Some Philosophical Remarks on the Current Definitions of Algorithms

Philippos Papayannopoulos
IHPST (UMR 8590),
Université Paris 1 Panthéon-Sorbonne and CNRS
ANR - The Geometry of Algorithms

13th Panhellenic Logic Symposium,
Volos, July 6-10 2022

Overview

Work from a broad project looking into various models of computation over countable and uncountable domains as well as into definitional approaches to algorithms per se:

- TMs, Russian school of constructive approaches (Kolmogorov, Markov), Shönhage machines, Computable analysis (Weihrauch school – TTE), BSS model (numerical analysis).
- Gurevich, Moschovakis, contemporary approach in France (based on Girard's geometry of interaction.)

Goal: understand the underlying notion of 'algorithm'.

Brief conclusion: we use "algorithms" in more than one way.

Overview

Work from a broad project looking into various models of computation over countable and uncountable domains as well as into definitional approaches to algorithms per se:

- TMs, Russian school of constructive approaches (Kolmogorov, Markov), Shönhage machines, Computable analysis (Weihrauch school – TTE), BSS model (numerical analysis).
- Gurevich, Moschovakis, contemporary approach in France (based on Girard's geometry of interaction.)

Goal: understand the underlying notion of 'algorithm'.

Brief conclusion: we use "algorithms" in more than one way.

In this talk: a higher level (informal) approach to what is at stake.

Only sequential, deterministic algorithms ("the idea of the '30s").

Future directions: geometric constructions, parallel algorithms (and perhaps analog/quantum algorithms).

Algorithms: the informal idea

A classical algorithm:

- is expressed as a set of instructions of finite size.
- has a set (perhaps empty) of inputs and a (set of) output(s).
- carried out in a discrete stepwise fashion (proceeds in discrete time).
- is deterministic (no resort to random methods).
- each step of an algorithm must precisely and unambiguously be specified to such a sufficient detail as no ingenuity whatsoever may be required by the computing agent.
- generally terminates after a finite number of steps.

Algorithms and models of computation

Such methods have been in use since ancient times. ▸ Examples

But there was no strong connection with the effectively computable functions, in the way that became prominent after the 1930s.

–Algorithms were primarily means for obtaining solutions to problems (equations, primality tests, etc.).

Algorithms and models of computation

Such methods have been in use since ancient times. ▸ Examples

But there was no strong connection with the effectively computable functions, in the way that became prominent after the 1930s.

–Algorithms were primarily means for obtaining solutions to problems (equations, primality tests, etc.).

–OTOH, the early models of computation arose from foundational concerns:

- The first recursive definitions appeared in Dedekind's and Skolem's works on the foundations of arithmetic, and recursion theory was developed through work in Hilbert's program and Gödel's Inc. theorems to its final form in the theories of general recursion (Gödel/Herbrand) and μ-recursion (Kleene).

Algorithms and models of computation

Such methods have been in use since ancient times. ‣ Examples

But there was no strong connection with the effectively computable functions, in the way that became prominent after the 1930s.

–Algorithms were primarily means for obtaining solutions to problems (equations, primality tests, etc.).

–OTOH, the early models of computation arose from foundational concerns:

- The first recursive definitions appeared in Dedekind's and Skolem's works on the foundations of arithmetic, and recursion theory was developed through work in Hilbert's program and Gödel's Inc. theorems to its final form in the theories of general recursion (Gödel/Herbrand) and μ-recursion (Kleene).
- λ-calculus was a sub-system of a broader system by Church (purporting to avoid the incompleteness results but in the end discovered inconsistent), whose initial sole purpose was to to distinguish the *function* of x (the assignment/rule) from its *values*.

Algorithms and models of computation

The connection between the two notions became strengthened by facts like the following:

Algorithms and models of computation

The connection between the two notions became strengthened by facts like the following:

- Turing's (1936) focused on the *process* of computing itself.
- The assertion that a specific function "can be computed by carrying out an effective process" became synonymous to the assertion that the function can be "computed by following an algorithm" (e.g., Church 1936).
- Although Turing nowhere mentions the term "algorithm" in (1936), the paper serves, among others, the purpose of providing an answer to Hilbert's request for an algorithmic method (Entscheidungsproblem).

Algorithms and computation: A marriage made in heaven?

After these developments, talk about ‘algorithms’ became almost inseparable from talk about ‘computation’.

Markov’s *Theory of Algorithms* was very much a continuation of these developments, but “algorithms” were now explicitly the subject matter.

Kolmogorov’s (and Uspensky’s) definition of algorithms (1958) justified the appropriateness of their approach against the backdrop of the CTT (in its later form, which concerns partial recurs. functions).

Algorithms and computation: A marriage made in heaven?

After these developments, talk about ‘algorithms’ became almost inseparable from talk about ‘computation’.

Markov’s *Theory of Algorithms* was very much a continuation of these developments, but “algorithms” were now explicitly the subject matter.

Kolmogorov’s (and Uspensky’s) definition of algorithms (1958) justified the appropriateness of their approach against the backdrop of the CTT (in its later form, which concerns partial recurs. functions).

Assertions like the following:

> “there is no algorithm that decides the validity for any first-order sentence”
>
> “‘if $\mathbf{P} \neq \mathbf{NP}$, there is no algorithm that solves the Boolean satisfiability problem in polynomial time.”

are staples in computability and complexity theory.

Algorithms and computation: A marriage made in heaven?

Furthermore, the view that TMs and the CTT explicate algorithms became part of the folklore of logic and CS textbooks:

> "*Partial recursive functions are the natural formalization of algorithms*" (Odifreddi 1999, p.3)
>
> The CTT "*imposes a precise, mathematical upper bound to the vague, intuitive but basic notion of algorithm that underlies the concept of effective computability*" (ibid., p.102)

Algorithms and computation: A marriage made in heaven?

Furthermore, the view that TMs and the CTT explicate algorithms became part of the folklore of logic and CS textbooks:

> "*Partial recursive functions are the natural formalization of algorithms*" (Odifreddi 1999, p.3)
>
> The CTT "*imposes a precise, mathematical upper bound to the vague, intuitive but basic notion of algorithm that underlies the concept of effective computability*" (ibid., p.102)

Also:

> *We therefore propose to adopt the Turing machine that halts on all inputs as the precise formal notion corresponding to the intuitive notion of an "algorithm." Nothing will be considered an algorithm if it cannot be rendered as a Turing machine that is guaranteed to halt on all inputs, and all such machines will be rightfully called algorithms.* (Lewis and Papadimitriou 1998, p.246)

The symbolic conception of algorithms

Implicit in the view that identifies algorithms with (instances of) machine models is a conception of algorithms as symbolic procedures.

Markov (1954 [1962]) considers only "algorithms in given alphabets" operating with concrete words (p.58-59).

"*Without fixing a standard way of writing numbers, to speak of the algorithm computing* [the value of a function from its input] *would make no sense.*" (Kolmogorov and Uspenskii 1958 [1963] fn.2)

The symbolic conception of algorithms

Implicit in the view that identifies algorithms with (instances of) machine models is a conception of algorithms as symbolic procedures.

Markov (1954 [1962]) considers only "algorithms in given alphabets" operating with concrete words (p.58-59).

"*Without fixing a standard way of writing numbers, to speak of the algorithm computing* [the value of a function from its input] *would make no sense.*" (Kolmogorov and Uspenskii 1958 [1963] fn.2)

"*Mechanical devices engaged in computation and humans following algorithms*[..] *do not encounter numbers themselves, but rather physical objects such as ink marks on paper. Since strings are the relevant abstract forms of these physical objects, algorithms should be understood as procedures for the manipulation of strings, not numbers.*" (Shapiro 1982)

The symbolic conception of algorithms

On the symbolic view, algorithms are tightly interlocked with specific notations (alphabets, languages, etc.).

The symbolic conception of algorithms

On the symbolic view, algorithms are tightly interlocked with specific notations (alphabets, languages, etc.).

This naturally bears on their identity conditions:

- Altering the symbols changes the algorithm too, since it affects the exact sequence of steps

The symbolic conception of algorithms

On the symbolic view, algorithms are tightly interlocked with specific notations (alphabets, languages, etc.).

This naturally bears on their identity conditions:

- Altering the symbols changes the algorithm too, since it affects the exact sequence of steps

This is very clearly seen in computation over the reals, where computability depends on the representations (base-`b`, nested intervals, Cauchy sequences, etc.).

Symbolic algorithms and mathematical practice

Many algorithms in ordinary math practice are not effective (e.g., Bisection algorithm, Newton's algorithm etc.)

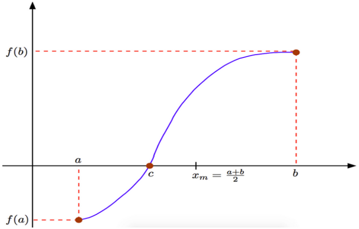
Symbolic algorithms and mathematical practice

Many algorithms in ordinary math practice are not effective (e.g., Bisection algorithm, Newton's algorithm etc.)

Example: Bisection for finding a root $x_0 \in [a, b]$ of a continuous function $f(x)$, when $f(a)f(b) < 0$. We start with an interval that is known to contain the root, x_0, and we approach it by iterated bisections of that interval.

BISECT (f, a_1, b_1)

1. Compute $c_i = \frac{a_i + b_i}{2}$ and go to 2;
2. If $f(c_i) = 0$, go to 5, else go to 3;
3. If $f(a_i)f(c_i) < 0$, set $b_{i+1} = c_i$ and $a_{i+1} = a_i$. Else, set $a_{i+1} = c_i$ and $b_{i+1} = b_i$. Go to 4;
4. Set $i = i + 1$ and go to 1.
5. Stop and return c_i



These algorithms do not have precisely defined sequences of steps or specific alphabets. But we recognize to them some natural structure and identity.

Algorithms

One might be inclined to see the issue as one concerning easily filled details (e.g., by invoking a distinction between higher/lower-level algorithms).

.

Algorithms

One might be inclined to see the issue as one concerning easily filled details (e.g., by invoking a distinction between higher/lower-level algorithms).

> *It must be clear that the ways and means which a mathematician is used to of describing a general procedure are in general too vague to come up really to the required standard of exactness. This applies for instance to the usual description of methods for the solution of a linear equation system. ... It is however clear to every mathematician that ... the* [additional necessary] *instruction*[s] *can be supplemented to make a complete instruction which does not leave anything open.* (Hermes 1969, p.2)

.

Algorithms

One might be inclined to see the issue as one concerning easily filled details (e.g., by invoking a distinction between higher/lower-level algorithms).

> *It must be clear that the ways and means which a mathematician is used to of describing a general procedure are in general too vague to come up really to the required standard of exactness. This applies for instance to the usual description of methods for the solution of a linear equation system. ... It is however clear to every mathematician that ... the* [additional necessary] *instruction*[s] *can be supplemented to make a complete instruction which does not leave anything open.* (Hermes 1969, p.2)

Thus, perhaps one could say that abstract algorithms (like the above) are higher-level algorithms, which do not describe exact computational procedures but some sort of patterns of actions or "algorithmic schemas".

Algorithms

One might be inclined to see the issue as one concerning easily filled details (e.g., by invoking a distinction between higher/lower-level algorithms).

> *It must be clear that the ways and means which a mathematician is used to of describing a general procedure are in general too vague to come up really to the required standard of exactness. This applies for instance to the usual description of methods for the solution of a linear equation system. ... It is however clear to every mathematician that ... the* [additional necessary] *instruction*[s] *can be supplemented to make a complete instruction which does not leave anything open.* (Hermes 1969, p.2)

Thus, perhaps one could say that abstract algorithms (like the above) are higher-level algorithms, which do not describe exact computational procedures but some sort of patterns of actions or "algorithmic schemas".
But this leads nowhere really. A great many of such algorithms rely essentially on some non-effective operations (mainly comparisons between reals, assuming a CTT).

The Abstract conception of algorithms

It seems a stretch to conceptualize algorithms like the above as specifying purely symbolic manipulations.

The Abstract conception of algorithms

It seems a stretch to conceptualize algorithms like the above as specifying purely symbolic manipulations.

Similarly, it seems unnatural to formalize them in a way that their identity conditions are highly sensitive to the chosen representations.
Rather, algorithms like the above seem to possess an identity and natural structure of their own.

The Abstract conception of algorithms

It seems a stretch to conceptualize algorithms like the above as specifying purely symbolic manipulations.

Similarly, it seems unnatural to formalize them in a way that their identity conditions are highly sensitive to the chosen representations.
Rather, algorithms like the above seem to possess an identity and natural structure of their own.

The same seems true of also some algorithms over countable domains (e.g., divide-and-conquer, such as Mergesort). Though these are effective.

The Abstract conception of algorithms

It seems a stretch to conceptualize algorithms like the above as specifying purely symbolic manipulations.

Similarly, it seems unnatural to formalize them in a way that their identity conditions are highly sensitive to the chosen representations.
Rather, algorithms like the above seem to possess an identity and natural structure of their own.

The same seems true of also some algorithms over countable domains (e.g., divide-and-conquer, such as Mergesort). Though these are effective.

One way to conceptualize the abstractness (for algorithms over countable domains) is to understand algorithms as entities hovering over (equivalence) classes, where the equivalence relations might be simulability between machine models. (This faces various problems (see Dean 2016) but it is a natural starting point.)

The Abstract conception of algorithms

It seems a stretch to conceptualize algorithms like the above as specifying purely symbolic manipulations.

Similarly, it seems unnatural to formalize them in a way that their identity conditions are highly sensitive to the chosen representations.
Rather, algorithms like the above seem to possess an identity and natural structure of their own.

The same seems true of also some algorithms over countable domains (e.g., divide-and-conquer, such as Mergesort). Though these are effective.

One way to conceptualize the abstractness (for algorithms over countable domains) is to understand algorithms as entities hovering over (equivalence) classes, where the equivalence relations might be simulability between machine models. (This faces various problems (see Dean 2016) but it is a natural starting point.)

With numerical analysis the situation is worse, because there isn't an abundance of machine models similar to computation over countable domains (except BSS and TTE, which are incompatible), and so no equivalence classes either.

A fundamental tension

So are algorithms processes of symbolic manipulations (thus, dependent on the particular symbols used) or do they have some identity and natural structure independent of particular symbolic systems?

A fundamental tension

So are algorithms processes of symbolic manipulations (thus, dependent on the particular symbols used) or do they have some identity and natural structure independent of particular symbolic systems?

> "*Notice, however, that the standard algorithm for multiplication ... works on numbers written in decimal notation. The same algorithm would either give the wrong results, or senseless results, for numbers written in unary, binary, hexadecimal or Roman notation.*" (Shapiro 2017)

A fundamental tension

So are algorithms processes of symbolic manipulations (thus, dependent on the particular symbols used) or do they have some identity and natural structure independent of particular symbolic systems?

> "*Notice, however, that the standard algorithm for multiplication ... works on numbers written in decimal notation. The same algorithm would either give the wrong results, or senseless results, for numbers written in unary, binary, hexadecimal or Roman notation.*" (Shapiro 2017)

> "*Sturm's algorithm is usually defined over the class of all polynomials with arbitrary real coefficients, and there is nothing in its description or analysis of its implementation that requires those coefficients be integers or rationals. If we want to implement the algorithm to some actual computer or abstract Turing machine, then, of course, we need to approximate the real coefficients by means of rationals, and choose certain symbolic representation. However, there are various ways to go about these choices and none of these is essentially included in Sturm's algorithm*" (Moschovakis 1997)

A fundamental tension

So are algorithms processes of symbolic manipulations (thus, dependent on the particular symbols used) or do they have some identity and natural structure independent of particular symbolic systems?

> "*Notice, however, that the standard algorithm for multiplication ... works on numbers written in decimal notation. The same algorithm would either give the wrong results, or senseless results, for numbers written in unary, binary, hexadecimal or Roman notation.*" (Shapiro 2017)

> "*Sturm's algorithm is usually defined over the class of all polynomials with arbitrary real coefficients, and there is nothing in its description or analysis of its implementation that requires those coefficients be integers or rationals. If we want to implement the algorithm to some actual computer or abstract Turing machine, then, of course, we need to approximate the real coefficients by means of rationals, and choose certain symbolic representation. However, there are various ways to go about these choices and none of these is essentially included in Sturm's algorithm*" (Moschovakis 1997)

The tension is fundamental and has to do with our pre-theoretical idea of what and algorithms is.

Another aspect of the same tension

Are algorithms syntactic or semantic objects?

Another aspect of the same tension

Are algorithms syntactic or semantic objects?

> "We are interested in the semantics-to-syntax analyses of algorithms like that of Turing. You study a species of algorithms. and you try to explicate it ... And you finish up with a syntactic artifact, like a particular kind of machines that execute all and only the algorithms ... in consideration." (Gurevich 2015, 188-9)

Another aspect of the same tension

Are algorithms syntactic or semantic objects?

> "We are interested in the semantics-to-syntax analyses of algorithms like that of Turing. You study a species of algorithms. and you try to explicate it ... And you finish up with a syntactic artifact, like a particular kind of machines that execute all and only the algorithms ... in consideration." (Gurevich 2015, 188-9)

Contrast:

Another aspect of the same tension

Are algorithms syntactic or semantic objects?

> "We are interested in the semantics-to-syntax analyses of algorithms like that of Turing. You study a species of algorithms. and you try to explicate it ... And you finish up with a syntactic artifact, like a particular kind of machines that execute all and only the algorithms ... in consideration." (Gurevich 2015, 188-9)

Contrast:

> "[Recursive definitions] are the "more abstract" machines which model single-valued algorithms, and I have avoided the word "definition" in their name since it suggests syntactical objects, which algorithms are not" (Moschovakis 2001)

Another aspect of the same tension

Are algorithms syntactic or semantic objects?

> "We are interested in the semantics-to-syntax analyses of algorithms like that of Turing. You study a species of algorithms. and you try to explicate it ... And you finish up with a syntactic artifact, like a particular kind of machines that execute all and only the algorithms ... in consideration." (Gurevich 2015, 188-9)

Contrast:

> "[Recursive definitions] are the "more abstract" machines which model single-valued algorithms, and I have avoided the word "definition" in their name since it suggests syntactical objects, which algorithms are not" (Moschovakis 2001)

> "[T]here are many ways to assign an iterator ... to each system of recursive equations ... and there is no single, natural way to choose any one of them as "canonical". This problem ... makes it very unlikely that we can usefully identify algorithms with computational procedures, or iterators." (Moschovakis 1998, 4.3)

What is at stake?

Resolving the tension is crucial, for it bears on the imposed identity conditions within any formal definition/framework:

What is at stake?

Resolving the tension is crucial, for it bears on the imposed identity conditions within any formal definition/framework:

OTOH, we surely want to say that various specific processes instantiate the same algorithm even when their implementation details are changed (e.g., order of steps in Newton's method or Mergesort, particular alphabet/programming language, number of tapes/states in TMs, etc.).

What is at stake?

Resolving the tension is crucial, for it bears on the imposed identity conditions within any formal definition/framework:

OTOH, we surely want to say that various specific processes instantiate the same algorithm even when their implementation details are changed (e.g., order of steps in Newton's method or Mergesort, particular alphabet/programming language, number of tapes/states in TMs, etc.).

This becomes even more apparent in:

- our use of proper names for algorithms (Euclid's algorithm, Lucas-Lehmer algorithm, Gauss elimination, etc.)

What is at stake?

Resolving the tension is crucial, for it bears on the imposed identity conditions within any formal definition/framework:

OTOH, we surely want to say that various specific processes instantiate the same algorithm even when their implementation details are changed (e.g., order of steps in Newton's method or Mergesort, particular alphabet/programming language, number of tapes/states in TMs, etc.).

This becomes even more apparent in:

- our use of proper names for algorithms (Euclid's algorithm, Lucas-Lehmer algorithm, Gauss elimination, etc.)
- our assignment of properties that purport to be "objective" and model-independent, such as:
  - asymptotic costs to algorithms over countable domains (number-theoretic, sorting, etc.)
  - properties like stability, convergence etc. to algorithms over uncountable domains (numerical)

What is at stake?

OTOH, we want to safeguard against triviality, esp., wrt algorithmic analysis and complexity. E.g., consider the sorting algorithm for the list B:

TRIVIALSORT(B):

1. "**return** sort(B)"

This is definitely effective and has a very convenient running time! $\mathsf{O}(1)$.

The way to avoid such cases is by assuming that proper algorithms are those able to be couched in an appropriate machine model. But this shifts again the conceptual priority to symbolic computations.

What is at stake?

OTOH, we want to safeguard against triviality, esp., wrt algorithmic analysis and complexity. E.g., consider the sorting algorithm for the list B:

TrivialSort(B):

1. "**return** sort(B)"

This is definitely effective and has a very convenient running time! O(1).

The way to avoid such cases is by assuming that proper algorithms are those able to be couched in an appropriate machine model. But this shifts again the conceptual priority to symbolic computations.

Furthermore, the claim that an algorithm remains the same even when the exact sequence of actions change seems to fly in the face of almost every (logic) textbook definition of algorithm.

An etiology: sharpening the requirement for small steps

In a sense, the tension can be seen underlying already the informal requirement of small steps in the very idea of 'algorithm' itself.

An etiology: sharpening the requirement for small steps

In a sense, the tension can be seen underlying already the informal requirement of small steps in the very idea of ‘algorithm’ itself.

First approximation (still vague): “at any given state the application of the next step must be immediately clear and recognizable, in the sense that no actual thought is needed to perform it.

An etiology: sharpening the requirement for small steps

In a sense, the tension can be seen underlying already the informal requirement of small steps in the very idea of 'algorithm' itself.

First approximation (still vague): "at any given state the application of the next step must be immediately clear and recognizable, in the sense that no actual thought is needed to perform it.

Two possibilities to further formalize:

1. understand this requirement in the most absolute sense possible: algorithmic steps somehow be immediate and "intuitively obvious" (whatever that means)

An etiology: sharpening the requirement for small steps

In a sense, the tension can be seen underlying already the informal requirement of small steps in the very idea of 'algorithm' itself.

First approximation (still vague): "at any given state the application of the next step must be immediately clear and recognizable, in the sense that no actual thought is needed to perform it.

Two possibilities to further formalize:

1. understand this requirement in the most absolute sense possible: algorithmic steps somehow be immediate and "intuitively obvious" (whatever that means)

Preceding philosophical attempts to found mathematics in self-evident operations of that sort can guide us. Cf. Hilbert (1926):

An etiology: sharpening the requirement for small steps

In a sense, the tension can be seen underlying already the informal requirement of small steps in the very idea of 'algorithm' itself.

First approximation (still vague): "at any given state the application of the next step must be immediately clear and recognizable, in the sense that no actual thought is needed to perform it.

Two possibilities to further formalize:

1. understand this requirement in the most absolute sense possible: algorithmic steps somehow be immediate and "intuitively obvious" (whatever that means)

Preceding philosophical attempts to found mathematics in self-evident operations of that sort can guide us. Cf. Hilbert (1926):
Express the entities under consideration in ways that are "*intuitively present as immediate experience prior to all thought*". Fro this use of concrete signs "*whose shape is ... immediately clear and recognizable*" (e.g., stroke notation). Then, various aspects of these objects (concatenation, order of occurrence, etc.) are "*given intuitively ... as something that neither can be reduced to anything else nor requires reduction*" (p.376).

An etiology: sharpening the requirement for small steps

In a sense, the tension can be seen underlying already the informal requirement of small steps in the very idea of 'algorithm' itself.

First approximation (still vague): "at any given state the application of the next step must be immediately clear and recognizable, in the sense that no actual thought is needed to perform it.

Two possibilities to further formalize:

1. understand this requirement in the most absolute sense possible: algorithmic steps somehow be immediate and "intuitively obvious" (whatever that means)

Preceding philosophical attempts to found mathematics in self-evident operations of that sort can guide us. Cf. Hilbert (1926):
Express the entities under consideration in ways that are "*intuitively present as immediate experience prior to all thought*". Fro this use of concrete signs "*whose shape is ... immediately clear and recognizable*" (e.g., stroke notation). Then, various aspects of these objects (concatenation, order of occurrence, etc.) are "*given intuitively ... as something that neither can be reduced to anything else nor requires reduction*" (p.376).

The absolute-immediate idea of step leads naturally to the symbolic approach to formalizing algorithms (e,g., Turing, computable analysis, Russian school, etc.)

An etiology: sharpening the requirement for small steps

Two possibilities to further formalize:

An etiology: sharpening the requirement for small steps

Two possibilities to further formalize:

2. understand it relative to the computing agent: the agent deals with actions that are immediately recognizable in the sense that s/he has achieved such a degree of familiarity with them as to immediately recognize them as clear and obvious.

An etiology: sharpening the requirement for small steps

Two possibilities to further formalize:

2. understand it relative to the computing agent: the agent deals with actions that are immediately recognizable in the sense that s/he has achieved such a degree of familiarity with them as to immediately recognize them as clear and obvious.

Examples:

- μ-recursive functions or λ-calculus, take certain operations, such as composition or substitution, as immediately recognizable
- truth-table validity tests take the assignment of truth values to the basic logical connectives as primitive
- elem. school algorithms for long multiplication and long division take multiplication tables as primitive
- more advanced number-theoretic algorithms (e.g., Euclid's algorithm) take $(\pm \times \div)$, as well as $(\leq, \geq)$ as primitive.

An etiology: sharpening the requirement for small steps

Two possibilities to further formalize:

2. understand it relative to the computing agent: the agent deals with actions that are immediately recognizable in the sense that s/he has achieved such a degree of familiarity with them as to immediately recognize them as clear and obvious.

Examples:

- μ-recursive functions or λ-calculus, take certain operations, such as composition or substitution, as immediately recognizable
- truth-table validity tests take the assignment of truth values to the basic logical connectives as primitive
- elem. school algorithms for long multiplication and long division take multiplication tables as primitive
- more advanced number-theoretic algorithms (e.g., Euclid's algorithm) take $(\pm \times \div)$, as well as $(\leq, \geq)$ as primitive.

(Numerical) algorithms seem to tacitly assume ‘immediate steps’ in this sense of familiar-immediate primitives in the domains of interest.

An etiology: sharpening the requirement for small steps

Two possibilities to further formalize:

2. understand it relative to the computing agent: the agent deals with actions that are immediately recognizable in the sense that s/he has achieved such a degree of familiarity with them as to immediately recognize them as clear and obvious.

Examples:

- μ-recursive functions or λ-calculus, take certain operations, such as composition or substitution, as immediately recognizable
- truth-table validity tests take the assignment of truth values to the basic logical connectives as primitive
- elem. school algorithms for long multiplication and long division take multiplication tables as primitive
- more advanced number-theoretic algorithms (e.g., Euclid's algorithm) take $(\pm \times \div)$, as well as $(\leq, \geq)$ as primitive.

(Numerical) algorithms seem to tacitly assume 'immediate steps' in this sense of familiar-immediate primitives in the domains of interest.

The familiar-relative idea of step leads naturally to the abstract conception of 'algorithm'. This goes hand in hand with a model-/level-/structure-relative idea of 'algorithm' (as in Moschovakis, Gurevich, Blum et al.).

Defining ‘algorithms’: An additional tension in the goals

Development of mathematical definitions resembles development of Covid19 tests:

$$\text{sensitivity} \longleftrightarrow \text{specificity}$$

Defining ‘algorithms’: An additional tension in the goals

Development of mathematical definitions resembles development of Covid19 tests:

$$\text{sensitivity} \longleftrightarrow \text{specificity}$$

Similarly:

$$\text{inclusiveness/abstractness/generality} \longleftrightarrow \text{domain-specific fecundity}$$

Defining ‘algorithms’: An additional tension in the goals

Development of mathematical definitions resembles development of Covid19 tests:

$$\text{sensitivity} \longleftrightarrow \text{specificity}$$

Similarly:

$$\text{inclusiveness/abstractness/generality} \longleftrightarrow \text{domain-specific fecundity}$$

There usually is a number of acceptable ways of trading off these virtues, and a number of possible concepts encapsulating them.

Defining ‘algorithms’: One more tension in the goals

The quest for definitions of algorithms exemplifies this predicament perhaps most strikingly:

Defining ‘algorithms’: One more tension in the goals

The quest for definitions of algorithms exemplifies this predicament perhaps most strikingly:

OTOH, the desire for inclusiveness pushes in the direction of a more abstract concept. One that ideally captures both algorithms over countable and uncountable domains. Model-/level-/structure-relative approaches meet that desideratum (e.g., Gurevich and Moschovakis).

But they are too general/powerful to provide a yardstick against which to reasonably measure algorithmic costs (no standard of powerfulness of operations).

Defining ‘algorithms’: One more tension in the goals

The quest for definitions of algorithms exemplifies this predicament perhaps most strikingly:

OTOH, the desire for inclusiveness pushes in the direction of a more abstract concept. One that ideally captures both algorithms over countable and uncountable domains. Model-/level-/structure-relative approaches meet that desideratum (e.g., Gurevich and Moschovakis).

But they are too general/powerful to provide a yardstick against which to reasonably measure algorithmic costs (no standard of powerfulness of operations).

OTOH, the desire for the formal concept to underpin a rich theory of complexity pushes in the direction of more and more domain-specific formal concepts (TMs (ordinary and TTE), K&U, RAM, BSS).

These all (can) underpin theories of complexity (classical and real), but hardly any of these can be used for more than one domain (e.g., countable and uncountable computations alike (except maybe TTE but too cumbersome and low-level)).

TRIVIALTHANKS(PLS13):

1. **return** Thank you(PLS13);

This work has been partially supported by the ANR project
The Geometry of Algorithms – GoA (ANR-20-CE27-0004).