id) * id \Longrightarrow ((E - id) + id) * id \Longrightarrow ((id - id) + id) * id$$

The last derived word doesn't contain any variables, so it is a valid word of the language defined above (the language of arithmetic expressions). For convenience we will use a shorter representation of the production set as:

$$P := \{E \rightarrow E + E \mid E - E \mid E * E \mid E / E \mid (E) \mid id\}$$

Example. A grammar, which generates the set $\{a^{2^n} \mid n \in N, n \geq 1\}$ (2^n the positive power of 2) can be defined as ({A, B, C, D, E, S}, {a, ϵ}, P, S)} with:

$P := \{S \rightarrow ACaB, Ca \rightarrow aaC, CB \rightarrow DB \mid E, aD \rightarrow Da, AD \rightarrow AC, aE \rightarrow Ea, AE \rightarrow \epsilon\}$

ϵ is here a special symbol representing the blank. We recommend to generate a few words with this production set to understand the generation mechanism.

We owe the proof that both models, TMs and the Type-0 formal grammars, are equivalent. If L = L(G) is generated for a non-restricted grammar G = (V, T, P, S), then L is a recursive enumerable language.

Proof. We use a 2-tape TM M (a 2-tape TM can be simulated by a 1-tape TM), which recognizes L(G). The first tape is an input tape with a string ω, which we want to examine. The second tape we use for the expression α of G, which represents the current state of the word generation. At the beginning M initialises α with S. Then M repeats following steps:

1. A position i will be chosen within α in a non-deterministic way ($1 \leq i \leq |\alpha|$)
2. From the set of productions, a production $\beta \rightarrow \gamma$ will be selected non-deterministically

3. If β exists from position i in α, so β will be replaced by γ which creates a new α (if | ß | < | γ | then additional place on the tape will be arranged moving the symbols right of β sufficient many cells to the right, if | γ | < | ß | the new blanks right of γ after replacement of ß by γ will be erased moving the rest string sufficient many cells to the left. This can always be achieved by the use of additional states in the head of M)
4. Then the new α will be compared with ω. If they are equal ω will be accepted as belonging to L(G). If not, we proceed with step 1

Clear, on the tape 2 occur only expressions of G, so M will accept if $\omega \in L(G)$. Thus $L(G) \subseteq L(M)$ is recursive enumerable. □

For the other direction (if L is a recursive enumerable language than L = L(G) for a non-restricted grammar G) we introduce a new symbol $\overset{*}{\Longrightarrow}$ which represents the transitive closure of a single derivation $\Longrightarrow$. $E \overset{*}{\Longrightarrow} \alpha$ means that we can generate α starting from E in one or many steps.

Proof. L is accepted by the TM $M = (Q, \Sigma, \Gamma, f, q_0, B, F)$. We will construct a grammar G, which generates non-deterministic a word from Σ^* in the form $[a_1, a_1]\ldots[a_n, a_n]$, where the first components of the 2-tuples don't change and the actions of M will be simulated on the second components. If M accepts the word $a_1\ldots a_n$ then the grammar will transfer the word of the form $[a_1, a_1]\ldots[a_n, a_n]$ into a word consisting only of terminal symbols. If M doesn't accept then the derivation doesn't end in a terminal word. A non-deterministic generation ensures that any arbitrary word can be examined.
G is defined as $G = (V, \Sigma, P, A_1)$ where $V = ((\Sigma \cup \{\epsilon\}) \times \Gamma) \cup Q \cup \{A_1, A_2, A_3\}$ and P is a set of following productions:

(1) $A_1 \rightarrow q_0A_2$
(2) $A_2 \rightarrow q_0[a, a]A_2$ for all $a \in \Sigma \mid A_3$
(3) $A_3 \rightarrow [\epsilon, B]A_3 \mid \epsilon$
(4) $q[a, X] \rightarrow [a, Y]p$ for all $a \in \Sigma \cup \{\epsilon\}$, all $q \in Q$ and $X, Y \in \Gamma$ with $f(q, X) = (p, Y, R)$
(5) $[b, Z]q[a, X] \rightarrow p[b, Z][a, Y]$ for all $a, b \in \Sigma \cup \{\epsilon\}$, all $q \in Q$ and $X, Y, Z \in \Gamma$ with $f(q, X) = (p, Y, L)$
(6) $[a, X]q \rightarrow qaq, q[a, X] \rightarrow qaq, q \rightarrow \epsilon$ for all $a \in \Sigma \cup \{\epsilon\}$, all $q \in F$ and $X \in \Gamma$

The above definition seems more complicated than it really is. Productions (1), (2) and (3) serve as generator for a word to be examined. (2) generates the word in the form of $[a_1, a_1]\ldots[a_n, a_n]$ and (3) "allocates" space right of the word, which is needed to simulate M. (4) and (5) simulate the transition function f with the move of the head one cell to the right (4) or to the left (5). (6) converts the expression to a terminal word in case M accepted $a_1\ldots a_n$, which means the final state of M is from the set of end states F.
We assume that M accepts the word $a_1\ldots a_n$ and uses m cells right of the word in this process. After execution of the productions (1), (2) and (3) we get:

$A_1 \overset{*}{\Longrightarrow} q_0[a_1, a_1]\ldots[a_n, a_n][\epsilon, B]^m$

Now we can apply (4) and (5) in simulation of M until an end state will be generated. We keep the original word in the first components of the 2-tupels, so the last step is to extract this terminal word out of the expression in case $q \in F \subseteq Q$. For $q \notin F$ these productions (6) don't exist, so the terminal word can't be extracted. We can show by the induction over the step number of M processing that:

if $\mathbf{q_0}a_1\ldots a_n \overset{*}{\Longrightarrow}^{(M)} X_1\ldots X_{r-1}\mathbf{q}X_r\ldots X_s$
then $\mathbf{q_0}[a_1, a_1]\ldots[a_n, a_n][\epsilon, B]^m \overset{*}{\Longrightarrow}^{(G)} [a_1, X_1]\ldots[a_{r-1}, X_{r-1}]\mathbf{q}[a_r, X_r]\ldots[a_{n+m}, X_{n+m}]$ (2.1)

where $a_1,\ldots,a_n \in \Sigma$, $a_{n+1} = a_{n+2} = \ldots = a_{n+m} = \epsilon$, $X_1\ldots X_{r-1}X_r\ldots X_s \in \Gamma$, $X_{s+1} = X_{s+2} = \ldots = X_{n+m} = B$
The induction hypothesis is trivially true for 0-movement (r = 1, s = n). We assume, that 2.1 holds for k-1 movements and:

$\mathbf{q_0}a_1\ldots a_n \overset{k-1}{\Longrightarrow}{}^{(M)} X_1\ldots X_{r-1}\mathbf{q}X_r\ldots X_s \Rightarrow^{(M)} Y_1\ldots Y_{t-1}\mathbf{p}Y_t\ldots Y_u$

If $t = r+1$ so the head movement was to the right, so $f(q, X_r) = (p, Y_r, R)$. But production (4) of G says, that $\mathbf{q}[a, X] \rightarrow [a, Y]\mathbf{p}$ so:

$$\mathbf{q_0}[a_1, a_1]\ldots[a_n, a_n][\epsilon, B]^m \overset{*}{=>}{}^{(G)} [a_1, Y_1]\ldots[a_{t-1}, Y_{t-1}]\mathbf{p}[a_t, Y_t]\ldots[a_{n+m}, Y_{n+m}] \quad (2.2)$$

with $Y_i = B$ for $i > u$.

If $t = r-1$ so the head movement was to the left, so $f(q, X_r) = (p, Y_r, L)$ with $r > 1$. But production (5) of G says, that $[b, Z]\mathbf{q}[a, X] \rightarrow \mathbf{p}[b, Z][a, Y]$ so again:

$$\mathbf{q_0}[a_1, a_1]\ldots[a_n, a_n][\epsilon, B]^m \overset{*}{=>}{}^{(G)} [a_1, Y_1]\ldots[a_{t-1}, Y_{t-1}]\mathbf{p}[a_t, Y_t]\ldots[a_{n+m}, Y_{n+m}] \quad (2.2)$$

with $Y_i = B$ for $i > u$.

Applying production (6) of G we get:

$$[a_1, Y_1]\ldots[a_{t-1}, Y_{t-1}]\mathbf{p}[a_t, Y_t]\ldots[a_{n+m}, Y_{n+m}] \overset{*}{=>}{}^{(G)} a_1,\ldots,a_n \quad \text{if } p \in F.$$

Thus, we have shown, that $A_1 \overset{*}{=>}{}^{(G)} \omega$ is true, if $\omega \in L(M)$ and so $L(M) \subseteq L(G)$. With the other proof direction, we obtain $L(M) = L(G)$. □

Both computation models are then equivalent. However, both are somehow unhandy when it comes to practical computation. We introduce two another computational models, which are equivalent to Type-0 formal grammars and Turing Machines and which are suitable in use with regard to pattern or string recognition and configuration problems.

3 Equivalence between WHILE computation and Turing Machines

We define now inductively WHILE programs as:

$x_i := x_j \pm c$ is a WHILE program for every $i, j, c \in N_0$ (addition/subtraction)
If P_1, P_2 are WHILE programs, then **P_1; P_2** is a WHILE program (composition)
If P is a WHILE program, then **WHILE $x_i \neq 0$ DO P END** for every $i \in N_0$ (WHILE loop)

Strings built from a finite set of input symbols Σ can always be considered as numerical values of basis $|\Sigma|$ ($|\Sigma|$ the number of members in Σ). The replacement of a character $a_{i(old)}$ by the character $a_{i(new)}$ in a string $s := 'a_0\ldots a_{i(old)}\ldots a_n' \rightarrow 'a_0\ldots a_{i(new)}\ldots a_n'$, $i \in \{0, .., n\}$ can then be seen as an numerical operation (we take thereby a_0 as the most right digit):

$$\mathbf{s_{num} := s_{num} - y_{i(old)} * |\Sigma|^i;\ s_{num} := s_{num} + y_{i(new)} * |\Sigma|^i}$$

and multiplication can be naturally represented as:

$y_0 * x_0$ ($y_0, x_0 \in N_0$): $c := x_0$; $x_0 := 0$; WHILE $y_0 \neq 0$ DO $x_0 := x_0 + c$; $y_0 := y_0 - 1$ END; res := x_0

which holds even for decimal numbers, when we strip off the decimal point from the decimal numbers, multiply them as natural numbers and set the decimal point in the result in due place.

Example. The change of string s := '837**7**5021' → '837**1**5021' as a natural number a = int(s), $a \in \{0, .., 9\}^*$ we trivially build two sums:

$s_{num} := 837\mathbf{7}5021 - 7 * 10^4 = 837\mathbf{0}5021$; $s_{num} := 837\mathbf{0}5021 + 1 * 10^4 = 837\mathbf{1}5021$

Definition. IF clause (**IF $y_0 \neq 0$ THEN P_1 ELSE P_2**) can be simulated by a WHILE program as:

z := y_0;
WHILE $y_0 \neq 0$ DO P_1; y_0:= 0 END;
y_0 := z;
WHILE $y_0 = 0$ DO P_2; y_0:= 1 END;
y_0 := z;

with the restriction, that P_1 or P_2 don't manipulate z inside P_1 or P_2.

We proof now the equivalence of the WHILE computation and Turing Machines.

Proof. Word ω is accepted by the TM M := (Q, Σ, Γ, f, q_0, B, F) defined like above. We show that M can then be simulated by the WHILE program W as follows:

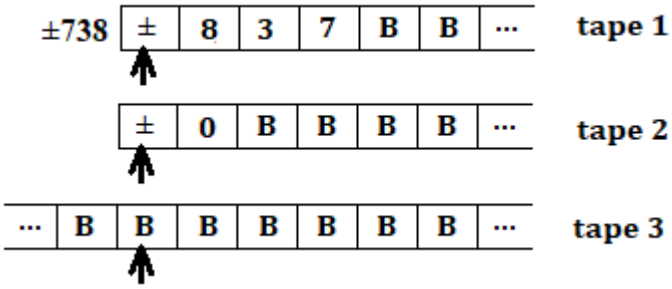
```
ω = a1…anBBB…;                    - (a1,..,an ∈ Σ)
y := num(ω);                      - y of type integer, not limited
stat := q0;                       - start status
i := 1;                           - head on position 1 at the beginning of computation
running := 1;
WHILE running ≠ 0 DO
        e := y mod | Σ |^i;
        c := 0;
        WHILE e ≥ | Σ |^(i-1) DO e := e - | Σ |^(i-1); c := c + 1 END;          (3.1)

        d := f_char(stat, c);
        y := y - c * | Σ |^i;
        y := y + d * | Σ |^i;
        statNew := f_stat(stat, c);
        j := f_step(stat, c);                                                   (3.2)
        IF i > 1 THEN i := i + j; ELSE IF  j ≠ -1 THEN i := i + j;              (3.3)
        stat := statNew;

        IF stat ∈ F THEN running := 0                                           (3.4)
        END
```

At the beginning of the computation, we convert the word ω into a numerical value of basis | Σ | and assign this value to the variable y. The variable y isn't limited in space so we can assign any value to it. Then we manipulate this value according to the processing of M. The strip of tape in M is traditionally restricted to the left. We follow this tradition so the conversion includes mirroring of ω and thus the most left cell i = 1 of M represents the rightest digit of y. At the beginning all cells right of i = n are blank (B) and the head is placed on cell i = 1.
The statements in the outer WHILE loop until (3.1) serve to extract the value of cell i resp. the digit in position i, which is assigned to the variable c. The transition function f of M is divided into three parts, which yield the new value of cell i according to f_{char}, the new head status statNew according to f_{stat} and the value j ∈ {-1, 0, 1}, which represents the movement of the head to the left, no movement or movement to the right, according to f_{step} (3.2). (3.3) calculates the new head position i and (3.4) checks if the new status stat ∈ F and terminates the computation in positive case. □

During the computation the r/w head of M can move over cell n and use the cells right of it. Therefore, y can't be limited to a highest value. M prevents a position of the r/w head left of the first cell - that is ensured for W by (3.3). The modulo function can easily be implemented with a WHILE loop. Eventually it should be noted, that WHILE programs do not always terminate and that the transition function f divided into f_{char}, f_{stat} and f_{step} can be undefined for some inputs – analogue f in M. We still have to proof the other direction.

Proof. Natural numbers can be represented in different ways on a tape stripe of a TM. We assume that W works with decimal numbers. However, any other basis for representation of

numbers would exact an analogue approach. We use a three tapes TM M_3, each for a different task, in the course of repeated addition/subtraction processing. We know, that our three-tapes Turing Machine M_3 can be simulated by a one-tape Turing Machine M as defined above[1]. On tape 1 we keep the current number to add/subtract in the decimal representation (with sign), but we reverse the digit order, so the first digit to the right of the sign is the digit of the lowest arity, as we use a left-limited tape (see below). On tape 2 we keep the power of 10 value corresponding to the digit position in the number processed on tape 1. We keep this value in unary representation (therefore **0** for the lowest digit, **0000000000** for the second lowest and so on). We put the sign of the number being processed in the cell one on tape 2. On tape 3 we keep the current sum in unary representation, so that the number of nulls (**0**) represents the current intermediate result of our addition/subtraction. At the beginning of the whole computation process the tape 3 is empty.
The initial allocations on the tapes are then as follows (see example for the number ±738; the r/w heads are at cells 1 for tape 1 or 2 and at an arbitrary cell for tape 3):



Following transition function f adds the decimal number on tape 1 to the current unary sum on tape 3 leaving the new sum on tape 3:

Tape 1 (main loop addition/subtraction, initial state q_I)

$f(q_I, +) = (q_R, +, R)$	move to the right
$f(q_R, X) = (u_R, X\text{-}1, n)$ for $X \in \{1, \ldots, 9\}$	e.g. $f(q_R, 8) = (u_R, 7, n)$; **n** no head move
…	
$f(q_R, 0) = (u_I, 0, R)$	move to the next cell to the right

Tape 1 (main loop subtraction, initial state q_I):

$f(q_I, -) = (q_L, -, R)$	move to the right
$f(q_L, X) = (u_L, X\text{-}1, n)$ for $X \in \{1, \ldots, 9\}$	e.g. $f(q_L, 8) = (u_L, 7, n)$; **n** no head move
…	
$f(q_L, 0) = (u_I, 0, R)$	move to the next cell to the right

Insert a new number to be added/subtracted:
$f(q_R, B) \rightarrow$ put new number $\pm y = num(\omega)$ on tape 1, put ±0 on tape2, move to position 1
$f(q_L, B) \rightarrow$ put new number $\pm y = num(\omega)$ on tape 1, put ±0 on tape2, move to position 1

[1] see e. g. in J. E. Hopcroft, J. D. Ullman: *Introduction to Automata Theory, Languages, And Computation* for a proof, that multi-tape TMs and TMs with tapes unlimited on both sides can be simulated by a "standard" TM as defined above.

Check condition after every loop:
IF tape 3 = <empty> THEN state = q_F and accept ELSE state = q_I

On tape 2 we generate and keep 10^i of **0**s, where $i \in N_0$ is the position of the digit processed on tape 1 (the most left position is i=0). Beginning with the first digit, each time we move one position to the right $i \rightarrow i+1$ on tape 1 we create 10^{i+1} of **0**s on tape 2. Remark[2]

Tape 2 (creation of 10^{i+1} of **0**s to the right of the sign cell in the situation, that we have currently 10^i **0**s on the tape 2)

In the first processing block we set a boundary sign at the beginning (#) and the end (**$**) of the current string of **0**s, so the first **0** will be replaced by # and the last by **$**: **±00…00 → ±#0…0$**

$f(u_I, +) = (u_I, +, R)$	
$f(u_I, -) = (u_I, -, R)$	
$f(u_I, 0) = (u_K, \#, R)$	replace first **0** by #
$f(u_K, 0) = (u_K, 0, R)$	move to the right over the **0**s until
$f(u_K, B) = (u_M, B, L)$	blank is reached, then move to the left
$f(u_M, 0) = (u_{\#}, \$, n)$	replace last **0** by **$**
$f(u_M, \#) = (u_9, \$, R)$	in the initial case with only one **0** moving left we find here the already set #, replace then # by **$** so ±# → ±$

Beginning at the cell with **#**, which we change to **0**, we make 9 additional steps to the right, over existing **0**s or creating new **0**s, in case the cell is still blank **B,** and then set the # mark at the 10^{th} position. We don't overwrite the **$** mark

$f(u_X, 0) = (u_{X-1}, 0, R)$ for $X \in \{1, \ldots, 9\}$	e.g. $f(u_8, 0) = (u_7, 0, R)$
…	
$f(u_X, B) = (u_{X-1}, 0, R)$ for $X \in \{1, \ldots, 9\}$	e.g. $f(u_8, B) = (u_7, 0, R)$
…	
$f(u_X, \$) = (u_{X-1}, \$, R)$ for $X \in \{1, \ldots, 9\}$	we don't change the **$** mark in the string

After these 9 steps we reach the status u_0 and depending on the current cell content we

$f(u_0, 0) = (u_\$, \#, n)$	replace **0** by # and change the state to $u_\$$
$f(u_0, B) = (u_\$, \#, n)$	replace **B** by # and change the state to $u_\$$
$f(u_0, \$) = (u_R, \#, L)$	replace **$** by #, reach state u_R and
$f(u_R, 0) = (u_M, \$, R)$	place **$** left of # so **..00$0..** → **..0$#0**..

Seach for the **$** mark and place the r/w head on this cell

$f(u_\$, X) = (u_\$, X, R)$ for any X but B	find the first blank to the right
$f(u_\$, B) = (u_{F\$}, B, n)$	
$f(u_{F\$}, X) = (u_{F\$}, X, L)$ for any X but $	then find **$** moving to the left
$f(u_{F\$}, \$) = (u_W, \$, n)$	change the state to u_W

Seach for the # mark and place the r/w head on this cell

$f(u_{\#}, X) = (u_{\#}, X, R)$ for any X but B	find the first blank to the right
$f(u_{\#}, B) = (u_{F\#}, B, n)$	
$f(u_{F\#}, X) = (u_{F\#}, X, L)$ for any X but #	then find # going to the left

[2] we leave it to the reader to write out the function f for the check if tape 3 is empty and to prepare tape 1 and tape 2 for the next addition/subtraction in case we didn't reach the final state q_F in the current loop.

$f(u_{F\#}, \#) = (u_9, 0, R)$ replace **#** by **0**, change the state to u_9 (repeat the 9 step process)

Once we have found **$** we move it one cell to the left, if we reach the first cell with ± the creation process of 10^{i+1} **0**s on tape 2 ends

$f(u_W, \$) = (u_W, 0, L)$
$f(u_W, 0) = (u_\#, \$, n)$
$f(u_W, \#) = (u_9, \$, R)$ **..0#$..** -> **..0$0..**
$f(u_W, +) = (u_Z, +, R)$
$f(u_W, -) = (u_Z, -, R)$

$f(u_Z, 0) = (u_Z, 0, R)$ move to the end of the string on tape 2 and
$f(u_Z, \#) = (u_V, 0, n)$ replace **#** by **0** at the end of the string
$f(u_V, 0) = (u_V, 0, L)$ move to the begin of the string and
$f(u_V, +) = (q_R, +, R)$ place the head on first **0,** return to tape 1
$f(u_V, -) = (q_L, -, R)$ place the head on first **0,** return to tape 1

In this way we generate a 10^{i+1} long string of **0**s from a string of 10^i **0**s on tape 2 each time we move one cell to the right on tape 1. It remains to build a sum in unary representation on tape 3. To do so we traverse the string of **0**s on tape 2 and transfer every **0** onto tape 3 each time we decrease a digit of the number on tape 1 by 1.

$f(u_R, 0) = (t_R, 0, R)$ if we read a **0** on tape 2 we write it onto tape 3
$f(u_L, 0) = (t_L, 0, R)$ if we read a **0** on tape 2 we write it onto tape 3
$f(u_R, B) = (u_V, B, L)$ if we find a blank **B** we move back to the first **0**
$f(u_L, B) = (u_V, B, L)$ if we find a blank **B** we move back to the first **0**

Tape 3 (creation of the sum in unary representation, tape is unlimited on both sides)

Addition
$f(t_R, 0) = (u_R, B, R)$ replace **0** by **B** then move to the right
$f(t_R, B) = (u_R, 0, R)$ replace **B** by **0** then move to the right

Subtraction
$f(t_L, 0) = (t_K, 0, L)$ move first to the left
$f(t_L, B) = (t_K, B, L)$ move first to the left
$f(t_K, 0) = (u_L, B, n)$ then replace **0** by **B**
$f(t_K, B) = (u_L, 0, n)$ then replace **B** by **0**

Please note: we start with an empty tape 3 and an arbitrary placed r/w head so **..BBB..** represents a 0 value (underscore indicates here the position of the r/w head). In unary representation the number i of **0**s gives us the integer i ∈ Z. The sign is represented as the position of the head, which after each loop step can be at the first blank **B** to the right of the **0**'s-block (positive integer) or the most left **0** in the **0**'s-block (negative integer), so e. g. **..B0000B..** represents -4 and **..B0000B..** represents +4. However, our focus lies here not on the evaluation of a result value but on checking if the tape 3 = 0, which means that we attained the final state q_F, and accepted the examined word. Note further, that the state determines on which tape the next operation takes place, so the states q… indicate, that we operate on tape 1, u… on tape 2 and t… on tape 3.

With the above definition of the transition function f we show by induction over the number of loops in W, that TM M_3 generates an unary result char(y_k) if W computes the sum y_k after k loops. M_3 terminates if the WHILE program W reaches $y_k = 0$ during the computation according to the loop condition **WHILE** $y_k \neq$ **0 DO P END**. Naturally, we can't say if W will terminate at all, we solely make sure that TM M_3 simulates W properly.

The induction hypothesis is trivially true for a 0-loop, where y = num(ω) = 0, so the tape 3 of M_3 remains empty and M_3 reaches its final state as after $f(q_I, +/-) = (q_{R/L}, +/-, R)$ the r/w head moves to the right and we transfer **0**s to tape 3 only for $f(q_{R/L}, X) = (u_{R/L}, X-1, n)$ for $X \in \{1, \ldots, 9\}$. For $f(q_{R/L}, 0) = (u_I, 0, R)$ we move to the right hit on a blank cell and check then tape 3, which still remains empty. We assume then that the hypothesis holds for k-1 loops. We show that

$$y \overset{k-1}{\Longrightarrow}_{(W)} y_{k-1} \Rightarrow_{(W)} y_k \text{ implies}$$

$$\omega = \text{char}(y) \overset{k-1}{\Longrightarrow}_{(M)} \omega_{k-1} = \text{char}(y_{k-1}) \Rightarrow_{(M)} \omega_k = \text{char}(y_k)$$

The initial numerical value y is processed by the WHILE program W which evaluates the value y_{k-1} after k – 1 WHILE loops. The TM $M_3 := (Q, \Sigma, \Gamma, f, q_0, B, F)$, which processes the word $\omega = \text{char}(y) = a_1..a_nBBB..$ ($a_1,..,a_n \in \Sigma$) evaluates the word $\omega_{k-1} = \text{char}(y_{k-1})$ after simulation of k – 1 additions/subtractions. Let the word on tape 3 be then $\omega_{k-1} = b_1 10^1 + \ldots + b_{i-1}10^{i-1} + b_i 10^i + \ldots + bj10^j$ in unary representation.
The next number to add/subtract is c so $y_k = y_{k-1} \pm c$ and $\chi = \text{char}(c)$. But then $\omega_k = \omega_{k-1} \pm_s \chi = \text{char}(y_{k-1}) \pm_s \text{char}(c) = \text{char}(y_{k-1} \pm c) = \text{char}(y_k)$, where $\pm_s$ is the simulation of the addition or subtraction in M_3 and the conclusion follows from the additivity of char(x) regarding $\pm_s$. In other words, the result on tape 3 after one loop when we process char(y_k) on an empty tape 3 is the same as if we processed char(c) on a tape 3 containing the intermediate sum of char(y_{k-1}) if $y_k = y_{k-1} \pm c$. In particular, if $y_k = 0$ so tape 3 of M_3 becomes empty and M_3 reaches the final state q_F.
Eventually, we have to show the correctness of the unary representation of integers on tape 3. We start with an empty tape and add **0**'s to the left (subtraction) or to the right (addition) of the r/w head. We generate **0**'s (left of the r/w head when we subtract from a negative integer or right of the r/w head when we add to a positive integer) or blanks (left of the r/w head when we subtract from a positive integer or right of the r/w head when we add to a negative integer) on tape 3 as packages of 10^i **0**'s corresponding to the decreasing integer to basis 10 at digit position i. After addition/subtraction of one integer c, tape 1 contains the integer 0 and tape 3 contains the unary representation of $y_k = y_{k-1} \pm c$, where y_{k-1} was the intermediate sum after k-1 additions/subtractions. □

The definition of a TM and the traceability of its processing are not practicable for complex evaluations. In fact, it is a theoretical model of computation and we have proven, that more convenient models like formal grammars or WHILE programs are equivalent to TMs, in the sense that every function, which is Turing-computable, can be implemented by means of formal grammars or WHILE programs and that the opposite conclusion holds as well. We will go a step further and show the following equivalence.

4 Equivalence between numerical WHILE computation and constraint satisfaction WHILE computation

We introduce now a computational model which differs from the numerical WHILE computation only in the aspect, that we rather use Boolean operations instead of numerical during the WHILE computation. We define constraint satisfaction (cs) programs as a 4-tuple (**Σ,** Ψ, F, WHILE cs-program) where

Σ is a finite set of valid input symbols α_i, i ∈ {0, …, n}
Ψ is a set of constraints ψ_k, k ∈ N with ψ_k: **Σ** → B or **Σ** × **Σ** → B, $\alpha_i \mapsto \{0,1\}$ or $\alpha_i \times \alpha_j \mapsto \{0,1\}$ for $\alpha_i, \alpha_j \in$ **Σ**
F is a Boolean function F: $B^d \rightarrow B$, $\psi_1(\alpha_i) \wedge \ldots \wedge \psi_c(\alpha_j) \wedge \psi_{c+1}(\alpha_i, \alpha_j) \wedge \ldots \wedge \psi_d(\alpha_i, \alpha_j) \mapsto \{0,1\}$ for $\alpha_i, \alpha_j \in \Sigma$ and $\psi_1, \ldots, \psi_d \in \Psi$

For $\psi_k(\alpha_i) = 0$ or $\psi_k(\alpha_i, \alpha_j) = 0$ (false) we say that constraint ψ_k is not satisfied with α_i resp. α_i, α_j else it's satisfied. We look for one or many sets of symbols S_i, which satisfy all $\psi_1, \ldots, \psi_d \in \Psi$. **Σ** can be a set of alphanumerical symbols or integers, so $\psi_k(\alpha_i)$ can be defined e. g. as $\psi_k(\alpha_i) := \alpha_i > C$ with a constant C (α_i, C ∈ N_0) or $\psi_k(\alpha_i, \alpha_j) := \alpha_i > \alpha_j$ ($\alpha_i, \alpha_j \in N_0$). Furthermore, we can define a set of constraints, which is variably large depending on the number of input symbols in the examined set e. g. $\psi_k(\alpha_i, \alpha_{i+1}) := \alpha_i \neq \alpha_{i+1}$ so for a set $\{\alpha_1, \alpha_2\}$ $\psi_1(\alpha_1, \alpha_2) := \alpha_1 \neq \alpha_2$ but for a set $\{\alpha_1, \alpha_2, \alpha_3\}$ $\psi_1(\alpha_1, \alpha_2) := \alpha_1 \neq \alpha_2$ and $\psi_2(\alpha_2, \alpha_3) := \alpha_2 \neq \alpha_3$.

With S_i, S_j being defined as sets of input symbols and ⊕ as a change operator $S_i := S_j \oplus \Delta s$, which changes the input set S_j in an appropriate way according to the change set Δs we define now inductively WHILE cs-programs as:

$S_i := S_j \oplus \Delta s$ is a WHILE cs-program for every i, j ∈ N_0 (change operator)
If P_1, P_2 are WHILE cs-programs, then **P_1; P_2** is a WHILE cs-program (composition)
If P is a WHILE cs-program, then **WHILE** $\psi_1(\alpha_i) \wedge \ldots \wedge \psi_c(\alpha_j) \wedge \psi_{c+1}(\alpha_i, \alpha_j) \wedge \ldots \wedge \psi_d(\alpha_i, \alpha_j) \neq 1$ **DO P END** for every i, j, c, d ∈ N_0 (WHILE loop)

As already stated, sometimes we are interested in only one, sometimes in many or all cs-solutions for a set of constraints. We can relax the set of constraints and hope to obtain more solution sets for our problem or restrain the set of constraints by adding new constraints to try to reduce the number of solution sets. We show now the equivalence between the two WHILE program models and give an example for a hexadecimal representation.

Proof. The change operator ⊕ can be interpreted as subtraction/addition of integers to basis | Σ | if we built a word $\omega = a_n \ldots a_0$ where $S_j = \{a_0, \ldots, a_n\} \in \Sigma$ and $x_j = \text{int}(\omega)$, $c = \text{int}(\Delta s)$ so $x_i := \text{int}(\omega) \pm \text{int}(\Delta s) = x_j \pm c$.
On the other hand, every subtraction/addition of integers represented to basis X = | Σ | can be expressed as a change operation when we define for all elements in Σ a change function P or M for addition/subtraction as $P(a_i, b_i) = (a_j, b_{i+1})$ or $M(a_i, b_i) = (a_j, b_{i+1})$ with $a_i, a_j, b_i, b_{i+1} \in \Sigma$, the carry word $\beta = b_{n+1}b_n \ldots b_0$ and the following procedural rule:
BUILD the word $\beta = b_{n+1}b_n \ldots b_0$, pad on the left side of ω, β with **0**'s so length(ω) = length(β)
WHILE $\beta \neq 0$ **DO**
for $a_i, b_i \in S_i$ evaluate $a_0, b_1 = P(a_0, b_0), \ldots, a_n, b_{n+1} = P(a_n, b_n)$ (for addition)
or $a_0, b_1 = M(a_0, b_0), \ldots, a_n, b_{n+1} = M(a_n, b_n)$ (for subtraction) and **BUILD** the new word $\beta = b_{n+1}b_n \ldots b_0$ with $b_0 = 0$
END
BUILD the result word $\omega = b_{n+1}a_n \ldots a_0$ □

Example. For hexadecimal **signs** {0, 1, …, 9, A, B, C, D, E, F} we would define P (addition) as:
P(0, 0) = (0, 0); P(1, 0) = (1, 0); …, P(E, 0) = (E, 0); P(F, 0) = (F, 0)
P(0, 1) = (1, 0); P(1, 1) = (2, 0); …, P(E, 1) = (F, 0); P(F, 1) = (0, 1)
…
P(0, E) = (E, 0); P(1, E) = (F, 0); …, P(E, E) = (C, 1); P(F, E) = (D, 1)
P(0, F) = (F, 0); P(1, F) = (0, 1); …, P(E, F) = (D, 1); P(F, F) = (E, 1)

It resembles the realisation of an adder by NAND-gates with a carry register, which uses nothing else as Boolean operations to implement addition/subtraction. Furthermore, we can easily see the commutativity of P. According to the above definition the "addition" of two hexadecimal words (e. g. 2F7BCC5 and 5EF189D) would yield in 2 loops:

α = 2F7BCC5 → 7D6C452 → 8E6D562
β = 5EF189D → 1101110 → 0000000
so 2F7BCC5 + 5EF189D = 8E6D562

A. Turing presented his model of computation (TM) in 1930's. With it he was able to prove properties of computation in general, like the decision and completeness problem. Though with a TM we can implement any computer algorithm, it's very laborious to formulate a TM-code even for relatively simple problems. Therefore, other, equivalent models had been developed, which are more suitable for practical implementations. The cs-WHILE computation isn't new but, to my knowledge, wasn't used extensive, if at all, for theoretical considerations. With this model we can deal with theoretical problems as well as implement comprehensible solutions for very concrete tasks. In the two next chapters we implement cs-solutions for two different problems: the map colouring problem and the pattern matching problem, which gives us a database for analysis of a putative plaintext in a substitution cipher.

5 Coloring of a map

The map colouring problem is the problem to colourise neighbour countries or regions (neighbour regions having a common border) with different colours on a map with many regions. There is one catch, we should use as few colours as possible. We don't need to evaluate all possible solutions or all permutations of one solution, one single solution would satisfy our inquisitiveness as for this task.

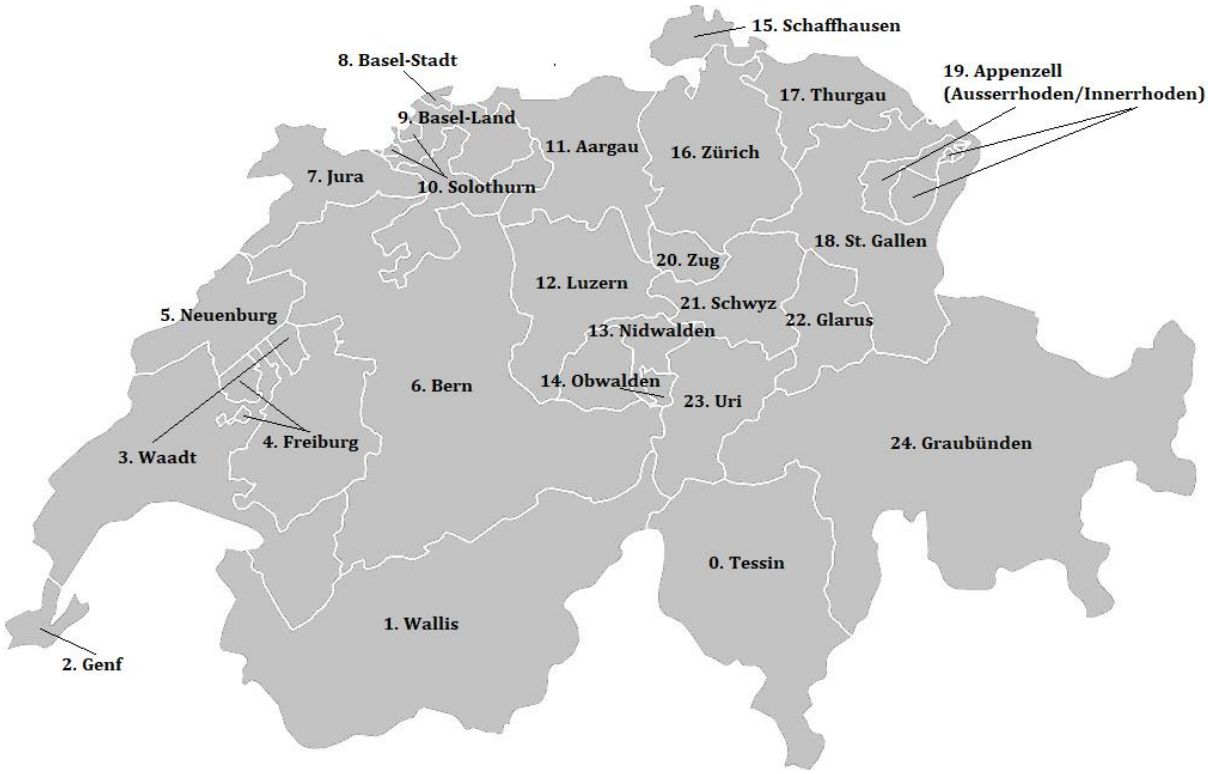
We choose thus a problem over regular sets, which are only subsets of recursively enumerable sets recognisable by a TM or a cs-WHILE program. This gives us however the certainty, that our algorithm will halt, with a solution for the problem or without.

We take a concrete map - that of the cantons of the Swiss Confederation - and try to find a solution with a minimal number of colours. We don't deal here with the four-colours problem as such - a theorem says, that no more than four colours are required to colour the regions of any map, if the regions are contiguous – but we expect to make do with 4 colours, despite the fact, that we have a few exclaves of some cantons, so that the regions aren't always contiguous. We define a set of constraints Ψ of only one type ψ(a, b) := diff(a, b), which means a and b must be different and use the symbol # for the "different"-relation.

With the cantons {0 Tessin, 1 Wallis, 2 Genf, 3 Waadt, 4 Freiburg, 5 Neuenburg, 6 Bern, 7 Jura, 8 Basel-Stadt, 9 Basel-Land, 10 Solothurn, 11 Aargau, 12 Luzern, 13 Nidwalden, 14 Obwalden, 15 Schaffhausen, 16 Zürich, 17 Thurgau, 18 St. Gallen, 19 Appenzell (Ausserrhoden/Innerrhoden), 20 Zug, 21 Schwyz, 22 Glarus, 23 Uri, 24 Graubünden}
e. g. 0#1 means then, that Tessin and Wallis need to be coloured with different colours.

We have to satisfy the following set of 54 constraints (see the map below):

Ψ := {0#1 0#23 0#24 1#3 1#6 1#23 2#3 3#4 3#5 3#6 4#5 4#6 5#6 5#7 6#7 6#10 6#11 6#12 6#13 6#14 6#23 7#9 7#10 8#9 9#10 9#11 10#11 11#12 11#16 11#20 12#20 12#21 12#13 12#14 13#14 13#23 14#23 15#16 15#17 16#17 16#18 16#20 16#21 17#18 18#19 18#21 18#22 18#24 20#21 21#22 21#23 22#23 22#24 23#24}



We use bash scripts for the implementation of these simple cs-programs. Bash scripts can be run on Unix/Linux. On Windows e. g. Cygwin (a collection of GNU and Open Source tools which provide functionality similar to a Linux distribution on Windows) can be installed[3] to run bash scripts. The code is as following (the arrays *map* and *arr* define the constraints Ψ and the set of result values, the last value of arr = 0 is a delimiter and not a result value):

```
#!/bin/bash
typeset -a map=(" 0#1 0#23 0#24 1#3 1#6 1#23 2#3 3#4 3#5 3#6 4#5 4#6 5#6 5#7 6#7 6#10 6#11 6#12 6#13
6#14 6#23 7#9 7#10 8#9 9#10 9#11 10#11 11#12 11#16 11#20 12#20 12#21 12#13 12#14 13#14 13#23 14#23
15#16 15#17 16#17 16#18 16#20 16#21 17#18 18#19 18#21 18#22 18#24 20#21 21#22 21#23 22#23 22#24
23#24 ")
typeset -a arr=(1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 0)
typeset -i n=0 m=0 k=0 l=0
```

[3] we do not guarantee and do not assume any liability, that the script is runnable on your operating system, we could run it on Unix and port it successfully to Cygwin on Windows

```
# main while loop

while (( arr[++m] != arr[-1] ))
do
	while ((n < m ))
	do
		diffstr=" ${n}#${m} "
		if [[ "${map[*]}" =~ "${diffstr}" ]]; then
		#echo {$diffstr}
		if (( arr[n] != arr[m] )); then (( n++ ))
		else (( arr[m]++ )); (( n = 0 ))
		fi
		else (( n++ ))
		fi
	done
	((n = 0 ))
done
echo ${arr[*]}
(( arr[m] == 0 )) && echo "assign colour [nb] to variables arr[0], ..., arr[24]"

# try to resolve following constraints to obtain a 4-colour result

while (( arr[++k] != arr[-1] ))
do
	if (( arr[k] > 4 )); then
	while (( l < k ))
	do
		if [[ "${map[*]}" =~ " ${l}#${k} " ]]; then
		echo "resolve constraint " ${l}#${k} " to potentially reduce the number of used colours"
		fi
		(( l++ ))
	done
	fi
	(( l = 0 ))
done
```

When we put this code into the executable file swiss_map.ksh we obtain the following result:

```
script_home@mycomp ~
$ ./swiss_map.ksh
1 2 1 3 1 2 4 1 1 2 3 1 2 1 3 1 2 3 1 2 3 4 2 5 3 0
assign colour [nb] to variables arr[0], ..., arr[24]
resolve constraint  0#23  to potentially reduce the number of used colours
resolve constraint  1#23  to potentially reduce the number of used colours
resolve constraint  6#23  to potentially reduce the number of used colours
resolve constraint  13#23  to potentially reduce the number of used colours
resolve constraint  14#23  to potentially reduce the number of used colours
resolve constraint  21#23  to potentially reduce the number of used colours
resolve constraint  22#23  to potentially reduce the number of used colours
```

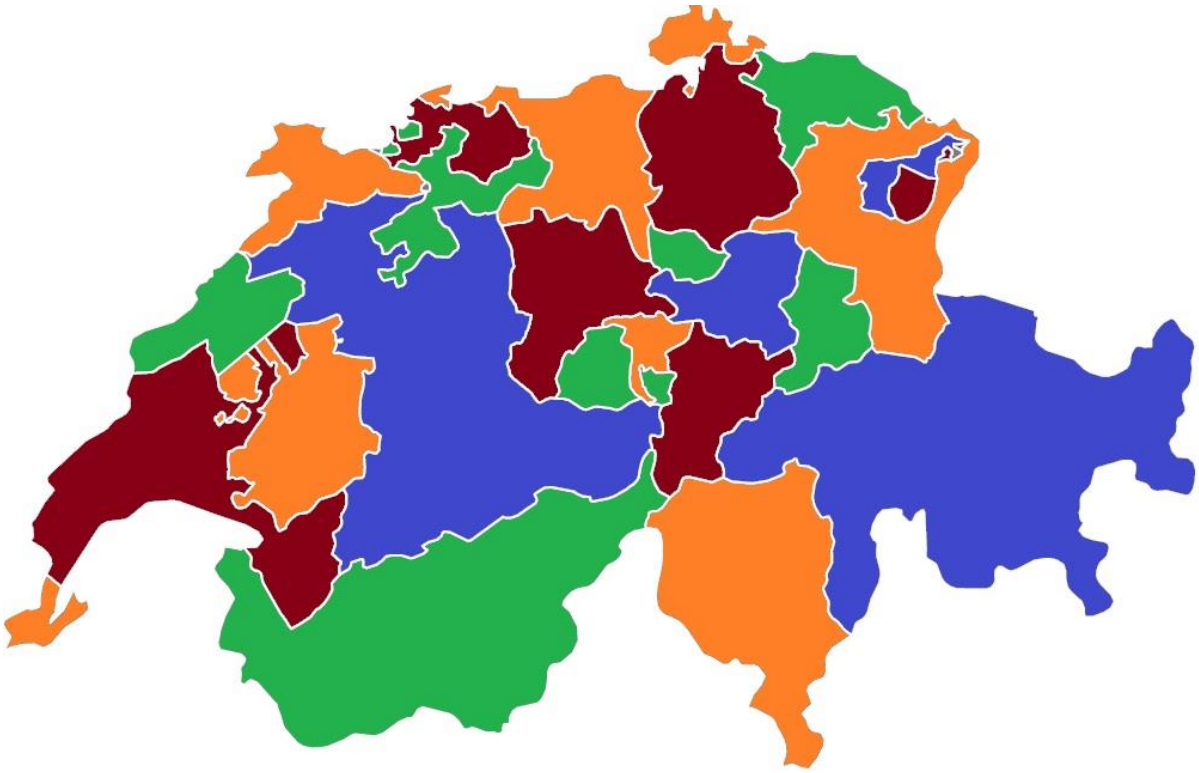
We got a result for colouring of the cantons with 5 colours and a hint which constraints possibly contributed to the extension of the result set to 5 colours. We can try to introduce additional constraints or reduce the possible result value range for specified variables to obtain a 4-colour result. We do so for the variables arr[1] and arr[22] specified in the resolving hints above and reduce the value range for them to the colour value 3 and higher (excluding 1 and 2). We can re-run the script with the following definition of *arr*:

```
typeset -a arr=(1 3 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 3 1 1 0)
```

Now we obtain the following 4-colour result (note, that we didn't differentiate between Appenzell Innerrhoden and Ausserrhoden and are free to choose different colours for them as they are completely surrounded by the canton St. Gallen):

```
script_home@mycomp ~
$ ./swiss_map.ksh
1 3 1 2 1 3 4 1 1 2 3 1 2 1 3 1 2 3 1 2 3 4 3 2 4 0
assign colour [nb] to variables arr[0], ..., arr[24]
```

This 4-colour solution for our colouring problem would thus result in the following possible colouring:



The resolving strategies (relaxation of constraints in case of too strong conditions or restraint of constraints in case of too many result sets) can be automatised. The program can learn from intermediate results how to manage the result sets and how to improve the computation in terms of reduction of time-consuming by changing the order of examined constraints.
This meta-level inspection of the own solving process is a subject of AI. I will come back to this topic in the following papers.