Enumerability, effectivity, decidability
Markov algorithms

András Máté

04.10.2024

Enumerability

In general, if we have a calculus to define some string class, we have an effective process to enumerate its members. We can enumerate the derivations in the calculus: first, the one-member derivations, then the two-member ones, etc.

Enumerability

In general, if we have a calculus to define some string class, we have an effective process to enumerate its members. We can enumerate the derivations in the calculus: first, the one-member derivations, then the two-member ones, etc.

There are infinitely many derivations in an usual calculus, but for a given number n you can always enumerate the n-member derivations in a sequence and then, you can produce one sequence from the sequence of sequences on the well-known way.

In general, if we have a calculus to define some string class, we have an effective process to enumerate its members. We can enumerate the derivations in the calculus: first, the one-member derivations, then the two-member ones, etc.

There are infinitely many derivations in an usual calculus, but for a given number n you can always enumerate the n-member derivations in a sequence and then, you can produce one sequence from the sequence of sequences on the well-known way.

The enumeration of derivations produces an enumeration of the derivable strings too. This informal consideration shows that inductively defined classes are effectively enumerable, i. e., we have a procedure that enumerates all of its members. What about the conversion of this claim? Is every effectively enumerable class inductively definable? We can have no answer yet.

Enumerability and decidability

If we had a calculus for the non-autonomous numerals we had an enumeration of the non-autonomous numerals, too. In that case we could decide about any given numeral n whether it is an autonomous numeral or not.

Enumerability and decidability

If we had a calculus for the non-autonomous numerals we had an enumeration of the non-autonomous numerals, too. In that case we could decide about any given numeral n whether it is an autonomous numeral or not.

Imagine that a printing machine prints the autonomous numbers in the order of enumeration and another one the non-autonomous numbers. After a finite time, n will occur as an output of either the first or the second machine and therefore we have a *decision procedure* for the membership of the class.

If we had a calculus for the non-autonomous numerals we had an enumeration of the non-autonomous numerals, too. In that case we could decide about any given numeral n whether it is an autonomous numeral or not.

Imagine that a printing machine prints the autonomous numbers in the order of enumeration and another one the non-autonomous numbers. After a finite time, n will occur as an output of either the first or the second machine and therefore we have a *decision procedure* for the membership of the class.

The generalization and the converse of the claim is obvious: we have an enumeration procedure both for a string class $\mathcal{B}$ over an alphabet $\mathcal{A}$ and its complement $\mathcal{A}^\circ - \mathcal{B}$ if and only if we have a decision procedure for $\mathcal{B}$.

Enumerability and decidability

If we had a calculus for the non-autonomous numerals we had an enumeration of the non-autonomous numerals, too. In that case we could decide about any given numeral n whether it is an autonomous numeral or not.

Imagine that a printing machine prints the autonomous numbers in the order of enumeration and another one the non-autonomous numbers. After a finite time, n will occur as an output of either the first or the second machine and therefore we have a *decision procedure* for the membership of the class.

The generalization and the converse of the claim is obvious: we have an enumeration procedure both for a string class $\mathcal{B}$ over an alphabet $\mathcal{A}$ and its complement $\mathcal{A}^{\circ} - \mathcal{B}$ if and only if we have a decision procedure for $\mathcal{B}$.

How to make precise and formally defined the notions used above: 'procedure', 'effective enumeration'? This is our next task.

The open question

The open question

We know that the string class $\mathcal{A}_0^\circ - \boldsymbol{Aut}$ is not inductively definable. Does it mean that it is not effectively enumerable, either?

The open question

We know that the string class $\mathcal{A}_0^\circ - \boldsymbol{Aut}$ is not inductively definable. Does it mean that it is not effectively enumerable, either?

Generalization: If a string class is not inductively definable, dores it imply that the class is not effectively enumerable, either?

The open question

We know that the string class $\mathcal{A}_0^\circ - \boldsymbol{Aut}$ is not inductively definable. Does it mean that it is not effectively enumerable, either?

Generalization: If a string class is not inductively definable, dores it imply that the class is not effectively enumerable, either?

Contrapositive form of the above (generalized) question: Is it true that an effectively enumerable class is always inductively definable?

The open question

We know that the string class $\mathcal{A}_0^\circ - \boldsymbol{Aut}$ is not inductively definable. Does it mean that it is not effectively enumerable, either?

Generalization: If a string class is not inductively definable, dores it imply that the class is not effectively enumerable, either?

Contrapositive form of the above (generalized) question: Is it true that an effectively enumerable class is always inductively definable?

If the answer is 'yes', then the class of autonomous numerals is not decidable (although it is enumerable).

We know that the string class $\mathcal{A}_0^{\circ} - \boldsymbol{Aut}$ is not inductively definable. Does it mean that it is not effectively enumerable, either?

Generalization: If a string class is not inductively definable, dores it imply that the class is not effectively enumerable, either?

Contrapositive form of the above (generalized) question: Is it true that an effectively enumerable class is always inductively definable?

If the answer is 'yes', then the class of autonomous numerals is not decidable (although it is enumerable).

But to establish such an answer, we need a (formal) notion of effective procedure.

Procedures, algorithms

A *procedure* or *algorithm* is a set of commands that you should perform in a prescribed sequence in order to solve a task of some type (class of tasks). Some well-known sorts of procedures:

A *procedure* or *algorithm* is a set of commands that you should perform in a prescribed sequence in order to solve a task of some type (class of tasks). Some well-known sorts of procedures:

Operations. Example: multiplication of numerals. Given any pair of numerals, produce a numeral which denotes the product of the two numbers.

Procedures, algorithms

A *procedure* or *algorithm* is a set of commands that you should perform in a prescribed sequence in order to solve a task of some type (class of tasks). Some well-known sorts of procedures:

Operations. Example: multiplication of numerals. Given any pair of numerals, produce a numeral which denotes the product of the two numbers.

Decision procedures. Example: Given a string from $\mathcal{A}^{\circ}_{Language(FOL)}$, decide whether it is a formula of $Language(FOL)$ or not.

Procedures, algorithms

A *procedure* or *algorithm* is a set of commands that you should perform in a prescribed sequence in order to solve a task of some type (class of tasks). Some well-known sorts of procedures:

Operations. Example: multiplication of numerals. Given any pair of numerals, produce a numeral which denotes the product of the two numbers.

Decision procedures. Example: Given a string from $\mathcal{A}^{\circ}_{Language(FOL)}$, decide whether it is a formula of $Language(FOL)$ or not.

Enumeration procedure for a given sequence (of strings). Example: from any string of the alphabet $\mathcal{A}_{cc}$, produce the next string in the lexicographic ordering.

Markov algorithms

Markov algorithms

Ways to formalize the notion of (finite, effectively performable) procedure: Turing machines, recursive functions, lambda calculus etc. We will use Markov algorithms.

Markov algorithms

Ways to formalize the notion of (finite, effectively performable) procedure: Turing machines, recursive functions, lambda calculus etc. We will use Markov algorithms.

A calculus tells us what we are *allowed* to do, an algorithm prescribes what we *must* do.

Markov algorithms

Ways to formalize the notion of (finite, effectively performable) procedure: Turing machines, recursive functions, lambda calculus etc. We will use Markov algorithms.

A calculus tells us what we are *allowed* to do, an algorithm prescribes what we *must* do.

Markov algorithms transform strings of a given alphabet into other strings. Every step of the algorithm is a substitution of a string by another string, prescribed by the commands of the algorithm and their order.

Ways to formalize the notion of (finite, effectively performable) procedure: Turing machines, recursive functions, lambda calculus etc. We will use Markov algorithms.

A calculus tells us what we are *allowed* to do, an algorithm prescribes what we *must* do.

Markov algorithms transform strings of a given alphabet into other strings. Every step of the algorithm is a substitution of a string by another string, prescribed by the commands of the algorithm and their order.

Markov algorithm (or normal algorithm) over an alphabet $\mathcal{A}$ (not containing the characters '$\rightarrow$' and '$\cdot$') is a finite, nonempty sequence N of $\mathcal{A}$-commands.

Ways to formalize the notion of (finite, effectively performable) procedure: Turing machines, recursive functions, lambda calculus etc. We will use Markov algorithms.

A calculus tells us what we are *allowed* to do, an algorithm prescribes what we *must* do.

Markov algorithms transform strings of a given alphabet into other strings. Every step of the algorithm is a substitution of a string by another string, prescribed by the commands of the algorithm and their order.

Markov algorithm (or normal algorithm) over an alphabet $\mathcal{A}$ (not containing the characters '$\rightarrow$' and '$\cdot$') is a finite, nonempty sequence N of $\mathcal{A}$-commands.

An $\mathcal{A}$-command is a string of the form $\ulcorner a \rightarrow b \urcorner$ or $\ulcorner a \rightarrow \cdot b \urcorner$ where a (the input of the command) and b (the output) are $\mathcal{A}$-strings. Commands of the latter form are called stop commands.

Application of a command resp. algorithm to a word

Application of a command resp. algorithm to a word

The command $C = a \to b$ resp. $a \to \cdot b$ is applicable to a string f if its input a occurs as a sub-string in f, i.e. $f = u^\frown a^\frown v$, where u and v can be any string over $\mathcal{A}$.

Application of a command resp. algorithm to a word

The command $C = a \to b$ resp. $a \to \cdot b$ is applicable to a string f if its input a occurs as a sub-string in f, i.e. $f = u^\frown a^\frown v$, where u and v can be any string over $\mathcal{A}$.

The application of C to f is the substitution of the *first* occurrence of a in f by b. The result: $C(f)$.

Application of a command resp. algorithm to a word

The command $C = a \to b$ resp. $a \to \cdot b$ is applicable to a string f if its input a occurs as a sub-string in f, i.e. $f = u^\frown a^\frown v$, where u and v can be any string over $\mathcal{A}$.

The application of C to f is the substitution of the *first* occurrence of a in f by b. The result: $C(f)$.

Steps of the application of an algorithm N to a string f_0 (informally):

Application of a command resp. algorithm to a word

The command $C = a \to b$ resp. $a \to \cdot b$ is applicable to a string f if its input a occurs as a sub-string in f, i.e. $f = u^\frown a^\frown v$, where u and v can be any string over $\mathcal{A}$.

The application of C to f is the substitution of the *first* occurrence of a in f by b. The result: $C(f)$.

Steps of the application of an algorithm N to a string f_0 (informally):

1. If no command in N is applicable to f_0, then f_0 blocks N, in symbols, $N(f) = \sharp$ ($\sharp \notin \mathcal{A}$).

Application of a command resp. algorithm to a word

The command $C = a \to b$ resp. $a \to \cdot b$ is applicable to a string f if its input a occurs as a sub-string in f, i.e. $f = u^\frown a^\frown v$, where u and v can be any string over $\mathcal{A}$.

The application of C to f is the substitution of the *first* occurrence of a in f by b. The result: $C(f)$.

Steps of the application of an algorithm N to a string f_0 (informally):

1. If no command in N is applicable to f_0, then f_0 blocks N, in symbols, $N(f) = \sharp$ ($\sharp \notin \mathcal{A}$).
2. Otherwise, apply the *first* applicable command C_0 to f_0. The result is $f_1 = C_0(f_0)$.

Application of a command resp. algorithm to a word

The command $C = a \to b$ resp. $a \to \cdot b$ is applicable to a string f if its input a occurs as a sub-string in f, i.e. $f = u^\frown a^\frown v$, where u and v can be any string over $\mathcal{A}$.

The application of C to f is the substitution of the *first* occurrence of a in f by b. The result: $C(f)$.

Steps of the application of an algorithm N to a string f_0 (informally):

1. If no command in N is applicable to f_0, then f_0 blocks N, in symbols, $N(f) = \sharp$ ($\sharp \notin \mathcal{A}$).
2. Otherwise, apply the *first* applicable command C_0 to f_0. The result is $f_1 = C_0(f_0)$.
3. If C_0 was a stop command, then N applies to f_0 and transforms it to f_1. In symbols, $N(f_0) = f_1$.

Application of a command resp. algorithm to a word

The command $C = a \to b$ resp. $a \to \cdot b$ is applicable to a string f if its input a occurs as a sub-string in f, i.e. $f = u^\frown a^\frown v$, where u and v can be any string over $\mathcal{A}$.

The application of C to f is the substitution of the *first* occurrence of a in f by b. The result: $C(f)$.

Steps of the application of an algorithm N to a string f_0 (informally):

1. If no command in N is applicable to f_0, then f_0 blocks N, in symbols, $N(f) = \sharp$ ($\sharp \notin \mathcal{A}$).
2. Otherwise, apply the *first* applicable command C_0 to f_0. The result is $f_1 = C_0(f_0)$.
3. If C_0 was a stop command, then N applies to f_0 and transforms it to f_1. In symbols, $N(f_0) = f_1$.
4. If it was not, then N leads f_0 to f_1 (in symbols, $N(f_0/f_1)$) and the algorithm continues with step 1, but f_1 takes the place of f_0. If we arrive to a stop command, then the original string, f_0 is transformed into the last result).

Possible outcomes of the application of an algorithm

Possible outcomes of the application of an algorithm

If we try to apply an algorithm N to a string f, there are three possibilities:

Possible outcomes of the application of an algorithm

If we try to apply an algorithm N to a string f, there are three possibilities:

1. After performing finitely many times the steps above, we arrive to a situation that no command in N applies to our last result. In this case, N does not apply to f or f blocks N, $N(f) = \sharp$.

Possible outcomes of the application of an algorithm

If we try to apply an algorithm N to a string f, there are three possibilities:

1. After performing finitely many times the steps above, we arrive to a situation that no command in N applies to our last result. In this case, N does not apply to f or f blocks N, $N(f) = \sharp$.
2. After finitely many steps, we arrive to a stop command. If the result of the application of this command was g, then N applies to f and transforms it to g, $N(f) = g$.

Possible outcomes of the application of an algorithm

If we try to apply an algorithm N to a string f, there are three possibilities:

1. After performing finitely many times the steps above, we arrive to a situation that no command in N applies to our last result. In this case, N does not apply to f or f blocks N, $N(f) = \sharp$.
2. After finitely many steps, we arrive to a stop command. If the result of the application of this command was g, then N applies to f and transforms it to g, $N(f) = g$.
3. We never arrive after finitely many steps to a stop command, nor to a blocking situation. In this case, N runs infinitely on f.

Possible outcomes of the application of an algorithm

If we try to apply an algorithm N to a string f, there are three possibilities:

1. After performing finitely many times the steps above, we arrive to a situation that no command in N applies to our last result. In this case, N does not apply to f or f blocks N, $N(f) = \sharp$.
2. After finitely many steps, we arrive to a stop command. If the result of the application of this command was g, then N applies to f and transforms it to g, $N(f) = g$.
3. We never arrive after finitely many steps to a stop command, nor to a blocking situation. In this case, N runs infinitely on f.

The first case can be avoided by inserting the command $\varnothing \to \cdot\varnothing$ to the end of the algorithm. It is applicable to any string and does nothing but stops the algorithm.

Formal definitions of the above notions

Formal definitions of the above notions

Simultaneous inductive definition of the relations $N(f) = \sharp$ (f blocks N), $N(f) = g$ (N transforms f into g) and $N(f/g)$ (N leads f to g). (N is an algorithm over $\mathcal{A}$, f and g are $\mathcal{A}$-strings and $\sharp \notin \mathcal{A}$.)

Simultaneous inductive definition of the relations $N(f) = \sharp$ (f blocks N), $N(f) = g$ (N transforms f into g) and $N(f/g)$ (N leads f to g). (N is an algorithm over $\mathcal{A}$, f and g are $\mathcal{A}$-strings and $\sharp \notin \mathcal{A}$.)

Formal definitions of the above notions

Simultaneous inductive definition of the relations $N(f) = \sharp$ (f blocks N), $N(f) = g$ (N transforms f into g) and $N(f/g)$ (N leads f to g). (N is an algorithm over $\mathcal{A}$, f and g are $\mathcal{A}$-strings and $\sharp \notin \mathcal{A}$.)

1. If no command in N is applicable to f, then $N(f) = \sharp$.

Formal definitions of the above notions

Simultaneous inductive definition of the relations $N(f) = \sharp$ (f blocks N), $N(f) = g$ (N transforms f into g) and $N(f/g)$ (N leads f to g). (N is an algorithm over $\mathcal{A}$, f and g are $\mathcal{A}$-strings and $\sharp \notin \mathcal{A}$.)

1. If no command in N is applicable to f, then $N(f) = \sharp$.
2. If C is the first command in N that is applicable to f, $C(f) = g$, then
   - a. if C is a stop command, then $N(f) = g$;
   - b. if C is not a stop command, then $N(f/g)$.

Formal definitions of the above notions

Simultaneous inductive definition of the relations $N(f) = \sharp$ (f blocks N), $N(f) = g$ (N transforms f into g) and $N(f/g)$ (N leads f to g). (N is an algorithm over $\mathcal{A}$, f and g are $\mathcal{A}$-strings and $\sharp \notin \mathcal{A}$.)

i. If no command in N is applicable to f, then $N(f) = \sharp$.
ii. If C is the first command in N that is applicable to f, $C(f) = g$, then
    a. if C is a stop command, then $N(f) = g$;
    b. if C is not a stop command, then $N(f/g)$.
iii. If $N(f/g)$ and $N(g/h)$, then $N(f/h)$.

Formal definitions of the above notions

Simultaneous inductive definition of the relations $N(f) = \sharp$ (f blocks N), $N(f) = g$ (N transforms f into g) and $N(f/g)$ (N leads f to g). (N is an algorithm over $\mathcal{A}$, f and g are $\mathcal{A}$-strings and $\sharp \notin \mathcal{A}$.)

1. If no command in N is applicable to f, then $N(f) = \sharp$.
2. If C is the first command in N that is applicable to f, $C(f) = g$, then
   1. if C is a stop command, then $N(f) = g$;
   2. if C is not a stop command, then $N(f/g)$.
3. If $N(f/g)$ and $N(g/h)$, then $N(f/h)$.
4. If $N(f/g)$ and $N(g) = h$, then $N(f) = h$.

Formal definitions of the above notions

Simultaneous inductive definition of the relations $N(f) = \sharp$ (f blocks N), $N(f) = g$ (N transforms f into g) and $N(f/g)$ (N leads f to g). (N is an algorithm over $\mathcal{A}$, f and g are $\mathcal{A}$-strings and $\sharp \notin \mathcal{A}$.)

i. If no command in N is applicable to f, then $N(f) = \sharp$.
ii. If C is the first command in N that is applicable to f, $C(f) = g$, then
    a. if C is a stop command, then $N(f) = g$;
    b. if C is not a stop command, then $N(f/g)$.
iii. If $N(f/g)$ and $N(g/h)$, then $N(f/h)$.
iv. If $N(f/g)$ and $N(g) = h$, then $N(f) = h$.
v. If $N(f/g)$ and $N(g) = \sharp$, then $N(f) = \sharp$.

Examples

Erase a letter. Be $a\epsilon\mathcal{A}$. Let us erase every occurrence of a from any string.

Examples

Erase a letter. Be $a\epsilon\mathcal{A}$. Let us erase every occurrence of a from any string.

1. $a \to \varnothing$
2. $\varnothing \to \cdot\varnothing$

Examples

Erase a letter. Be $a \epsilon \mathcal{A}$. Let us erase every occurrence of a from any string.

$$1. \quad a \rightarrow \varnothing$$

$$2. \quad \varnothing \rightarrow \cdot\varnothing$$

Erase every letter.

Examples

Erase a letter. Be $a \epsilon \mathcal{A}$. Let us erase every occurrence of a from any string.

1. $a \to \varnothing$
2. $\varnothing \to \cdot\varnothing$

Erase every letter.

1. $x \to \varnothing \qquad x \in \mathcal{A}$
2. $\varnothing \to \cdot\varnothing$

Examples

Erase a letter. Be $a\epsilon\mathcal{A}$. Let us erase every occurrence of a from any string.

$$\begin{array}{ll} 1. & a \to \varnothing \\ 2. & \varnothing \to \cdot\varnothing \end{array}$$

Erase every letter.

$$\begin{array}{lll} 1. & x \to \varnothing & x \in \mathcal{A} \\ 2. & \varnothing \to \cdot\varnothing & \end{array}$$

The letter x is a metalanguage variable for letters and the first command is an usual and obvious abbreviation of n commands, if $\mathcal{A}$ has n members.

A mirroring algorithm

A mirroring algorithm

We can use auxiliary letters in algorithms as well as in calculi. It means only that to solve algorithmically some problem concerning the $\mathcal{A}$-strings, we write an algorithm over some $\mathcal{B} \supset \mathcal{A}$ and *we regard* the members of $\mathcal{B} - \mathcal{A}$ auxiliary letters.

A mirroring algorithm

We can use auxiliary letters in algorithms as well as in calculi. It means only that to solve algorithmically some problem concerning the $\mathcal{A}$-strings, we write an algorithm over some $\mathcal{B} \supset \mathcal{A}$ and *we regard* the members of $\mathcal{B} - \mathcal{A}$ auxiliary letters.

The following algorithm brings any $\mathcal{A}$-string $a_0 a_1 \ldots a_n$ into the string $a_0 a_1 \ldots a_n \mid a_n a_{n-1} \ldots a_0$ ($\mid \notin \mathcal{A}$, and the algorithm uses the auxiliary letters A, C, too.).

We can use auxiliary letters in algorithms as well as in calculi. It means only that to solve algorithmically some problem concerning the $\mathcal{A}$-strings, we write an algorithm over some $\mathcal{B} \supset \mathcal{A}$ and *we regard* the members of $\mathcal{B} - \mathcal{A}$ auxiliary letters.

The following algorithm brings any $\mathcal{A}$-string $a_0 a_1 \ldots a_n$ into the string $a_0 a_1 \ldots a_n \mid a_n a_{n-1} \ldots a_0$ ($\mid \notin \mathcal{A}$, and the algorithm uses the auxiliary letters A, C, too.).

1.	$Cxy \to yCx$	$x \in \mathcal{A},\ y \in \mathcal{A} \cup \{\mid\}$
2.	$Cx \to x$	$x \in \mathcal{A}$
3.	$xA \to AxCx$	$x \in \mathcal{A}$
4.	$A \to .\varnothing$	
5.	$\mid x \to x \mid$	$x \in \mathcal{A}$
6.	$x \mid \to xA \mid$	$x \in \mathcal{A}$
7.	$\varnothing \to \mid$	

Homework

Homework

Write an algorithm that decides identity of strings of some alphabet $\mathcal{A}$ in the following sense: Let V and W arbitrary $\mathcal{A}$-strings. Your algorithm should transform the string $V \mid W$ into Y if they are the same string, and in N if they are different. (Y, $\mid$ and N are auxiliary letters.)