Markov Algorithm

CHEN Yuanmi

December 18, 2007

1 Abstract

Markov Algorithm can be understood as a priority string rewriting system. In this short paper we give the definition of Markov algorithm and also several examples. We will then focus on the demonstration of the equivalence of capability in terms of calculation between a Markov algorithm and a Turing machine.

2 Definition of Markov algorithm

In this section, we are going to introduce the formal definition of Markov Algorithm, then we will move on to see some concrete examples.

2.1 Formal definition

Definition 1 (Markov Algorithm). A Markov algorithm is a rewriting system denoted as $(\Sigma, P, P_0)$, where

- $\Sigma$ is the finite set of alphabet.
- $\epsilon$ is the empty word.
- $P$ is a finite set of productions given in a linear order
  \[ \alpha_1 \rightarrow \beta_1, \ldots, \alpha_k \rightarrow \beta_k, \]
  where $\alpha_i, \beta_i \in \Sigma^*$.
- $P_0$ is a subset of $P$, composed of final productions denoted by $\alpha \rightarrow \cdot \beta$.

Example 1 (String searching). $\text{MA}_1 = \{\Sigma, P, P_0\}$.

- $\Sigma = \{0', 1', a'\}$;
- Productions in $P$:
  \begin{align*}
  p_1: \ & a0 \rightarrow 0a \\
  p_2: \ & a1 \rightarrow 0a
  \end{align*}
\[ p_3: \ a \rightarrow \ \varepsilon \]
\[ p_4: \ 1101 \rightarrow 1101a \]

\[ \textbf{Remark: } \text{MA}_1 \text{ searches in input the pattern } "1101". \]

Given a Markov algorithm \( \text{MA} \) as above, we can define a binary relationship \( \Rightarrow_{\text{MA}} \) or simply \( \Rightarrow \), as follows.

**Definition 2.** Let \( \text{MA} \) be a Markov algorithm, \( u, v \in \Sigma^* \), \( u \Rightarrow v \) iff all of the following conditions are satisfied.

1. There exists \( i, 1 \leq i \leq k \), and words \( u_1, u_2 \in \Sigma^* \) such that \( u = u_1 \alpha_i u_2 \), \( v = u_1 \beta_i u_2 \).

2. \( \alpha_j \) is not a subword of \( u \) for all \( j < i \).

3. \( \alpha_i \) occurs as a subword of \( u_1 \alpha_i \) only once.

4. One of the words \( \alpha_1, \ldots, \alpha_k \) is a subword of \( v \), and \( \alpha_i \rightarrow \beta_i \) is not a final production.

**Definition 3.** Let \( u, v \in \Sigma^* \), \( u \Rightarrow v \) iff the conditions 1,2,3 in the previous definition are satisfied, and the condition 4 is violated.

Furthermore, \( u \Rightarrow^* v \) holds if there is a finite sequence of words

\[ u = u_0, u_1, \ldots, u_n, v, n \geq 0 \]

such that \( u_i \Rightarrow u_{i+1} \) holds for \( 0 \leq i \leq n - 1 \), and \( u_n \Rightarrow v \), or else \( u = v \) and none of the words \( \alpha_1, \ldots, \alpha_k \) is a subword of \( u \).

Given the priority order of the substitution rules, we manage to get rid of indeterministic factors in the action of the algorithm. Therefore we have

**Lemma 4.** For all \( u \in \Sigma^* \), there exists at most one \( v \in \Sigma^* \) such that \( u \Rightarrow^* v \)

**Proof.** \( u \Rightarrow^* v \) iff exist a sequence \( u = u_0, u_1, \ldots, u_n, v \), such that \( u_i \Rightarrow u_{i+1} \) for \( 0 \leq i \leq n - 1 \), and \( u_n \Rightarrow v \).

For each \( 0 \leq i \leq n - 1 \), if \( u_i \Rightarrow u_{i+1} \) and \( u_i \Rightarrow u_i' \), suppose \( \alpha_{j_i} \rightarrow \beta_{j_i} \) is the production referred to in the former relation, and \( \alpha_{j_i'} \rightarrow \beta_{j_i'} \) in the latter, then according to the definition of "\( \Rightarrow \)" property 2, we have \( j_i = j_i' \), i.e. \( \alpha_{j_i} = \alpha_{j_i'} \).

Then, since \( \alpha_{j_i} \) is the left-most substring match in \( u_i \) (Property 3), we have \( u_{i+1} = u_{i+1}' \). Inductively and similar reasoning with the last step, we see that if there exists \( v \) such that \( u \Rightarrow^* v \), such \( v \) must be unique. \( \square \)

**Definition 5.** For any word \( u \in \Sigma^* \), if there exist a word \( v \) such that \( u \Rightarrow^* v \), we say that our Markov algorithm \( \text{MA} \) \textit{halts} with \( u \) and \textit{translates} \( u \) into \( v \). If the last rule it applies belongs to \( P_0 \), then we say the algorithm \textit{accepts} the input. Otherwise the machine \textit{rejects} the input.

If a Markov algorithm does not halts with an input, it iterates unlimitedly.
2.2 Several examples

Example 2 (Greatest common deviser). $\text{MA}_2 = \{\Sigma, P, P_0\}$.

- $\Sigma = \{', a', A', B', C', \#\}$.
- Productions in $P$:
  
  $p_1$: $aA \rightarrow Aa$
  $p_2$: $a\#a \rightarrow A\#$
  $p_3$: $a\# \rightarrow \#B$
  $p_4$: $B \rightarrow a$
  $p_5$: $A \rightarrow C$
  $p_6$: $C \rightarrow a$
  $p_7$: $\# \rightarrow \epsilon$
  $p_8$: $\epsilon \rightarrow \epsilon$

- $P_0 = \{p_8\}$

Remark: The production $p_8$ will only be applied once, and it simply ensures that all input will be accepted.

Given the input in form of $"_{x}{aa...a\#}_{y}{aa...a}"$, the algorithm will, thanks to the Euclidean algorithm, halt with the output in the form of $"_{x}{aa...a}_{y}\{d\}$, where $d = \text{g.c.d}(x, y)$. Given other input, the machine halts, though with meaningless outputs.

Example 3 (Multiplication). $\text{MA}_3 = \{\Sigma, P, P_0\}$.

- $\Sigma = \{', a', A', B', C', \#\}$.
- Productions in $P$:
  
  $p_1$: $Ba \rightarrow aB$
  $p_2$: $Aa \rightarrow aBA$
  $p_3$: $a \rightarrow \epsilon$
  $p_4$: $a\# \rightarrow \#A$
  $p_5$: $\#a \rightarrow \#$
  $p_6$: $\# \rightarrow \epsilon$
  $p_7$: $B \rightarrow a$
  $p_8$: $\epsilon \rightarrow \epsilon$

- $P_0 = \{p_8\}$

Remark: Given the input in form of $"_{x}{aa...a\#}_{y}{aa...a}"$, the algorithm will halt with the output in the form of $"_{x\times y}{aa...a}"$. 


3 Equivalence with Turing machine

Intuitively, Turing machine is more agile than Markov algorithm, since its read/write head can move to both directions, in contrast to the left-to-right access of Markov algorithms. Thus, it is easy to understand the principle of how a Turing machine can simulate a Markov algorithm, though the details may be laborious.

On the other hand, we are going to see with a little more labor, we can simulate the Turing machine with the Markov algorithm as well.

3.1 The definition of Turing machine

Definition 6 (Turing machine). A Turing machine is composed of \((Q, \Sigma, E, q_0, F)\), where

- \(Q = q_0, \ldots, q_n\) is a finite set of control states.
- \(\Sigma\) is the alphabet, including an non-erasable mark \(\delta\) at the beginning of the band.
- \(E\) is the finite set of transitions in the form of \((p, a, q, b, x)\) where \(p\) and \(q\) are the states, \(a\) and \(b\) are the symbols and \(x\) belongs to \(\{←, →, □\}\) signifies the moving direction of the read/write head.
- \(q_0 \in Q\) is the initial state of all calculations.
- \(F\) is the set of the final states accepted by the Turing machine.

A Turing machine is deterministic if for all state \(p\) and all symbol \(a\), there is at most one transition in the form of \((p, a, q, b, x)\). Then \(E\) is also called the function of transition.

The Turing machine works with a single band infinitely long. At the beginning, the read/writing head is placed on the symbol \(\delta\). For all instant, it does not exceed the left boundary of the band. Given an input, the Turing machine is said to halt with it, if after a finite series of step the machine transitions come to an end. In this case it either accepts it, if the final state is in \(F\), or rejects it, if the final state is in \(Q \setminus F\).

3.2 Simulation of Turing machine by Markov Algorithm

Since we have already known that indeterministic Turing machine is equivalent with deterministic ones, we will now only focus us to prove the equivalence between a deterministic Turing machine and a Markov algorithm.

Proposition 7. Any deterministic Turing Machine can be simulated by a Markov Algorithm.

Proof. Given a Turing Machine \(\text{TM} = \{Q, \Sigma, E, q_0, F\}\). We construct a Markov Algorithm \(\text{MA} = \{\Sigma', P, P_0\}\), where
\[ \Sigma' = \Sigma \cup \{ '0', '1', 'r' \}, \text{ with } '0', '1', 'r' \not\in \Sigma. \]

Thus we can code TM’s set of states \( Q \) with \( '0', '1', 'r' \). We note the coding of \( q_i \in Q \) as \( c_i \in \{ '0', '1' \}^* \), \( 0 \leq i \leq n \).

- The productions in \( P \) is composed of the following,
  - \( p_0 = \delta \rightarrow r c_0 \)
  - For each transition \( t_i \in E \), \( 1 \leq i \leq n \), if \( t_i = (q_j, a, q_k, b, \leftarrow) \), we construct one production
    \[ p_i = a c_j \rightarrow c_k b \]
    where as for \( t_i = (q_j, a, q_k, b, \rightarrow) \), we construct a series of productions as:
    \[ p_{id} = a c_j d \rightarrow b d c_k \quad \forall d \in \Sigma \]
    and finally for \( t_i = (q_j, a, q_k, b, \square) \), we construct
    \[ p_i = a c_j \rightarrow b c_k \]
  - \( p_{n+1} = r \rightarrow \delta. \)

- For each \( q_i \in F \), we add productions to \( P_0 \). We position them at back of the list so as to be applied only when the previous productions failed to match.
  \[ p'_i = q_i \rightarrow \epsilon \]

For an arbitrary input, MA codes the starting state and adds it to the head of the string, which also "represents" the position of read/writing head. Since TM is deterministic, for all pairs of \( (p, a) \), there exists at most one corresponding transition, thus at most one production in \( P - P_0 \).

If the input can be accepted by TM, then all transitions will be simulated by productions appearing in the \( P - P_0 \), followed by one of the productions in \( P_0 \), since original TM arrives at final state. Therefore the input is also accepted by MA, with exactly the same result.

If the input is rejected by TM, the same calculation is simulated by productions in \( P - P_0 \), in spite that the no production in \( P_0 \) will be applied. Thus the input was rejected by MA.

If TM iterates with the input, MA does so as well, because at all instant there is applicable productions in \( P - P_0 \).

Conclusion: this MA can simulate all calculations on corresponding TM.

\[ \square \]

**Corollaire 8** (Complexity of time and space).

\[ Time_{MA}(n) = O(Time_{TM}(n)) \]
\[ Space_{MA}(n) = O(Space_{TM}(n)) \]

**Proof.** According to the proof of the previous proposition, we simulate every transition in a single production, and with at most \( O(\log(|Q|)) \) more space consumption (constant). \[ \square \]
3.3 Simulation of Markov Algorithm by deterministic Turing machine

By instinct, we know that Turing machine is more agile than Markov algorithm thanks for the set of states. Thus this part seems more evident then the previous part, though it is a bit more verbose to illustrate.

Proposition 9. Any markov algorithm can be simulated by a deterministic Turing machine.

Proof. Suppose we are given a Markov algorithm $MA = (Σ, P, P_0)$.

We have already known how to build a finite and normalized automata that search a certain pattern $α$ in a string from left to right. We note this automata $S_α$ where $α$ is the pattern searched for, and $Ψ_{S_α}$. It accepts the input when it find the matching pattern, or rejects it if fail to find one.

Meanwhile, it is easy to build a Turing machine that overwrites the given word $α$ with $β$. We denote $W_β$ this machine and $Ψ_{W_β}$ its set of states.

We now construct a Turing machine $TM = \{Q, Σ', E, q_0, F\}$ such that

- $Q = \bigcup_{a→β∈P} (Ψ_{S_a} ∪ Ψ_{W_β}) \cup \{q_i\}_{0≤i≤n}$.
- $Σ' = Σ$.
- $q_0$ is the initial state of the automata $Ψ_{S_a}$.
- The components in E are transitions composed of the following:
  - Transitions $(q_i, #, ˇq_i, #, □)_{0≤i≤n}$, where $ˇq_i$ is the initial state of the automata $Ψ_{S_a}$.
  - Transitions $(q_i, a, q_i, a, □)_{a∈Σ^*}$, where $q_i$ is the rejected state of the automata $Ψ_{S_a}$.
  - Transitions $(q_i, a, q_i, a, □)_{a∈Σ}$, where $q_i$ is the accepted state of the automata $Ψ_{S_a}$, and $q_i$ is the initial state of the $Ψ_{W_β}$.
  - Transitions of the rewriting Turing machine $Ψ_{W_β}$.
  - Transitions $(q_i, a, q_0, a, □)_{a∈Σ}$, where $q_i$ is the final state of the writing automata.
  - Transitions in Writing automata $Ψ_{W_β}$ that substitute the string $α_i$ with $β_i$.
- $F$ includes all $q_i$, the the final state of $Ψ_{W_β}$, where $p_i ∈ P_0$. 
In this way, the Turing machine begin by scanning through the input trying to match \( \alpha_i \). If it fails to then return to the head and move on to search for the next pattern according to the order as in \( P \), until it finds a match, substitute it with \( \beta \). If the original MA accepts it, then Turing machine will arrive at the acceptable state as well, or else it returns to the initial state and start a new round.

3.4 Conclusion

With the equivalence between the calculability of a Turing machine and Markov algorithm, it then follows naturally that a function is Turing-calculable iff it is Markov-calculable.

References


