**Caveat lector:** This note is not even a first draft, but more of a rough sketch, with many topics still to be written and/or unwritten. But the semester is over, so it's time to put it down. Please send bug reports and suggestions to jeffe@illinois.edu.

*Any sufficiently advanced technology is indistinguishable from magic.*
— Arthur C. Clarke, "Hazards of Prophecy: The Failure of Imagination" (1962)

*Any technology that is distinguishable from magic is insufficiently advanced.*
— Barry Gehm, quoted by Stan Schmidt in *ANALOG* magazine (1991)

# 8 Universal Models of Computation

★★★

Remind about the Church-Turing thesis.
There is some confusion here between **universal models of computation** and the somewhat wider class of **undecidable problems/languages**.

## 8.1 Universal Turing Machines

The pinnacle of Turing machine constructions is the ***universal*** Turing machine. For modern computer scientists, it's useful to think of a universal Turing machine as a "Turing machine *interpreter* written in Turing machine". Just as the input to a Python interpreter is a string of Python source code, the input to our universal Turing machine $U$ is a string $\langle M, w\rangle$ that encodes both an arbitrary Turing machine $M$ and a string $w$ in the input alphabet of $M$. Given these encodings, $U$ simulates the execution of $M$ on input $w$; in particular,

- $U$ accepts $\langle M, w\rangle$ if and only if $M$ accepts $w$.
- $U$ rejects $\langle M, w\rangle$ if and only if $M$ rejects $w$.

In the next few pages, I will sketch a universal Turing machine $U$ that uses the input alphabet $\{0, 1, [, ], \bullet, |\}$ and a somewhat larger tape alphabet. However, I do *not* require that the Turing machines that $U$ simulates have similarly small alphabets, so we first need a method to encode *arbitrary* input and tape alphabets.

### Encodings

Let $M = (\Gamma, \square, \Sigma, Q, \mathit{start}, \mathit{accept}, \mathit{reject}, \delta)$ be an arbitrary Turing machine, with a single half-infinite tape and a single read-write head. (I will consistently indicate the states and tape symbols of $M$ in *slanted green* to distinguish them from the upright red states and tape symbols of $U$.)

We encode each symbol $a \in \Gamma$ as a unique string $|a|$ of $\lceil \lg(|\Gamma|)\rceil$ bits. Thus, if $\Gamma = \{0, 1, \$, x, \square\}$, we might use the following encoding:

$$\langle 0\rangle = 001, \qquad \langle 1\rangle = 010, \qquad \langle \$\rangle = 011, \qquad \langle x\rangle = 100, \qquad \langle \square\rangle = 000.$$

The input string $w$ is encoded by its sequence of symbol encodings, with separators $\bullet$ between every pair of symbols and with brackets [ and ] around the whole string. For example, with this encoding, the input string $001100$ would be encoded on the input tape as

$$\langle 001100\rangle = [001\bullet001\bullet010\bullet010\bullet001\bullet001]$$

Similarly, we encode each state $q \in Q$ as a distinct string $\langle q \rangle$ of $\lceil \lg |Q| \rceil$ bits. Without loss of generality, we encode the start state with all 1s and the reject state with all 0s. For example, if $Q = \{\mathit{start}, \mathit{seek1}, \mathit{seek0}, \mathit{reset}, \mathit{verify}, \mathit{accept}, \mathit{reject}\}$, we might use the following encoding:

$$\langle \mathit{start} \rangle = 111 \qquad \langle \mathit{seek1} \rangle = 010 \qquad \langle \mathit{seek0} \rangle = 011 \qquad \langle \mathit{reset} \rangle = 100$$
$$\langle \mathit{verify} \rangle = 101 \qquad \langle \mathit{accept} \rangle = 110 \qquad \langle \mathit{reject} \rangle = 000$$

We encode the machine $M$ itself as the string $\langle M \rangle = [\langle \mathit{reject} \rangle \bullet \langle \square \rangle] \langle \delta \rangle$, where $\langle \delta \rangle$ is the concatenation of substrings $[\langle p \rangle \bullet \langle a \rangle | \langle q \rangle \bullet \langle b \rangle \bullet \langle \Delta \rangle]$ encoding each transition $\delta(p, a) = (q, b, \Delta)$ such that $q \neq \mathrm{reject}$. We encode the actions $\Delta = \pm 1$ by defining $\langle -1 \rangle := 0$ and $\langle +1 \rangle := 1$. Conveniently, every transition string has exactly the same length. For example, with the symbol and state encodings described above, the transition $\delta(\mathit{reset}, \$) = (\mathit{start}, \$, +1)$ would be encoded as

$$[100 \bullet 011 | 001 \bullet 011 \bullet 1].$$

Our first example Turing machine for recognizing $\{0^n 1^n 0^n \mid n \geq 0\}$ would be represented by the following string (here broken into multiple lines for readability):

```
[000•000][[001•001|010•011•1][001•100|101•011•1]
          [010•001|010•001•1][010•100|010•100•1]
          [010•010|011•100•1][011•010|011•010•1]
          [011•100|011•100•1][011•001|100•100•1]
          [100•001|100•001•0][100•010|100•010•0]
          [100•100|100•100•0][100•011|001•011•1]
          [101•100|101•011•1][101•000|110•000•0]]
```

Finally, we encode any *configuration* of $M$ on $U$'s work tape by alternating between encodings of states and encodings of tape symbols. Thus, each tape cell is represented by the string $[\langle q \rangle \bullet \langle a \rangle]$ indicating that (1) the cell contains symbol $a$; (2) if $q \neq \mathit{reject}$, then $M$'s head is located at this cell, and $M$ is in state $q$; and (3) if $q = \mathit{reject}$, then $M$'s head is located somewhere else. Conveniently, each cell encoding uses exactly the same number of bits. We also surround the entire tape encoding with brackets [ and ].

For example, with the encodings described above, the initial configuration $(\mathit{start}, \underset{\blacktriangle}{0}01100, 0)$ for our first example Turing machine would be encoded on $U$'s tape as follows.

$$\langle \mathit{start}, \underset{\blacktriangle}{0}01100, 0 \rangle = \underbrace{[[111 \bullet 001][000 \bullet 001][000 \bullet 010][000 \bullet 010][000 \bullet 001][000 \bullet 001]]}_{\textbf{start } 0 \quad \text{reject } 0 \quad \text{reject } 1 \quad \text{reject } 1 \quad \text{reject } 0 \quad \text{reject } 0}$$

Similarly, the intermediate configuration $(\mathit{reset}, \$0x\underset{\blacktriangle}{1}x0, 3)$ would be encoded as follows:

$$\langle \mathit{reset}, \$\$x\underset{\blacktriangle}{1}x0, 3 \rangle = \underbrace{[[000 \bullet 011][000 \bullet 011][000 \bullet 100][010 \bullet 010][000 \bullet 100][000 \bullet 001]]}_{\text{reject } \$ \quad \text{reject } 0 \quad \text{reject } x \quad \textbf{reset } 1 \quad \text{reject } x \quad \text{reject } 0}$$

### Input and Execution

Without loss of generality, we assume that the input to our universal Turing machine $U$ is given on a separate read-only ***input tape***, as the encoding of an arbitrary Turing machine $M$ followed by an encoding of its input string $x$. Notice the substrings [[ and ]] each appear only only once on the input tape, immediately before and after the encoded transition table, respectively. $U$ also has a read-write ***work tape***, which is initially blank.

We start by initializing the work tape with the encoding $\langle start, x, 0\rangle$ of the initial configuration of $M$ with input $x$. First, we write `[[`$\langle start\rangle$`•`. Then we copy the encoded input string $\langle x\rangle$ onto the work tape, but we change the punctuation as follows:

- Instead of copying the left bracket `[`, write `[[`$\langle start\rangle$`•`.
- Instead of copying each separator `•`, write `][`$\langle reject\rangle$`•`
- Instead of copying the right bracket `]`, write two right brackets `]]`.

The state encodings $\langle start\rangle$ and $\langle reject\rangle$ can be copied directly from the beginning of $\langle M\rangle$ (replacing `0`s for `1`s for $\langle start\rangle$). Finally, we move the head back to the start of $U$'s tape.

At the start of each step of the simulation, $U$'s head is located at the start of the work tape. We scan through the work tape to the unique encoded cell `[`$\langle p\rangle$`•`$\langle a\rangle$`]` such that $p \neq reject$. Then we scan through the encoded transition function $\langle\delta\rangle$ to find the unique encoded tuple `[`$\langle p\rangle$`•`$\langle a\rangle$`|`$\langle q\rangle$`•`$\langle b\rangle$`•`$\langle\Delta\rangle$`]` whose left half matches our the encoded tape cell. If there is no such tuple, then $U$ immediately halts and rejects. Otherwise, we copy the right half $\langle q\rangle$`•`$\langle b\rangle$ of the tuple to the work tape. Now if $q = accept$, then $U$ immediately halts and accepts. (We don't bother to encode *reject* transformations, so we know that $q \neq reject$.) Otherwise, we transfer the state encoding to either the next or previous encoded cell, as indicated by $M$'s transition function, and then continue with the next step of the simulation.

During the final state-copying phase, we ever read two right brackets `]]`, indicating that we have reached the right end of the tape encoding, we replace the second right bracket with `[`$\langle reject\rangle$`•`$\langle\square\rangle$`]]` (mostly copied from the beginning of the machine encoding $\langle M\rangle$) and then scan back to the left bracket we just wrote. This trick allows our universal machine to *pretend* that its tape contains an infinite sequence of *encoded* blanks `[`$\langle reject\rangle$`•`$\langle\square\rangle$`]` instead of *actual* blanks `□`.

### Example

As an illustrative example, suppose $U$ is simulating our first example Turing machine $M$ on the input string `001100`. The execution of $M$ on input $w$ eventually reaches the configuration (*seek1*, `$$x1x0`, 3). At the start of the corresponding step in $U$'s simulation, $U$ is in the following configuration:

`[[000•011][000•011][000•100][010•010][000•100][000•001]]`

First $U$ scans for the first encoded tape cell whose state is not *reject*. That is, $U$ repeatedly compares the first half of each encoded state cell on the work tape with the prefix `[`$\langle reject\rangle$`•` of the machine encoding $\langle M\rangle$ on the input tape. $U$ finds a match in the fourth encoded cell.

`[[000•011][000•011][000•100][010•010][000•100][000•001]]`

Next, $U$ scans the machine encoding $\langle M\rangle$ for the substring `[010•010` matching the current encoded cell. $U$ eventually finds a match in the left size of the the encoded transition `[010•010|011•100•1]`. $U$ copies the state-symbol pair `011•100` from the right half of this encoded transition into the current encoded cell. (The underline indicates which symbols are changed.)

`[[000•011][000•011][000•100][011•100][000•100][000•001]]`

The encoded transition instructs $U$ to move the current state encoding one cell to the right. (The underline indicates which symbols are changed.)

```
[[000•011][000•011][000•100][000•100][011•100][000•001]]
```

Finally, $U$ scans left until it reads two left brackets `[[`; this returns the head to the left end of the work tape to start the next step in the simulation. $U$'s tape now holds the encoding of $M$'s configuration (*seek0*, *$$xxx0*, 4), as required.

```
[[000•011][000•011][000•100][000•100][011•100][000•001]]
```

## 8.2 Two-Stack Machines

A ***two-stack machine*** is a Turing machine with two tapes with the following restricted behavior. At all times, on each tape, every cell to the right of the head is blank, and every cell at or to the left of the head is non-blank. Thus, a head can only move right by writing a non-blank symbol into a blank cell; symmetrically, a head can only move left by erasing the rightmost non-blank cell. Thus, each tape behaves like a stack. To avoid underflow, there is a special symbol at the start of each tape that cannot be overwritten. Initially, one tape contains the input string, with the head at its *last* symbol, and the other tape is empty (except for the start-of-tape symbol).

★★★

Simulate a doubly-infinite tape with two stacks, one holding the tape contents to the left of the head, the other holding the tape contents to the right of the head. For each transition of a standard Turing machine $M$, the stack machine pops the top symbol off the (say) left stack, changes its internal state according to the transition $\delta$, and then either pushes a new symbol onto the right stack, or pushes a new symbol onto the left stack and then moves the top symbol from the right stack to the left stack.

## 8.3 Counter Machines

★★★

A configuration of a $k$-counter machine consists of $k$ non-negative integers and an internal state from some finite set $Q$. The transition function $\delta : Q \times \{0,+1\}^k \to Q \times \{-1,0,+1\}^k$ takes an internal state and the signs of the counters as input, and produces a new internal state and changes to counters as output.

- Prove that any Turing machine can be simulated by a three-counter machine. One counter holds the binary representation of the tape after the head; another counter holds the reversed binary representation of the tape before the head. Implement transitions via halving, doubling, and parity, using the third counter for scratch work.
- Prove that two counters can simulate three. Store $2^a 3^b 5^c$ in one counter, use the other for scratch work.
- Prove that a three-counter machine can compute any computable function: Given input $(n,0,0)$, we can compute $(f(n),0,0)$ for *any* computable function $f$. First transform $(n,0,0)$ to $(2^n,0,0)$ using all three counters; then run two- (or three-)counter TM simulation to obtain $(2^{f(n)},0,0)$; and finally transform $(2^{f(n)},0,0)$ to $(f(n),0,0)$ using all three counters.
- **HARD:** Prove that a two-counter machine cannot transform $(n,0)$ to $(2^n,0)$. [Barzhdin 1963, Yao 1971, Schröpel 1972]

## 8.4 FRACTRAN

★★★

FRACTRAN [Conway 1987]: A one-counter machine whose "program" is a sequence of rational numbers. The counter is initially 1. At each iteration, multiply the counter by the first rational number that yields an integer; if there is no such number, halt.

- Prove that for any computable function $f : \mathbb{N} \to \mathbb{N}$, there is a FRACTRAN program that transforms $2^{n+1}$ into $3^{f(n)+1}$, for all natural numbers $n$.
- Prove that every FRACTRAN program, given the integer 1 as input, either outputs 1 or loops forever. It follows that there is no FRACTRAN program for the increment function $n \mapsto n+1$.

## 8.5 Post Correspondence Problem

Given $n$ of pairs of strings $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$, is there a finite sequence of integers $(i_1, i_2, \ldots, i_k)$ such that $x_{i_1} x_{i_2} \cdots x_{i_k} = y_{i_1} y_{i_2} \cdots y_{i_k}$? For notation convenience, we write each pair vertically as $\left[\begin{smallmatrix} x \\ y \end{smallmatrix}\right]$ instead of horizontally as $(x, y)$. For example, given the string pairs

$$a = \begin{bmatrix} 0 \\ 100 \end{bmatrix},\ b = \begin{bmatrix} 01 \\ 00 \end{bmatrix},\ c = \begin{bmatrix} 110 \\ 11 \end{bmatrix},$$

we should answer TRUE, because

$$cbca = \begin{bmatrix} 110 \\ 11 \end{bmatrix}\begin{bmatrix} 01 \\ 00 \end{bmatrix}\begin{bmatrix} 110 \\ 11 \end{bmatrix}\begin{bmatrix} 0 \\ 100 \end{bmatrix}$$

gives us 110110100 for both concatenations. As more extreme examples, the shortest solutions for the input

$$a = \begin{bmatrix} 0 \\ 001 \end{bmatrix},\ b = \begin{bmatrix} 001 \\ 1 \end{bmatrix},\ c = \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

have length 75; one such solution is $aacaacabbabccaaccaaaacbaabbaacbacbbccbbacbaccbcb$ $acbbacbaccbacbbbacccbabbccbaacaacaaacbabbaacacbccbbabacbcaaccbacabbbbabcccc$ $bcaababaaccbcbbbacccbabbccb$. The shortest solution for the instance

$$a = \begin{bmatrix} 0 \\ 000 \end{bmatrix},\ b = \begin{bmatrix} 0 \\ 0101 \end{bmatrix},\ c = \begin{bmatrix} 01 \\ 1 \end{bmatrix},\ d = \begin{bmatrix} 1111 \\ 10 \end{bmatrix}$$

is the unbelievable $a^2b^8a^4c^{16}ab^4a^2b^4ad^4b^3c^8a^6c^8b^2c^4bc^6d^2a^{18}d^2c^4dcad^2cb^{54}c^3dca^2c^{111}dc$ $a^6d^{28}cb^{17}c^{63}d^{16}c^{16}d^4c^4dc$, which has total length 451. Finally, the shortest solution for the instance

$$a = \begin{bmatrix} 0 \\ 00010 \end{bmatrix},\ b = \begin{bmatrix} 010 \\ 01 \end{bmatrix},\ c = \begin{bmatrix} 100 \\ 0 \end{bmatrix},$$

has length 528.

★★★

The simplest universality proof simulates a tag-Turing machine.

## 8.6 Matrix Mortality

★★★

Given a set of integer matrices $A_1, \ldots, A_k$, is the product of any sequence of these matrices (with repetition) equal to 0? Undecidable by reduction from PCP, even for two $15 \times 15$ matrices or six $3 \times 3$ matrices [Cassaigne, Halava, Harju, Nicolas 2014]

## 8.7 Dynamical Systems

★★★

Ray Tracing [Reif, Tygar, and Yoshida 1994] The configuration of a Turing machine is encoded as the $(x, y)$ coordinates of a light path crossing the unit square $[0,1] \times [0,1]$, where the $x$-(resp. $y$-)coordinate encodes the tape contents to the left (resp. right) of the head. Need either quadratic-surface mirrors or refraction to simulate transitions.

N-body problem [Smith 2006]: Similar idea

Skolem-Pisot reachability: Given an integer vector $x$ and an integer matrix $A$, does $A^n x = (0, \dots)$ for any integer $n$? [Halava, Harju, Hirvensalo, Karhumäki 2005] It's surprising that this problem is undecidable; the similar mortality problem for one matrix is not.

## 8.8 Wang Tiles

★★★

Turing machine simulation is straightforward. *Small* Turing-complete tile sets via affine maps (via two-stack machines) are a little harder.

## 8.9 Combinator Calculus

In the 1920s, Moses Schönfinkel developed what can now be interpreted as a model of computation now called *combinator calculus* or *combinatory logic*. Combinator calculus operates on ***terms***, where every term is either one of a finite number of ***combinators*** (represented here by upper case letters) or an ordered pair of terms. For notational convenience, we omit commas between components of every pair and parentheses around the *left* term in every pair. Thus, SKK(IS) is shorthand for the term $(((\mathtt{S}, \mathtt{K}), \mathtt{K}), (\mathtt{I}, \mathtt{S}))$.

We can "evaluate" any term by a sequence of rewriting rules that depend on its first primitive combinator. Schönfinkel defined three primitive combinators with the following evaluation rules:

- Identity: $\mathtt{I}x \mapsto x$
- Constant: $\mathtt{K}xy \mapsto x$
- Substitution: $\mathtt{S}xyz \mapsto xz(yz)$

Here, $x$, $y$, and $z$ are variables representing unknown but arbitrary terms. "Computation" in the combinator calculus is performed by repeatedly evaluating arbitrary (sub)terms with one of these three structures, until all such (sub)terms are gone.

For example, the term S(K(SI))Kxy (for any terms $x$ and $y$) evaluates as follows:

$$\begin{aligned}
\underline{\mathtt{S}(\mathtt{K}(\mathtt{SI}))\mathtt{K}}\mathtt{x}y &\mapsto \underline{\mathtt{K}(\mathtt{SI})\ \mathtt{x}}(\mathtt{K}x)y && \text{Substitution} \\
&\mapsto \underline{\mathtt{SI}(\mathtt{Kx})\ \mathtt{y}} && \text{Constant} \\
&\mapsto \underline{\mathtt{I}\mathtt{y}}(\mathtt{K}xy) && \text{Substitution} \\
&\mapsto y(\underline{\mathtt{K}\mathtt{x}\mathtt{y}}) && \text{Identity} \\
&\mapsto yx && \text{Constant}
\end{aligned}$$

Thus, we can define a new combinator R := S(K(SI))K that upon evaluation reverses the next two terms: $\mathtt{R}xy \mapsto yx$.

On the other hand, evaluating SII(S(KI)(SII)) leads to an infinite loop:

$$\begin{aligned}
\underline{\texttt{SII(S(KI)(SII))}} &\mapsto \underline{\texttt{I(S(KI)(SII))}}\texttt{(I(S(KI)(SII)))} && \text{Substitution}\\
&\mapsto \texttt{S(KI)(SII)}\underline{\texttt{(I(S(KI)(SII)))}} && \text{Identity}\\
&\mapsto \underline{\texttt{S(KI)(SII)(S(KI)(SII))}} && \text{Identity}\\
&\mapsto \underline{\texttt{KI(S(KI)(SII))}}\texttt{(SII(S(KI)(SII)))} && \text{Substitution}\\
&\mapsto \underline{\texttt{I(SII(S(KI)(SII)))}} && \text{Constant}\\
&\mapsto \texttt{SII(S(KI)(SII))} && \text{Identity}
\end{aligned}$$

★★★ Wikipedia sketches a direct *undecidability* proof. Is there a Turing-completeness proof that avoids λ-calculus?

## Exercises

1. A ***tag-***Turing machine has two heads: one can only read, the other can only write. Initially, the read head is located at the left end of the tape, and the write head is located at the first blank after the input string. At each transition, the read head can either move one cell to the right or stay put, but the write head *must* write a symbol to its current cell and move one cell to the right. Neither head can ever move to the left.

   Prove that any standard Turing machine can be simulated by a tag-Turing machine. That is, given any standard Turing machine $M$, describe a tag-Turing machine $M'$ that accepts and rejects exactly the same input strings as $M$.

2. *(a) Prove that any standard Turing machine can be simulated by a Turing machine with only three states. *[Hint: Use the tape to store an encoding of the state of the machine yours is simulating.]*

   ★(b) Prove that any standard Turing machine can be simulated by a Turing machine with only *two* states.

3. A ***two-dimensional*** Turing machine uses an infinite two-dimensional grid of cells as the tape; at each transition, the head can move from its current cell to any of its four neighbors on the grid. The transition function of such a machine has the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\uparrow, \leftarrow, \downarrow, \rightarrow\}$, where the arrows indicate which direction the head should move.

   (a) Prove that any two-dimensional Turing machine can be simulated by a standard Turing machine.

   (b) Suppose further that we endow our two-dimensional Turing machine with the following additional actions, in addition to moving the head:

   - Insert row: Move all symbols on or above the row containing the head up one row, leaving the head's row blank.
   - Insert column: Move all symbols on or to the right of the column containing the head one column to the right, leaving the head's column blank.
   - Delete row: Move all symbols above the row containing the head down one row, deleting the head's row of symbols.
   - Delete column: Move all symbols the right of the column containing the head one column to the right, deleting the head's column of symbols.

Show that any two-dimensional Turing machine that can add an delete rows can be simulated by a standard Turing machine.

4. A ***binary-tree*** Turing machine uses an infinite binary tree as its tape; that is, *every* cell in the tape has a left child and a right child. At each step, the head moves from its current cell to its Parent, its Left child, or to its Right child. Thus, the transition function of such a machine has the form $\delta : Q \times \Gamma \to Q \times \Gamma \times \{\mathsf{P}, \mathsf{L}, \mathsf{R}\}$. The input string is initially given along the left spine of the tape.

   Show that any binary-tree Turing machine can be simulated by a standard Turing machine.

5. A ***stack-tape*** Turing machine uses an semi-infinite tape, where every cell is actually the top of an independent stack. The behavior of the machine at each iteration is governed by its internal state and the symbol *at the top* of the current cell's stack. At each transition, the head can optionally push a new symbol onto the stack, or pop the top symbol off the stack. (If a stack is empty, its "top symbol" is a blank and popping has no effect.)

   Show that any stack-tape Turing machine can be simulated by a standard Turing machine. (Compare with Problem **??**!)

6. A ***tape-stack*** Turing machine has two actions that modify its work tape, in addition to simply writing individual cells: it can ***save*** the entire tape by pushing in onto a stack, and it can ***restore*** the entire tape by popping it off the stack. Restoring a tape returns the content of every cell to its content when the tape was saved. Saving and restoring the tape do not change the machine's state or the position of its head. If the machine attempts to "restore" the tape when the stack is empty, the machine crashes.

   Show that any tape-stack Turing machine can be simulated by a standard Turing machine.

★★★

- Tape alphabet = $\mathbb{N}$.
  - Read: zero or positive. Write: $+1$, $-1$
  - Read: even or odd. Write: $+1$, $-1$, $\times 2$, $\div 2$
  - Read: positive, negative, or zero. Write: $x + y$ (merge), $x - y$ (merge), 1, 0
- Never three times in a row in the same direction
- Hole-punch TM: tape alphabet $\{\square, \blacksquare\}$, and only $\square \mapsto \blacksquare$ transitions allowed.