FACTA UNIVERSITATIS (NIŠ)
SER. MATH. INFORM. 24 (2009), 53–72

TURING MACHINE AND ITS SYMBOLIC SIMULATION *

Saša V. Vukašinović, Predrag S. Stanimirović, Marko D. Petković, and Miroslav D. Ćirić

Abstract. We describe an implementation of the Turing machine by means of symbolic processing, functional programming and 2D graphical primitives available in the programming language MATHEMATICA. We determine the number of tapes and the number of rules corresponding to a given non-deterministic Turing machine, and prove that these numbers are bounded, regardless the halting problem. These results are used in the simulation of the non-deterministic Turing machine.

1. Introduction and preliminaries

A Turing machine M is a finite device which performs operations on a paper tape. This tape is infinite in both directions, and divided into single cells (squares). At any time, each square of the tape contains a single symbol from a fixed finite list of input symbols Σ or the blank character #. Cells that have not been written to before are assumed to be filled with the blank symbol. Formally, a Turing machine is defined as a 6-tuple

$$(Q, q_0, \#, \Sigma, \Gamma, \delta) \tag{1.1}$$

where:

Q is a non-empty and finite set of states;

q_0 is the initial state;

is the blank symbol (the only symbol allowed to occur on the tape infinitely often at any step during the computation);

Γ is the alphabet of the tape, defined by $\Gamma = \Sigma \cup \{\#\}$;

$\delta : Q \times \Gamma \to Q \times \Sigma \times \{L, R, P\}$ is a partial transition function.

We stipulate that the machine M starts computation in the initial state q_0 and the reading head positioned on the first left cell which is not the blank character

Received March 03, 2009.
2000 *Mathematics Subject Classification.* Primary 68Q05; Secondary 68W30
*Supported by Ministry of Science and Technological Development, Republic of Serbia, Grant no. 144011.

#. The number of different states is always finite. The action that M takes at any instant depends on the current state of M and the symbol currently being scanned. This dependence is defined in M's transition function δ, which is defined by the set T of rules of the form $\{q_i, s_j\} \to \{q_k, s_l, \alpha\}$, where $q_i, q_k \in Q$, $s_j \in \Gamma$, $s_l \in \Sigma$ and $\alpha \in \{L, R, P\}$. The rule $\{q_i, s_j\} \to \{q_k, s_l, \alpha\}$ specifies the action when M is in the state q_i and the head scans the symbol s_j. In this case, the machine do the following in sequence: (i) takes the new state q_k, (ii) in the scanned cell writes the symbol s_l or erases the symbol s_j and (iii) moves the head (if $\alpha = R$, moves the reading head one cell to the right, if $\alpha = L$, moves the tapehead one cell to the left, and if $\alpha = P$ keeps the reading head in the same place). We use the following convention: if s_l is the space character (SPACE), the machine erases s_j from the tape. The Turing Machine will halt if its current state and current symbol do not match the first two components of any rule.

We have described the *deterministic Turing machine*. Now, we will describe the *non-deterministic Turing machine*. A Turing machine is non-deterministic if its action table has at least one entry for each combination of symbol and state. The formal definition of the non-deterministic Turing machine corresponds to definition of the deterministic Turing machine, except in the last component in (1.1). Namely, in the non-deterministic case δ is a relation from $Q \times \Gamma$ to $Q \times \Sigma \times \{L, R, P\}$ instead of a partial transition function. It means that for each element $\{q_i, s_j\} \in Q \times \Gamma$ there exist a number of different triples $\{q_k, s_l, \alpha\} \in Q \times \Sigma \times \{L, R, P\}$ during the execution of a simple action of the Turing machine M. Formally δ is defined by:

$$\delta(\{q_i, s_j\}) \subseteq \bigcup_{q_k \in Q,\ s_l \in \Sigma,\ \alpha \in \{L,R,P\}} \{q_k, s_l, \alpha\}. \tag{1.2}$$

This is one of many variations on the Turing machine theme (see for example [3, 7, 8]). Hopcroft and Ullman ([7]) formally define a (one-tape) Turing machine as a 7-typle

$$(Q, q_0, \#, \Sigma, \Gamma, \delta, F) \tag{1.3}$$

where $F \subseteq Q$ is the set of final or accepting states. Also, there exist definitions where instructions are defined as 4-tuples.

It is known that all of these definitions are equivalent. About the Turing machine the reader is also referred to [2, 5, 10, 11, 12].

We observed several simulations ([6, 16, 17, 18]) of the Turing machine, written in procedural programming languages, and we will give some notes about that. Main criteria used in this analysis are: existence of non-deterministic part, simple usage, graphical illustrations, power in computations and code simplicity. The simulation in [18] has two parts. In the first part it simulates deterministic Turing machine. This simulation uses object oriented programming, but it is not user friendly and does not use symbolic computation. Second part contains simulation of the non-deterministic Turing machine. This simulation does only recognition of input word but not any computation. Moreover, graphical presentation is not used.

The simulation [16] is very good visual simulation with powerful computation, but there is not non-deterministic part. This simulation is, essentially, an improvement of the simulation [17]. The simulation [6] is the specific usage of the Turing machine and it is interesting as a similar work in the same MATHEMATICA package.

Consequently, Turing machine interpreters are so far written in procedural programming languages. Turing machine as a C++/Java function possesses the following framework:

```
int anArbitraryTuringMachine(char tape[])
    { unsigned state = initialState; unsigned current = 0;
       do { //series of if statements like this ...
               if(state == p && tape[current] == x)
               { state = q; tape[current] =y;}
               //or
               if(state == p && tape[current] == x)
               { state = q; ++current; }
              // or - // etc ...
   while (state != haltState)
}
```

In [4] we found a simulation of deterministic 3-state, 2-color Turing machine in MATHEMATICA. Turing machines discussed in "A New Kind of Science" [15] are deterministic. There is an another class of so called "non-deterministic" Turing machines. Modern computers are deterministic, so any executing over any non-deterministic type of machine (*FA* or Turing) must be simulated.

In this paper we describe an implementation of the deterministic and non-deterministic Turing machine in the package MATHEMATICA. We mainly use possibility of symbolic processing in MATHEMATICA. Besides the symbolic possibilities, we use its power in functional programming as well as in the representation of two-dimensional graphical images.

Motivation for this work is to develop an user friendly interpreter for educational purpose and experiments in Theoretical Computer Science. Also, our motivation is to exhibit symbolic and graphical possibilities of the package MATHEMATICA, and exceed the framework described above. Programs written in procedural computer languages such as C help with computations, but are of limited value in developing understanding of included algorithms, because very little information about the intermediate steps is presented. The user interface for such programs is primitive at best and typically requires definitions of user functions and data files corresponding to a specific problem. Many researches have a need for the capability to develop a "rapid-prototype" code to test the behavior of an algorithm before invest in a substantial effort in developing a code for the algorithm in a procedural language such as C.

About the package MATHEMATICA see for example [1, 9, 13, 14].

The paper is organized as follows. In the second section we describe an implementation of the deterministic Turing machine. In the third section we give several theoretical results. We determine the number $n \in N$ of deterministic Turing machines (i.e. tapes) corresponding to a given non-deterministic Turing machine, and prove that the number of tapes is bounded, regardless the halting problem. Moreover, we prove that all programs performing actions on these tapes are of fixed length. In the Section 4 we describe a simulation of the non-deterministic Turing machine. Work of the non-deterministic Turing Machine M can be considered as a parallel work of n deterministic Turing machines with identical number of rules.

2. Simulation of deterministic Turing machine

We restate main details from the functional programming available in MATHEMATICA, which are used in the simulation.

`Map[f, expr]` or `f /@ expr` applies function f to each element on the first level in *expr*.

`Scan[f, expr]` evaluates f applied to each element of *expr* in turn. Unlike *Map*, *Scan* does not build up a new expression to return.

`Function[body]` or `body&` is a pure function. The formal parameters are # (or #1), #2, etc.

The contents of the tape is represented by the global array $TAPE[1], TAPE[2], \ldots$ in which you fill in only those elements that you need at a particular time, i.e. elements corresponding to non-blank cells in the tape. The value $TAPE[i]$ represents the contents of i-th cell on the tape, with respect to last non-blank cell on the left. Array *TAPE* can be defined by a string s, using the function $InitTape[s]$. This function defines global array *TAPE*, such that $TAPE[i] = s[i]$. Global variables *MINPOS* and *MAXPOS* denote the minimal and maximal index, respectively, between the indices in the array *TAPE*. All remainder cells contain the blank character # (also called *empty* cells). Also, global variable *POS* denotes the index of the actually scanned cell. In the function $InitTape[s]$ we use the mapping function *Map* and a pure function during the construction of the array *TAPE*:

```
InitTape[s_String] := (
    MAXPOS = 0;
    Clear[TAPE];
    TAPE[_] := "#";    (* Fills the tape by blank characters *)
    Map[(TAPE[++MAXPOS] = #) &, Characters[s]];
    MINPOS = POS = 1;         )
```

Immediately after the initialization of the tape by means of the function $InitTape[s]$, the minimal index *MINPOS* as well as the actual position *POS* are both equal to 1, and the value for global variable *MAXPOS* is equal to length of string s. These values can be modified later.

Example 2.1. *The tape which is defined by the string "1101" can be filled using the following expression*

```
In[1]:= InitTape["1101"]
```

The contents of the tape can be shown in this way:

```
In[2]:= ?TAPE
        Global `TAPE
        TAPE[1] = "1"
        TAPE[2] = "1"
        TAPE[3] = "0"
        TAPE[4] = "1"
        TAPE[_] := "#"
```

The alphabet Σ (Γ), the set of states Q and actual state as well as the transition function δ are defined using the set of rules

$$T = \left\{ \{q_{i_m}, s_{j_m}\} \to \{q_{k_m}, s_{l_m}, \alpha\},\ m = 1, \ldots, k \right\}. \tag{2.1}$$

The sets Q, Σ and Γ are implicitly defined by

$$Q = \bigcup_{m=1}^{k} \{q_{i_m}\} \bigcup \bigcup_{m=1}^{k} \{q_{k_m}\},\ \ \Sigma = \bigcup_{m=1}^{k} \{s_{l_m}\},\ \ \Gamma = \bigcup_{m=1}^{k} \{s_{j_m}\} \bigcup \Sigma \bigcup \{\#\}.$$

The transition function δ is defined by the rules contained in T:

$$\delta(\{q_1, s_1\}) = \begin{cases} \{q_2, s_2, \alpha\}, & \{q_1, s_1\} \to \{q_2, s_2, \alpha\} \in T \\ \{\}, & \{q_1, s_1\} \to \{q_2, s_2, \alpha\} \notin T. \end{cases} \tag{2.2}$$

The function δ is defined in the function *InitDelta*[*T*], where it is denoted by the user-defined function *DELTA*. Also, during the evaluation of the function *InitDelta*[*T*] we define the initial state q_0, denoted by the global variable *STATE*. In the function *InitDelta*[*T*] we use the function *Scan* which applies a pure function to each element of the list *T*.

```
InitDelta[T_] := (
    Clear[DELTA];
    DELTA[_, _] := {};  (* The second case in (2.2)  *)
    Scan[(DELTA[#[[1,1]], #[[1,2]]] = #[[2]]) &, T];
                        (* The first case in (2.2)  *)
    STATE = T[[1,1,1]]; )
```

Example 2.2. *Let the set T is defined by the list of rules*

```
In[3]:=T={{A,"0"}->{A,"1",R},{A,"1"}->{A,"1",R},{A,"#"}->{B,"  ",L}}
```

After the application of the function InitDelta[T] we get the following initial state q_0, denoted by STATE, and the following definition of the function DELTA:

```
In[4]:=InitDelta[T];
       STATE
Out[4]=A

In[5]:= ?DELTA
     Global‘DELTA
     DELTA[A, #] = {B,  , L}
     DELTA[A, 0] = {A, 1, R}
     DELTA[A, 1] = {A, 1, R}
     DELTA[_, _] := {}
```

In the auxiliary function *TapeToList*[*t*] we construct a list whose elements are characters contained in the array *t* of indexed variables, starting from the position *MINPOS* and ending at the position *MAXPOS*. It is clear that the value for the formal parameter *t* is the global array *TAPE*.

```
TapeToList[t_] :=
  Module[{l = {}, i},
    For[i=MINPOS, i<=MAXPOS, i++, AppendTo[l, t[i]]];
    Return[l];
  ]
```

All relevant elements which determine the situation of the machine *M* are incorporated in the ordered quintuple

$$\{STATE, MINPOS, POS, MAXPOS, TapeToList[TAPE]\} \tag{2.3}$$

In the sequel we construct a global list *History* which contains elements of the form (2.3). In this way, the list *History* stores all steps of the evaluation. It can be formed using the functions *Move* and *Compute*. The formal parameter of the function *Move* is the list $\{s, x, d\}$ which represents the right hand side of a selected rule $\{a, b\} \to \{s, x, d\}$ from *T*.

By means of the function *Move*[$\{s, x, d\}$] we define the list of the form (2.3) and append it to the end of the list *History*.

```
Move[{s_, x_, d_}] := (
  If[x === " ",               (* condition for erasing TAPE[POS] *)
    MAXPOS = MAXPOS - 1; TAPE1 = TAPE;
```

```
    For[ji = POS, ji <= MAXPOS, ji++,
        TAPE[ji] = TAPE1[ji + 1]];
        TAPE[MAXPOS + 1] =.   (* Erase TAPE[POS]  *)
    ];
  POS = POS + If[d === R, 1, If[d === L, -1, 0]];
  MINPOS = Min[POS, MINPOS];  MAXPOS = Max[POS, MAXPOS];
  STATE = s;
  AppendTo[History,{STATE,MINPOS,POS,MAXPOS,TapeToList[TAPE]}];
  True            )

Move[{}] = False;
```

The machine stops when the value of the function *Move* is equal to *False*. This means that there is no rule $\{q_i, s_j\} \to \{q_k, s_l, \alpha\}$ from T in which s_l, q_k, α agrees with $s\_, x\_, d\_$, respectively. The stopping criterion of our machine is different with respect the stopping criterion in simulation [18], where the set of final states is used to halt the evaluation.

Formal parameters of the next function *Compute* are:

T - the set of rules (2.1),

s - a string which determines the contents of the tape.

Values for the formal parameters s, x, d in each call of the function *Move* are generated applying the function *DELTA*:

```
Compute[T_, s_] := (
   InitTape[s];
   InitDelta[T];
   History = {{STATE,MINPOS,POS,MAXPOS,TapeToList[TAPE]}};
   While[Move[DELTA[STATE, TAPE[POS]]]]
                  );
```

Example 2.3. *Consider the set of rules T defined as in Example 2.2. Then the expression Compute*[*T*, "110101010"] *produces the following list, which represents value of the global variable History:*

```
{{A,1,1,9,{1,1,0,1,0,1,0,1,0}},
 {A,1,2,9,{1,1,0,1,0,1,0,1,0}},
 {A,1,3,9,{1,1,0,1,0,1,0,1,0}},
 {A,1,4,9,{1,1,1,1,0,1,0,1,0}},
 {A,1,5,9,{1,1,1,1,0,1,0,1,0}},
 {A,1,6,9,{1,1,1,1,1,1,0,1,0}},
 {A,1,7,9,{1,1,1,1,1,1,0,1,0}},
 {A,1,8,9,{1,1,1,1,1,1,1,1,0}},
```

```
 {A,1,9,9,{1,1,1,1,1,1,1,1,1,0}},
 {A,1,10,10,{1,1,1,1,1,1,1,1,1,#}},
 {B,1,9,9,{1,1,1,1,1,1,1,1,1,1}}
}
```

Example 2.4. *In this example we show erasing the content of scanned cell when value of parameter x in function* `Move` *is the space character. Consider the set of rules*

```
T={{A,"0"}->{A,"  ",R},{A,"1"}->{A,"1",R},{A,"#"}->{B,"  ",L}}.
```

The evaluation of the expression `Compute[T,"101"]` *produces the following list History:*

```
{{A,1,1,3,{1,0,1}},
 {A,1,2,3,{1,0,1}},
 {A,1,3,3,{1,1,#}},
 {B,1,2,2,{1,1}}
}
```

We are going to write the function *DumpHistory* which produces a 2D graphical representation of the list *History*. For this purpose we use a number of graphics primitives from MATHEMATICA [13, 14].

`Line[{{`x_1`,`y_1`},...}]`, is a graphics primitive which represents a line joining a sequence of points.

`Rectangle[{`x_{min}`,`y_{min}`}, {`x_{max}`,`y_{max}`}]`, is a two-dimensional graphics primitive that represents a filled rectangle, oriented parallel to the axes.

`Text[expr,{x,y}]` is a graphics primitive that represents text corresponding to the printed form of *expr*, centered at the point $\{x, y\}$.

`Graphics[primitives, options]` represents a two-dimensional graphical image;

`Show[graphics, options]` displays two- and three-dimensional graphics, using the options specified.

By means of the function `cellc[i,j,k]` we construct the list of two graphics primitives. The first graphics primitive is a line joining four vertices of a square. It is determined by the standard function *Line*:

```
Line[{{i,j}, {i+1,j}, {i+1,j+1}, {i,j+1}, {i,j}}]
```

The second graphics primitive is defined by the function *Text*[*k*,{*i*+0.5,*j*+0.5}], and represents text corresponding to the printed form of *k*, centered at the point specified by $\{i + 0.5, j + 0.5\}$, overlaying on the center of the first graphics object.

```
cellc[i_, j_, k_] := {
    Line[{{i,j}, {i+1,j}, {i+1,j+1}, {i,j+1}, {i,j}}],
    Text[k, {i+0.5, j+0.5}]
                    }
```

Function `Headc[i,j,s]` generates a list whose both elements are graphics primitives. The first element, denoted by *s*, represents a graphics directive which specifies that graphical objects which follow will be displayed, if possible, in the color given. The second element is a filled rectangle, oriented parallel to the axes, and determined by the parameters *i* and *j*.

```
Headc[i_, j_, s_RGBColor] :=
    {s, Rectangle[{i, j+1}, {i+1, j+2}]}
```

In the function `line[s,mp,p,MP,tape]` formal parameters are used in the following sense:

s is the actual state,

mp, *p* and *MP* denote the minimal index, actual index and the maximal index of non-blank cells, and

tape means the list corresponding to the global array *TAPE*, generated by the expression *TapeToList*[*TAPE*].

More precisely, the function `line[s,mp,p,MP,tape]` uses five parameters, which are defined by the actual situation (2.3) of the machine, and generates the list of two elements. The first element is generated by the function `Headc[p,j++,Color[s]]`, where the initial value for variable *j* from global environment is equal to 0. The second element is the list of graphics objects `cellc[i,j,g]`, where *i* runs through values from *mp* to *MP*. This list is generated applying the standard MATHEMATICA function *Table*. Parameter *g* is defined in this way: if *i* is not equal to the index *p* of the scanned cell, then $g = tape[[i - mp + 1]]$; otherwise, $g = \{tape[[i - mp + 1]], s\}$. In this way, in the case $i = p$ function *celc* generates text corresponding to the contents of $tape[[i - mp + 1]]$; otherwise, it generates text $\{tape[[i - mp + 1]], s\}$.

```
line[{s_, mp_, p_, MP_, tape_}] :=
  { Headc[p, (j++), Color[s]],
    Table[cellc[i,j,
                If[i!=p,
                  tape[[i-mp+1]],     (* Then *)
                  {tape[[i-mp+1]],s}  (* Else *)
              ] ],
          {i,mp,MP}
    ]
  }
```

The result of the function *Color*[*s*] is the graphic directive *RGBColor*, dependent from the actual state *s*. For example, it can be defined by means of the following sequence of assignments:

```
Needs["Graphics`Colors`"];
Color[_] := Gray;
Color[A] = Blue;
```

```
Color[B] = Red;
Color[C] = Cyan;
Color[D] = Pink;
Color[E] = Magenta;
Color[F] = Yellow;
Color[G] = Green;
Color[H] = Brown;
```

The procedure `DumpHistory` gives the tabular representation of actions performed by the Turing machine, and stored into the list *History*. In this procedure we apply the function *line* to each element contained in the list produced by reverting the list *History*. In this way, we obtain 2*D* graphical representation of the history of the evaluation. Graphical representation represents the filled part of the tape, scanned cells as well as the actual states. Each row of the list *History* is represented by a row in 2*D* graphical representation.

```
DumpHistory := (
  j = 0;
  Show[Graphics[line /@ Reverse[History]],
           AspectRatio -> Automatic,ImageSize -> {600,600}]
               );
```

Example 2.5. *Consider the set of rules T as in Example 2.2:*

```
In[5]:=T={{A,"0"}->{A,"1",R},{A,"1"}->{A,"1",R},{A,"#"}->{B,"  ",L}}
```

*Assume that the contents of the tape is defined by the string "*110101010*". Then we use the expression Compute*[*T*,*"*110101010*"*] and the procedure DumpHistory to produce* 2*D graphical representation of the history of the evaluation which is represented by the list in Example 2.3:*

```
In[6]:= Compute[T, "110101010"]
In[7]:= DumpHistory;
```

In this picture, contents of any scanned cell is represented as two-element list and colored. First element of such a list represents the contents of the tape, while the second element is the actual state. Let us mention that the states of the machine are also indicated by the color which is defined by means of the function Color.

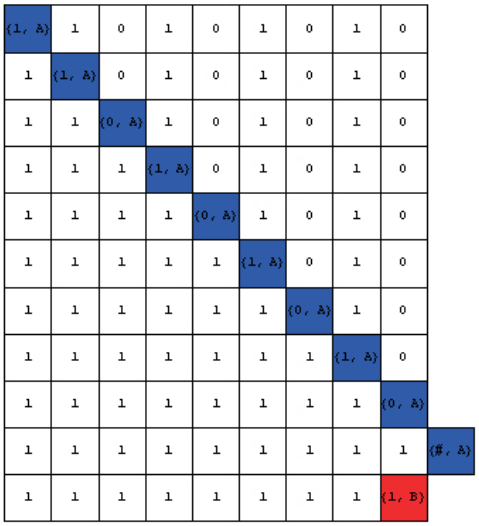

FIG. 2.1: Simulation of the deterministic Turing machine

3. Correlations between deterministic and non-deterministic Turing machine

An apparently more radical reformulation of the notion of Turing machine allows the machine to explore alternative computations in parallel. If the machine specified multiple transitions for a given state/symbol pair, all the resulting computations are continued in parallel. One way to visualize this is that the machine spawns an exact copy of itself and the tape for each alternative available transition.

A necessary quantity for the useful simulation of the non-deterministic Turing machine is the number of corresponding deterministic Turing machines, i.e. the number of tapes in non-deterministic machine.

In the following definition we introduce the notion called *similarity rules*.

Definition 3.1. *Two rules* $\{A_i, a_i\}->\{B_i, b_i, D_i\}$ *and* $\{A_j, a_j\}->\{B_j, b_j, D_j\}$ *satisfy the* **similarity rules** *relation if* $A_i = A_j$ *and* $a_i = a_j$. *This fact we denote by*

$$\{A_i, a_i\}->\{B_i, b_i, D_i\} \sim \{A_j, a_j\}->\{B_j, b_j, D_j\} \Leftrightarrow A_i = A_j \wedge a_i = a_j.$$

It is clear that $\sim$ is the equivalence relation. Assume that there exists p disjoint classes. Denote a class of equivalence corresponding to the situation $\{A_j, a_j\}$ by

$$T_j = \{\{A_j, a_j\}->\{B_{j_1}, b_{j_1}, D_{j_1}\}, \ldots, \{A_j, a_j\}->\{B_{j_{n_j}}, b_{j_{n_j}}, D_{j_{n_j}}\}\}, \quad 1 \le j \le p. \tag{3.1}$$

It is easy to see $\bigcup_{j=1}^{p} T_j = T$.

Theorem 3.1. **(a)** *The number of tapes corresponding the non-deterministic Turing machine M defined in (1.1) is determined by the number*

$$n = \prod_{i=1}^{p} n_i \tag{3.2}$$

where p is the number of equivalence classes T_i of the form (3.1) in T and n_i is the number of rules in class T_i, $i = 1, \ldots, p$.

(b) *An arbitrary from generated tapes uses the rules of the form*

$$\{r_{1,i_1}, \ldots, r_{p,i_p}\},\ r_{j,i_j} \in T_j,\ 1 \le i_j \le n_j,\ 1 \le j \le p, \tag{3.3}$$

where r_{j,i_j} denotes i_j-th rule from the class T_j.

(c) *The total number of rules in M is equal to $n \cdot p$.*

Proof. **(a), (b)** The proof of these parts proceeds by the mathematical induction on the number p of classes. By $T^{(k)}$ we denote the sublist containing the first k classes from T. Turing machine corresponding to $T^{(k)}$ is denoted by M_k.

1. For $p = 1$, complete list $T^{(1)}$ is of the form

$$\begin{aligned} T_1 &= \{\{A_1, a_1\} \text{-> } \{B_{1_1}, b_{1_1}, D_{1_1}\}, \cdots, \{A_1, a_1\} \text{-> } \{B_{1_{n_1}}, b_{1_{n_1}}, D_{1_{n_1}}\}\} \\ &= \{r_{1,1}, \ldots, r_{1,n_1}\}, \end{aligned}$$

and the number of rules in $T^{(1)}$ is n_1. Therefore, M_1 has n_1 tapes, whose rules are of the form

$$\{r_{1,i_1}\},\ 1 \le i_1 \le n_1.$$

2. Suppose that the claim is valid for machine M_{k-1}. The number of tapes for this machine is $\prod_{i=1}^{k-1} n_i$. In view of (3.3), rules on the generated tapes possess the form

$$\{r_{1,i_1}, \ldots, r_{k-1,i_{k-1}}\},\ r_{j,i_j} \in T_j,\ 1 \le i_j \le n_j,\ 1 \le j \le k-1. \tag{3.4}$$

3. If we add k-th class T_k into $T^{(k-1)}$ with n_k elements $\{r_{k,1}, \ldots, r_{k,n_k}\}$, we obtain the list of rules $T^{(k)}$ and corresponding Turing machine M_k. for each rule $r_{k,i_k} \in T_k$, $1 \le i_k \le n_k$ there exists exactly $\prod_{i=1}^{k-1} n_i$ tapes in M_{k-1}. It means that M has $\prod_{i=1}^{k} n_i$ tapes, whose rules are

$$\{r_{1,i_1}, \ldots, r_{k-1,i_{k-1}}, r_{k,i_k}\},\ r_{j,i_j} \in T_j,\ 1 \le i_j \le n_j,\ 1 \le j \le k.$$

Therefore, the claims are valid for an arbitrary positive integer p.

(c) Follows from **(a)** and **(b)**.

Remark 3.1. *The halting problem is well known (see for example [3, 5]). There cannot exists an algorithm which, for each Turing machine M and each configuration determines whether M will eventually halt. This does not mean that we cannot determine whether a specific M in a specific configuration will halt. A Turing machine can go into an infinite loop. In this case, the number n of tapes defined in (3.2) can be finite or infinite. In the following statement we show that for any finite set Q and any finite alphabet Γ the number of tapes is also finite.*

Theorem 3.2. *If $|Q| < \infty$, $|\Gamma| < \infty$, then the Turing machine M defined in (1.1) satisfies:*

(a) *the number n of tapes corresponding to M is limited by*

$$n \le (|Q| \cdot (|\Gamma| - 1) * 3)^{|Q|\cdot|\Gamma|} < \infty; \tag{3.5}$$

(b) *the total number of rules in M is bounded by*

$$n \cdot p \le (|Q| \cdot (|\Gamma| - 1) * 3)^{|Q|\cdot|\Gamma|} \cdot |Q| \cdot |\Gamma| < \infty. \tag{3.6}$$

Proof. **(a)** Any class T_j in (3.1) is defined by the situation $\{A_j, a_j\}$, $A_j \in Q$, $a_j \in \Gamma$. Therefore, the number of classes satisfies

$$p \le |Q| \cdot |\Gamma| < \infty. \tag{3.7}$$

Since $|Q| < \infty, |\Gamma| < \infty$, each number $n_j = |T_j|$, $j = 1, \ldots, p$ is finite, and bounded by

$$n_j \le |Q| \cdot |\Sigma| \cdot 3 = |Q| \cdot (|\Gamma| - 1) \cdot 3.$$

Therefore, in view of (3.6) and (3.7) we have

$$n = \prod_{j=1}^{p} n_j \le (|Q| \cdot (|\Gamma| - 1) \cdot 3)^p \le (|Q| \cdot (|\Gamma| - 1) \cdot 3)^{|Q|\cdot|\Gamma|} < \infty.$$

(b) This part of Theorem follows from part **(c)** of Theorem 3.1, (3.5) and (3.7).

Corollary 3.1. *The number of rules which specify the behavior of any deterministic Turing machine (1.1) is limited by*

$$|Q||\Gamma|.$$

Proof. In the deterministic case we have $n = \prod_{i=1}^{p} n_i = 1$, so that in view of (3.7) the number of rules is bounded by

$$p \le |Q||\Gamma|.$$

Remark 3.2. *Corollary 3.1 is known result from [4].*

Example 3.1. *The equivalence classes corresponding to the list of rules*

```
T={{A,"#"}->{B,"a",R},{B,"#"}->{C,"a",R},{C,"#"}->{G,"a",R},
   {A,"#"}->{D,"b",R},{D,"#"}->{F,"b",R},{G,"#"}->{G," ",L},
   {F,"#"}->{F," ",L}}
```

are equal to

```
T1={{A,#}->{B,a,R},{A,#}->{D,b,R}},
T2={B,#}->{C,a,R},T3={C,#}->{G,a,R},
T4={D,#}->{F,b,R},T5={F,#}->{F, ,L},T6={G,#}->{G, ,L}
```

Their cardinal numbers are $2, 1, 1, 1, 1, 1$*, respectively. In view of Theorem 3.1, the non-deterministic Turing machine defined by T consists of two deterministic Turing machines, whose rules are of the form* (3.3)*. Hence, these rules are defined by the first and the second element of the following list, respectively.*

```
{{{A,#}->{D,b,R},{B,#}->{C,a,R},{C,#}->{G,a,R},{D,#}->{F,b,R},
  {F,#}->{F, ,L},{G,#}->{G, ,L}},
 {{A,#}->{B,a,R},{B,#}->{C,a,R},{C,#}->{G,a,R},{D,#}->{F,b,R},
  {F,#}->{F, ,L},{G,#}->{G, ,L}}}
```

4. Simulation of non-deterministic Turing machine

Our implementation of non-deterministic Turing machine is based on this main idea: replace a given non-deterministic Turing machine by the corresponding set of deterministic Turing machines.

The formal parameter of the function `Clases[T_]` is the list of rules of the given non-deterministic machine. Its output is the list *TList*, which contains classes of the form (3.1). Standard function `Union[list]` gives a sorted version of a list, in which all duplicated elements have been dropped.

```
Clases[T_] := Module[{TT, TList, i},
  TT=Union[T];
  TList={{TT[[1]]}};
  For[i=2, i<=Length[TT], i++,
     If [TT[[i-1, 1]] =!= TT[[i, 1]],  (* Then *)
       AppendTo[TList, {TT[[i]]}];
       ,   (* Else *)
       AppendTo[TList[[Length[TList]]], TT[[i]]];
     ];
  ];
  Return[TList];
];
```

Example 4.1. *The similarity rules classes corresponding to list*

```
T={{A,"#"}->{G," ",L},{A,"0"}->{A,"1",R},{A,"0"}->{A,"0",R},
   {A,"5"}->{A,"2",R},{A,"3"}->{A,"1",R},{A,"3"}->{A,"2",R},
   {A,"3"}->{A,"3",R},{A,"1"}->{A,"0",R},{A,"5"}->{A,"1",R},
   {A,"4"}->{A,"2",R},{A,"4"}->{A,"3",R},{A,"1"}->{A,"4",R},
   {A,"2"}->{A,"1",R},{A,"2"}->{A,"2", R}}
```

are contained into the following list

```
TList={ {{A,#}->{G, ,L}},  {{A,0}->{A,0,R}, {A,0}->{A,1,R}},
{{A,1}->{A,0,R},{A,1}->{A,4,R}},   {{A,2}->{A,1,R}, {A,2}->{A,2,R}},
{{A,3}->{A,1,R}, {A,3}->{A,2,R}, {A,3}->{A,3,R}},
{{A,4}->{A,2,R}, {A,4}->{A,3,R}},
{{A,5}->{A,1,R}, {A,5}->{A,2,R}}  }
```

Their cardinal numbers are $1,2,2,2,3,2,2$*, respectively. The number of tapes in* M *is* $1 \cdot 2 \cdot 2 \cdot 2 \cdot 3 \cdot 2 \cdot 2 = 96$.

We now describe the simulation of the non-deterministic Turing machine. Denote by T the list of rules.

The following standard functions for list operations are used [13]:

`Join[list`$_1$`, list`$_2$`, ... ]` concatenates lists together.

`Flatten[list, n]` flattens nested lists to level n.

Rules corresponding to generated deterministic Turing machines are placed into the list *ND*. List *ND* is generated using the classes *Cls* of the *similarity rules* relation. List *Cls* is the output of the function *Clases*. Into the cycle

```
ND=Table[
     Join[ND[[k]],{Cls[[i,j]]}],{j,Length[Cls[[i]]]},{k,Length[ND]}
   ];
```

we concatenate all elements from the class T_i to each elements `ND[[k]]` of the list *ND*.

In the starting moment all of generated deterministic machines possess identical tapes, filled by the array *TAPE*. Assume that the array *TAPE* is produced applying the function *InitTape*[*word*], where *word* is a given string. Therefore, it is sufficient to evaluate the expressions

```
Compute[ND[[i]], word]; DumpHistory
```

for each $i = 1, \ldots, Length[ND]$.

```
DumpND[T_] := Module[{ND, Cls, i, j, k},
 Cls=Clases[T];
 ND=Table[{Cls[[1, j]]}, {j, Length[Cls[[1]]]}];
 For[i=2, i<=Length[Cls], i++,
```

```
   ND=Table[
     Join[ND[[k]],{Cls[[i,j]]}],{j,Length[Cls[[i]]]},{k,Length[ND]}
   ];
   ND=Flatten[ND, 1];
 ];
 For[i=1, i<=Length[ND], i++,
    Print["Tape",i];  Compute[ND[[i]],word]; DumpHistory
 ]
];
```

Example 4.2. *Consider the following list of rules T and value of global variable word.*

```
T={ {A,"#"}->{B,"a",R},{B,"#"}->{C,"a",R},{C,"#"}->{G,"a",R},
    {A,"#"}->{D,"b",R},{D,"#"}->{F,"b",R},
    {G,"#"}->{G," ",L},{F,"#"}->{F," ",L}};
 word="#";
 DumpND;

Tape1
History={{A,1,1,1,{#}},{D,1,2,2,{b,#}},{F,1,3,3,{b,b,#}},
            {F,1,2,2,{b,b}}}
```

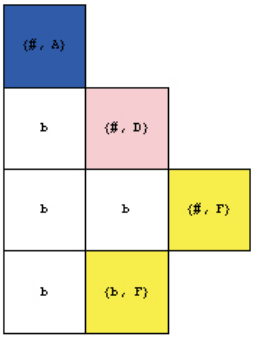

FIG. 4.1: Simulation of nondeterministic Turing machine corresponding to Tape1

```
Tape2
History = {{A,1,1,1,{#}},{B,1,2,2,{a,#}},{C,1,3,3,{a,a,#}},
           {G,1,4,4,{a,a,a,#}},{G,1,3,3,{a,a,a}}}
```

Example 4.3. *In this example we verify that non-deterministic Turing machine reduces to identical deterministic in the case* $p = 1$.

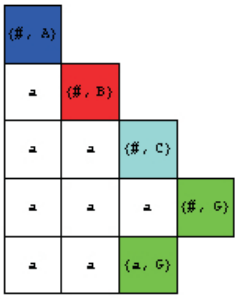

FIG. 4.2: Simulation of nondeterministic Turing machine corresponding to Tape2

```
T={{A,"0"}->{A,"0",R},{A,"1"}->{A,"0",R},{A,"#"}->{G," ",L}};
word="101";
DumpND;

Tape1
History = {{A,1,1,3,{1,0,1}},{A,1,2,3,{0,0,1}},{A,1,3,3,{0,0,1}},
           {A,1,4,4,{0,0,0,#}},{G,1,3,3,{0,0,0}}}
```

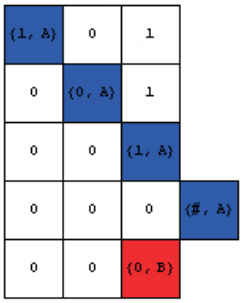

FIG. 4.3: Nondeterministic Turing machine reduces to deterministic

Example 4.4. *In this example we apply our simulator of nondeterministic Turing machine to the list of rules*

```
T={{A,"#"}->{C,"1",L}, {A,"1"}->{B,"#",R}, {A,"1"}->{B,"#",L},
    {B,"#"}->{A,"1",R}, {B,"1"}->{C,"1",R}, {C,"#"}->{B,"1",R},
    {C,"1"}->{A,"#",L},{C,"1"}->{A,"#",R}};
```

and start the evaluation from the blank tape:

```
word = "#";
```

This machine spawns four deterministic machines, whose evaluations are illustrated by the following picture:

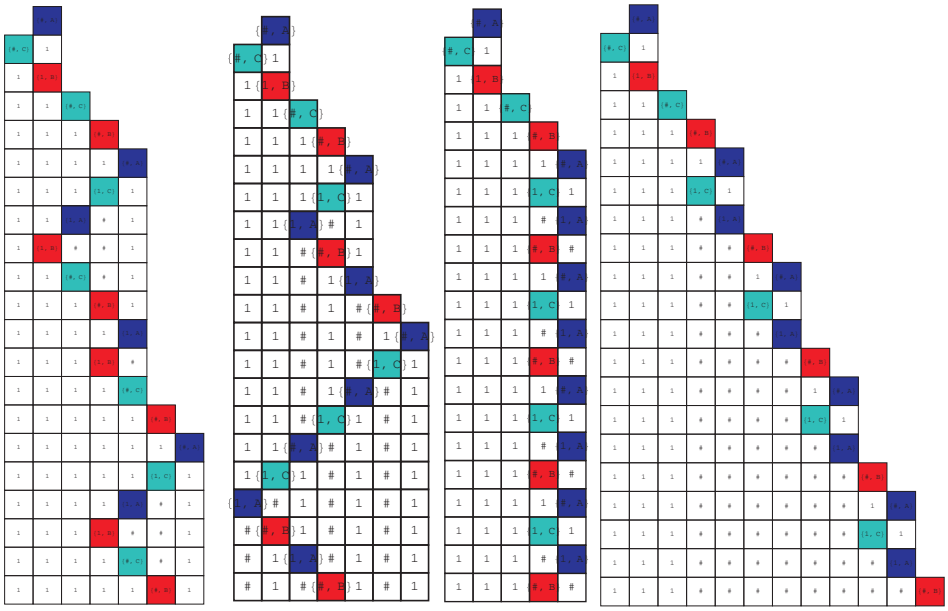

FIG. 4.4: Four deterministic machines initiated by the nondeterministic machine

Remark 4.1. *With respect to our implementation of deterministic Turing machines, in* [15] *the content of the tape is represented by a color, whereas the head has several possible states, indicated by the directions of the arrows.*

5. Conclusion

An efficient implementation of the deterministic Turing machine in the package MATHEMATICA is described. Possibility of symbolic processing in MATHEMATICA are used. Besides the symbolic possibilities, we use MATHEMATICA power in the representation of two-dimensional graphical images.

The number of deterministic Turing machines corresponding to a given nondeterministic Turing machine is determined in (3.2). We proved that this number is bounded, regardless the halting problem. Also, it is verified that each tape of the machine possess a certain number of rules defined by (3.3).

Using these theoretical results we derive corresponding implementation of nondeterministic Turing machine.

REFERENCES

1. N. Blachman: *Mathematica: A Practical Approach*. Englewood Cliffs, New Jersey: Prentice-Hall, 1992.
2. N. Cutland: *Computability: An Introduction to Recursive Functions Theory*. Cambridge University Press, Cambridge, London, New York, Sydney,1986.
3. M. D. Davies and E. J. Weyuker: *Computability, Complexity, and Languages*. Academic Press, New York, London, Paris, 1983.
4. W. W. Eric: *Turing Machine, From MathWorld – A Wolfram Web Resource*. http://mathworld.wolfram.com/TuringMachine.html.
5. D. Gries: *Compiler Construction for Digital Computers*. John Wiley & Sons, Inc., New York, London, Sydney, Toronto, 1971.
6. J. Hertel: *Quantum turing machine simulator*. The Mathematica Journal, Vol. 8, Issue 3, Wolfram Media, Inc., 2002.
7. J. E. Hopcroft and J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation*, AddisonWesley Publishing Co., Reading Massachusetts, 1979.
8. J. E. Hopcroft, R. Motwani, and J. D. Ullman: *Introduction to Automata Theory, Languages, and Computation*. 2nd ed., Addison-Wesley, 2001.
9. R. Maeder: *Programming in Mathematica*. 3rd ed., Redwood City, California: Adisson-Wesley, 1996
10. R. McNaughton: *Elementary Computability, Formal Language and Automata*. Prentice-Hall, 1982.
11. J. P. Tremblay and P. G. Sorenson: *The Theory and Practice of Compiler Writing*. McGraw-Hill Book Company, New York, 1985.
12. A. M. Turing: *On computable numbers, with an application to the Entscheidungsproblem*. Proc. London Math. Soc. Ser 2, 1936.
13. S. Wolfram: *The Mathematica Book*. 4th ed., Wolfram Media/Cambridge University Press, 1999.
14. S. Wolfram: *Mathematica Book, Version 3.0*. Addison-Wesley Publishing Co, Redwood City, California, 1997.
15. S. Wolfram: *A New Kind of Science*. Wolfram Media, 2002; pp. 78–81, 888–889, 110, 1119, 1128.
16. www.cheran.ro/vturing.
17. www.userpages.wittenberg.edu/bshelburne/Turing.htm.
18. www.csee.umbc.edu/~squire/cs451_ sim.html.

Saša V. Vukašinović
Faculty of Science and Mathematics
Department of Mathematics and Informatics
Višegradska 33, 18000 Niš, Serbia
sasavukasinovic_7@msn.com

Predrag S. Stanimirović
Faculty of Science and Mathematics
Department of Mathematics and Informatics
Višegradska 33, 18000 Niš, Serbia
pecko@pmf.ni.ac.rs

Marko D. Petković
Faculty of Science and Mathematics
Department of Mathematics and Informatics
Višegradska 33, 18000 Niš, Serbia
dexter_of_nis@neobee.net

Miroslav D. Ćirić
Faculty of Science and Mathematics
Department of Mathematics and Informatics
Višegradska 33, 18000 Niš, Serbia
mciric@pmf.ni.ac.rs