Towards a Formal Model of *Algorithms*

Declan Thompson

Fourth Year Talk, 22/23 May 2020

Philosophy of Computer Science

- Ethical issues
  - AI
  - Big data
  - Privacy
- Analysis of software development
  - Verification and correctness
  - Programming language semantics
  - Ontology of programs
- Artifacts in computer science
  - Implementation
  - Software vs. hardware
- Foundations of theoretical computer science
  - What is computable in theory? In practice?
  - What is computation?
  - What are algorithms?

What would an adequate formal characterisation of *algorithm* look like?

Research focus

- How should we understand claims about algorithms?
- What do we expect of a formal model of algorithms?

Computer scientists say things like:

- Program X implements Prim's algorithm.
- MergeSort and QuickSort are different sorting algorithms.
- The Euclidean algorithm is correct for finding the greatest common divisor of two positive integers.
- The Gale-Shapley algorithm runs in quadratic time.
- Shor's algorithm is a quantum algorithm.

Outline

1. What are algorithms?
   - Running Example: Prim's Algorithm
   - Two approaches to *algorithms*
2. Extant formal accounts of *algorithm*
   - The Traditional Approach
   - Algorithmic Realism
3. A New Direction
   - Trace Sets
   - Recovering Computability Theory

Outline

1. What are algorithms?
   - Running Example: Prim's Algorithm
   - Two approaches to *algorithms*
2. Extant formal accounts of *algorithm*
   - The Traditional Approach
   - Algorithmic Realism
3. A New Direction
   - Trace Sets
   - Recovering Computability Theory

Problem: Minimum Spanning Tree

Task (MST)

Given: A weighted connected simple graph $G = (V, E, w)$

Return: A spanning tree $G_1 = (V_1, E_1, w)$ of G which minimises

$$\sum_{e \in E_1} w(e)$$

Solution: Prim's Algorithm

Prim(G, v):

1. Intialize $V_1 = \{v\}$, $E_1 = \emptyset$, and set $G_1 = (V_1, E_1)$.
2. *While* there is an edge that connects a vertex in V_1 to a vertex not in V_1 *do*
   a. Find an edge $e = \{u, v'\}$ with smallest weight $w(e)$ such that $u \in V_1$ and $v' \notin V_1$.
   b. Set $V_1 = V_1 \cup \{v'\}$, $E_1 = E_1 \cup \{e\}$, and $G_1 = (V_1, E_1)$.
3. Output $G_1 = (V_1, E_1)$.

(Khoussainov and Khoussainova 2012, p. 172)

Claims about Prim's Algorithm

- **Theorem 17.3** If G is a connected weighted graph then the $Prim(G, v)$-algorithm produces a minimum spanning tree for G.

  (Khoussainov and Khoussainova 2012, p. 173)

- [I]f we use a Fibonacci heap to implement the min-priority queue Q, the running time of Prim's algorithm improves to $O(E + V \lg V)$.

  (Cormen et al. 2009, p. 636)

- Kruskal's algorithm is generally slower than Prim's algorithm

  (Sedgewick and Wayne 2011, p. 625)

What is Prim's algorithm?

- An abstract object?
- A mathematical object?
- A syntactic object?
- Something else?

What is an algorithm?

Editor:
We are making this communication intentionally short to leave as much room as possible for the answers.

1. Please define "Algorithm."
2. Please define "Formula."
3. Please state the difference.

(Wangsness and Franklin 1966, p. 243)

Since Turing, Kleene, Markov and others, we have several precise definitions which have proved to be equivalent. In each case a distinguished sufficiently powerful algorithmic language (= programming language) is specified and an algorithm is defined to be any program written in this language (Turing Machines, μ-recursive functions, normal Markov algorithms, and so on) terminating when executed.

(Huber 1966, p. 653)

A concept like "abstract algorithm" without reference to any algorithmic language does not exist. In order to specify an algorithm one has to give the specifications in some algorithmic language.

(Huber 1966, p. 654)

What is an algorithm?

Editor:
We are making this communication intentionally short to leave as much room as possible for the answers.

1. Please define "Algorithm."
2. Please define "Formula."
3. Please state the difference.

(Wangsness and Franklin 1966, p. 243)

Since Turing, Kleene, Markov and others, we have several precise definitions which have proved to be equivalent. In each case a distinguished sufficiently powerful algorithmic language (= programming language) is specified and an algorithm is defined to be any program written in this language (Turing Machines, μ-recursive functions, normal Markov algorithms, and so on) terminating when executed.

(Huber 1966, p. 653)

A concept like "abstract algorithm" without reference to any algorithmic language does not exist. In order to specify an algorithm one has to give the specifications in some algorithmic language.

(Huber 1966, p. 654)

What is an algorithm?

Editor:
We are making this communication intentionally short to leave as much room as possible for the answers.

1. Please define "Algorithm."
2. Please define "Formula."
3. Please state the difference.

(Wangsness and Franklin 1966, p. 243)

Since Turing, Kleene, Markov and others, we have several precise definitions which have proved to be equivalent. In each case a distinguished sufficiently powerful algorithmic language (= programming language) is specified and an algorithm is defined to be any program written in this language (Turing Machines, μ-recursive functions, normal Markov algorithms, and so on) terminating when executed.

(Huber 1966, p. 653)

A concept like "abstract algorithm" without reference to any algorithmic language does not exist. In order to specify an algorithm one has to give the specifications in some algorithmic language.

(Huber 1966, p. 654)

What is an algorithm?

Editor:
We are making this communication intentionally short to leave as much room as possible for the answers.

1. Please define "Algorithm."
2. Please define "Formula."
3. Please state the difference.

(Wangsness and Franklin 1966, p. 243)

To me the word algorithm denotes an abstract method for computing some function, while a program is an embodiment of a computational method in some programming language. I can write several different programs for the same algorithm (e.g., in ALGOL 60 and in PL/I, assuming these languages are given an unambiguous interpretation).

Of course if I am pinned down and asked to explain more precisely what I mean by these remarks, I am forced to admit that I don't know any way to define any particular algorithm except in a programming language. … But I believe algorithms were present long before Turing et al. formulated them, just as the concept of the number "two" was in existence long before the writers of first grade textbooks and other mathematical logicians gave it a certain precise definition.

(Knuth 1966, p. 654)

What is an algorithm?

Editor:
We are making this communication intentionally short to leave as much room as possible for the answers.

1. Please define "Algorithm."
2. Please define "Formula."
3. Please state the difference.

(Wangsness and Franklin 1966, p. 243)

To me the word algorithm denotes an abstract method for computing some function, while a program is an embodiment of a computational method in some programming language. I can write several different programs for the same algorithm (e.g., in ALGOL 60 and in PL/I, assuming these languages are given an unambiguous interpretation).

Of course if I am pinned down and asked to explain more precisely what I mean by these remarks, I am forced to admit that I don't know any way to define any particular algorithm except in a programming language. … But I believe algorithms were present long before Turing et al. formulated them, just as the concept of the number "two" was in existence long before the writers of first grade textbooks and other mathematical logicians gave it a certain precise definition.

(Knuth 1966, p. 654)

Two approaches to *algorithms*

The traditional approach

Talk about *algorithms* should be understood as talk about *programs*, within the traditional bounds of computability theory.

- Reductive/nominalist approach
- Strong Church's Thesis

Algorithmic Realism

Algorithms should be understood as legitimate mathematical objects, distinct in kind from programs.

Outline

The Traditional Approach

The Traditional Approach

Talk about *algorithms* should be understood as talk about *programs*.

Programs are sets of instructions written in formal *programming languages*.

- Unambiguous
- Finitary

LazyPrimMST

```
public class LazyPrimMST
{
   private boolean[] marked;        // MST vertices
   private Queue<Edge> mst;         // MST edges
   private MinPQ<Edge> pq;          // crossing (and ineligible) edges

   public LazyPrimMST(EdgeWeightedGraph G)
   {
      pq = new MinPQ<Edge>();
      marked = new boolean[G.V()];
      mst = new Queue<Edge>();

      visit(G, 0);   // assumes G is connected (see Exercise 4.3.22)
      while (!pq.isEmpty())
      {
         Edge e = pq.delMin();                      // Get lowest-weight
         int v = e.either(), w = e.other(v);        // edge from pq.
         if (marked[v] && marked[w]) continue;      // Skip if ineligible.
         mst.enqueue(e);                            // Add edge to tree.
         if (!marked[v]) visit(G, v);               // Add vertex to tree
         if (!marked[w]) visit(G, w);               // (either v or w).
      }
   }

   private void visit(EdgeWeightedGraph G, int v)
   {  // Mark v and add to pq all edges from v to unmarked vertices.
      marked[v] = true;
      for (Edge e : G.adj(v))
         if (!marked[e.other(v)]) pq.insert(e);
   }

   public Iterable<Edge> edges()
   {  return mst;  }

   public double weight()     // See Exercise 4.3.31.
}
```

(Sedgewick and Wayne 2011, p. 619)

Algorithm vs. Program

Prim(G, v):

1. Intialize $V_1 = \{v\}$, $E_1 = \emptyset$, and set $G_1 = (V_1, E_1)$.
2. *While* there is an edge that connects a vertex in V_1 to a vertex not in V_1 *do*
   a. Find an edge $e = \{u, v'\}$ with smallest weight $w(e)$ such that $u \in V_1$ and $v' \notin V_1$.
   b. Set $V_1 = V_1 \cup \{v'\}$, $E_1 = E_1 \cup \{e\}$, and $G_1 = (V_1, E_1)$.
3. Output $G_1 = (V_1, E_1)$.

(Khoussainov and Khoussainova 2012, p. 172)

public LazyPrimMST(EdgeWeightedGraph G)

- EdgeWeightedGraph is an abstract data type defined elsewhere
- G is represented by an array of sets of edges

$$\begin{bmatrix} \{\langle\{0,1\},1\rangle,\langle\{0,2\},1\rangle\} \\ \{\langle\{0,1\},1\rangle,\langle\{1,2\},2\rangle,\langle\{1,3\},2\rangle\} \\ \{\langle\{0,2\},1\rangle,\langle\{1,2\},2\rangle,\langle\{2,3\},3\rangle\} \\ \{\langle\{1,3\},2\rangle,\langle\{2,3\},3\rangle\} \end{bmatrix}$$

```
Edge e = pq.delMin(); // Get lowest-weight edge
```

- pq is a *priority queue*
- pq.delMin() returns the edge with the lowest weight *that was added to pq first*

Algorithms $\neq$ Programs

- Programs must represent abstract objects.
  Algorithms work directly with them.
- Programs must be fully specified.
  Algorithms leave room for implementation details.
- Programs are strongly language dependent.
  Algorithms are somehow language independent.

$$\begin{array}{c} \text{algorithm} \\ \Big| \\ \text{implemented by} \\ \Big\downarrow \\ \text{program} \end{array}$$

Algorithmic Realism

Algorithmic Realism

Algorithms should be understood as legitimate mathematical objects, distinct in kind from programs.

What sort of mathematical object?

Equivalence classes of programs

(Milner 1971; Yanofsky 2011)

- Algorithms as equivalence classes under a relation of "essential sameness" between programs (cf. Hume's principle)
- Walter Dean (2007, 2016) has cast doubt on this approach

Generalised programs

- Yiannis Moschovakis (1984, 1998) uses abstract recursion
- Yuri Gurevich (2000, 2001) generalises standard machine models

Gurevich's Sequential Algorithms

Gurevich (2000) argues *sequential algorithms* are associated with:

- A set S of states
  - $s \in S$ is a first order structure
  - $s, t \in S$ share the same domain and vocabulary V
- A transition function $\tau : S \to S$
  - There is a finite set of terms T over V such that when $s_1, s_2 \in S$ agree on the values of all $t \in$, τ modifies them in the same way.

Generalised programs

- Allow programs to operate directly with abstract objects
- Allow arbitrary functions as operations

Issues

- We still can't handle under-determined steps (e.g. 2a)
- Heavy reliance on choice of vocabulary
- How to identify the right level of abstraction?

Taking stock

Traditional approach Too fine-grained and language dependent

Equivalence classes Does the "essential sameness" relation exist?

Generalised programs Forced into arbitrary choices

What is our focus?

- Algorithms as effective procedures
- Algorithms as distinct methods for solving a problem

Taking stock

Traditional approach Too fine-grained and language dependent

Equivalence classes Does the "essential sameness" relation exist?

Generalised programs Forced into arbitrary choices

What is our focus?

- Algorithms as effective procedures
- **Algorithms as distinct methods for solving a problem**

A More Radical Algorithmic Realism

Idea: Divorce algorithms from models of computation.
Think in terms of *behaviours*.

Goals

- ☐ Keep things abstract
- ☐ Allow for under-determined steps
- ☐ Connect back to computability theory

Outline

1. What are algorithms?
   - Running Example: Prim's Algorithm
   - Two approaches to *algorithms*

2. Extant formal accounts of *algorithm*
   - The Traditional Approach
   - Algorithmic Realism

3. A New Direction
   - Trace Sets
   - Recovering Computability Theory

Algorithmic traces

An *execution trace* is the sequence of updates when the algorithm is run.

Idea

An algorithm is the set of its own traces.

Operations

Each step in a trace corresponds to an abstract operation:

- function
- relation
- *behaviour*

Idea

A trace set is assembled from a set of operations.

Tasks and Contexts

Algorithms are solutions to problems, or *tasks*.
Algorithms for the same task use the same operations.

Idea

A task in a context provides a set of assumed operations

Kruskal(G, v):

a. Set $T = (V, E_1)$, $E_1 = \emptyset$. (Thus T has n components).
b. *While* T has more than one component do
   i. Find an edge e such that $e = \{x, y\}$ has the smallest weight, and x and y belong to two distinct components of T.
   ii. Declare E_1 to be $E_1 \cup \{e\}$.
c. Output $(V, E1)$.

(Khoussainov and Khoussainova 2012, p. 175)

Trace Sets

Trace set

Let $\mathfrak{T}$ be the set of all traces.
An *n*-ary *trace set* A for task F in context $\mathfrak{c}$ is a function $\mathrm{A} : \mathfrak{D}^n \to 2^{\mathfrak{T}}$ such that

1. Each $\sigma \in \mathrm{A}(\vec{\mathbf{a}})$ is a concatenation of assumed operations for F in $\mathfrak{c}$;
2. If $\sigma \in \mathrm{A}(\vec{\mathbf{a}})$ and $\tau \in \mathrm{A}(\vec{\mathbf{b}})$ follow a different sequence of operations, then some assumed operation must distinguish them.

Trace Sets

Trace set

Let $\mathfrak{T}$ be the set of all traces.
An n-ary *trace set* A for task F in context $\mathfrak{c}$ is **a function** $\mathrm{A} : \mathfrak{D}^n \to 2^{\mathfrak{T}}$ such that

1. Each $\sigma \in \mathrm{A}(\vec{\mathbf{a}})$ is a concatenation of assumed operations for F in $\mathfrak{c}$;
2. If $\sigma \in \mathrm{A}(\vec{\mathbf{a}})$ and $\tau \in \mathrm{A}(\vec{\mathbf{b}})$ follow a different sequence of operations, then some assumed operation must distinguish them.

Trace Sets

Trace set

Let $\mathfrak{T}$ be the set of all traces.
An n-ary *trace set* A **for task** F in context $\mathfrak{c}$ is a function $\mathrm{A} : \mathfrak{D}^n \to 2^{\mathfrak{T}}$ such that

1. Each $\sigma \in \mathrm{A}(\vec{\mathbf{a}})$ is a concatenation of assumed operations for F in $\mathfrak{c}$;
2. If $\sigma \in \mathrm{A}(\vec{\mathbf{a}})$ and $\tau \in \mathrm{A}(\vec{\mathbf{b}})$ follow a different sequence of operations, then some assumed operation must distinguish them.

Trace Sets

Trace set

Let $\mathfrak{T}$ be the set of all traces.
An *n*-ary *trace set* A for task F **in context $\mathfrak{c}$** is a function $\mathrm{A} : \mathfrak{D}^n \to 2^{\mathfrak{T}}$ such that

1. Each $\sigma \in \mathrm{A}(\vec{\mathbf{a}})$ is a concatenation of assumed operations for F in $\mathfrak{c}$;
2. If $\sigma \in \mathrm{A}(\vec{\mathbf{a}})$ and $\tau \in \mathrm{A}(\vec{\mathbf{b}})$ follow a different sequence of operations, then some assumed operation must distinguish them.

Trace Sets

Trace set

Let $\mathfrak{T}$ be the set of all traces.
An *n*-ary *trace set* A for task F in context $\mathfrak{c}$ is a function $\mathrm{A} : \mathfrak{D}^n \to 2^{\mathfrak{T}}$ such that

1. Each $\sigma \in \mathrm{A}(\vec{\mathbf{a}})$ is a concatenation of assumed operations for F in $\mathfrak{c}$;
2. If $\sigma \in \mathrm{A}(\vec{\mathbf{a}})$ and $\tau \in \mathrm{A}(\vec{\mathbf{b}})$ follow a different sequence of operations, then some assumed operation must distinguish them.

Trace Sets

Trace set

Let $\mathfrak{T}$ be the set of all traces.
An n-ary *trace set* A for task F in context $\mathfrak{c}$ is a function $\mathrm{A} : \mathfrak{D}^n \to 2^{\mathfrak{T}}$ such that

1. Each $\sigma \in \mathrm{A}(\vec{\mathbf{a}})$ is a concatenation of assumed operations for F in $\mathfrak{c}$;
2. If $\sigma \in \mathrm{A}(\vec{\mathbf{a}})$ and $\tau \in \mathrm{A}(\vec{\mathbf{b}})$ follow a different sequence of operations, then some assumed operation must distinguish them.

Trace Sets

Goals

- ☐ Keep things abstract
- ☐ Allow for under-determined steps
- ☐ Connect back to computability theory

The Return of Computability Theory

What is computable in theory by a finite agent?

- What is an *effective procedure*?
- Models of computation: Turing machines, μ-recursive functions, ...
- Church-Turing Thesis: The intuitively computable functions are the Turing-computable functions

What is computable by a trace set?

- Assumed operations can be uncomputable
- Traces can be infinitely long
- A trace might describe an uncomputable sequence

Are these problems?

Control Equivalence

Control equivalence

A *control equivalence* is an equivalence relation on the steps in the traces of a trace set, such that if $\sigma[\alpha] \leftrightarrows \tau[\beta]$ then either

1. the same assumed operation is applied to both $\sigma[\alpha]$ and $\tau[\beta]$, and $\sigma[\alpha+1] \leftrightarrows \tau[\beta+1]$, or
2. some assumed operation distinguishes $\sigma[\alpha]$ from $\tau[\beta]$.

A *finite control equivalence* is a control equivalence with finitely many equivalence classes.

Finite Control

Finite Control

Theorem

Let $\mathfrak{M} = \langle S, \Sigma, \Gamma, \delta, s_0, s_a, s_r \rangle$ be any Turing machine. Then $\textsc{Run}_{\mathfrak{M}}$ has a finite control equivalence.

Finite Control

Theorem

Let $\mathfrak{M} = \langle S, \Sigma, \Gamma, \delta, s_0, s_a, s_r \rangle$ be any Turing machine. Then $\textsc{Run}_{\mathfrak{M}}$ has a finite control equivalence.

Theorem

Let $\textsc{a}$ be a trace set using Turing machine operations and Γ a finite alphabet such that

1. $\textsc{a}(v) \neq \emptyset$ iff $v \in \Gamma^*$; *and*
2. $|\textsc{a}(v)| = 1$ for all $v \in \Gamma^*$; *and*
3. $\sigma \in \textsc{a}(v)$ is a sequence of Turing machine configurations; *and*
4. $\textsc{a}$ has a finite control equivalence $\leftrightarrows$.

Then there is a Turing machine $\mathfrak{M} = \langle S, \Gamma, \Gamma, \delta, s_0, s_a, s_r \rangle$ corresponding to the sequences of configurations given by $\textsc{a}$.

Summary

- Algorithms $\neq$ programs
- Extant accounts of algorithmic realism force arbitrary choices
    - Heavy reliance on traditional models of computation
- Trace sets provide a promising new model for *algorithms*
    - Sufficiently abstract
    - Natural fit with intuition
    - Compatible with computability theory

What next?

- Specifying trace sets
- Recovering complexity theory
- Implementation

Outside Computer Science

Applications:

- Philosophy of mind
- Constructivism
- Law
- Many more...

Intuitive similarities:

- Plans/procedures
- Mathematical proofs
- Stories

Thank you

- Thomas H. Cormen et al., eds. *Introduction to Algorithms*. 3rd ed. Cambridge, Mass: MIT Press, 2009. 1292 pp. ISBN: 978-0-262-03384-8 978-0-262-53305-8.
- Walter Dean. “What Algorithms Could Not Be”. Rutgers University - Graduate School - New Brunswick, 2007.
- Walter Dean. “Algorithms and the Mathematical Foundations of Computer Science”. In: *Gödel’s Disjunction: The Scope and Limits of Mathematical Knowledge*. First edition. Oxford, United Kingdom: Oxford University Press, 11 Aug. 2016, pp. 19–66. ISBN: 978-0-19-182037-3.
- Yuri Gurevich. “Sequential Abstract-State Machines Capture Sequential Algorithms”. In: *ACM Transactions on Computational Logic* 1.1 (1 July 2000), pp. 77–111.
- Yuri Gurevich. *The Sequential ASM Thesis*. 2001.
- Hartmut G. M. Huber. “Algorithm and Formula”. In: *Communications of the ACM* 9.9 (1966), pp. 653–654.

- Bakhadyr Khoussainov and Nodira Khoussainova. *Lectures on Discrete Mathematics for Computer Science*. Algebra and Discrete Mathematics v. 3. New Jersey: World Scientific, 2012. 346 pp. ISBN: 978-981-4340-50-2.
- Donald E. Knuth. "Algorithm and Program; Information and Data". In: *Communications of the ACM* 9.9 (1966), p. 654.
- Robin Milner. "An Algebraic Definition of Simulation Between Programs". In: *Proceedings of the 2nd International Joint Conference on Artificial Intelligence*. IJCAI'71. London, England: Morgan Kaufmann Publishers Inc., 1971, pp. 481–489.
- Yiannis N Moschovakis. "Abstract Recursion as a Foundation for the Theory of Algorithms". In: *Computation and Proof Theory*. Ed. by Egon Börger et al. Vol. 1104. Berlin, Heidelberg: Springer Berlin Heidelberg, 1984, pp. 289–364. ISBN: 978-3-540-13901-0 978-3-540-39119-7.
- Yiannis N Moschovakis. "On Founding the Theory of Algorithms". In: *Truth in mathematics* (1998), pp. 71–104.

- Robert Sedgewick and Kevin Daniel Wayne. *Algorithms*. 4th ed. Upper Saddle River, NJ: Addison-Wesley, 2011. 955 pp. ISBN: 978-0-321-57351-3.
- T. Wangsness and J. Franklin. "“Algorithm” and “Formula”". In: *Communications of the ACM* 9.4 (1 Apr. 1966), p. 243.
- N. S. Yanofsky. "Towards a Definition of an Algorithm". In: *Journal of Logic and Computation* 21.2 (1 Apr. 2011), pp. 253–286.

What do the textbooks say?

- "Informally, an ***algorithm*** is any well-defined computational procedure that takes some value, or set of values, as ***input*** and produces some value, or set of values, as ***output***. An algorithm is thus a sequence of computational steps that transform the input into the output." (Cormen et al. 2009, p. 5)
- "The term *algorithm* is used in computer science to describe a finite, deterministic, and effective problem-solving method suitable for implementation as a computer program." (Sedgewick and Wayne 2011, p. 4)
- Brassard and Bratley
- Harel
- Knuth