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**

<table>
<thead>
<tr>
<th>Task (MST)</th>
</tr>
</thead>
</table>
| **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. Initialize $V_1 = \{v\}$, $E_1 = \emptyset$, 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)

- If 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?

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, $\mu$-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, $\mu$-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, $\mu$-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?

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)
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

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
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**

```java
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);
            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. Initialize *V*₁ = {*v*}, *E*₁ = ∅, 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* ∈ *V*₁ and *v'* /∈ *V*₁.
   
   b. Set *V*₁ = *V*₁ ∪ {*v'*}, *E*₁ = *E*₁ ∪ {*e*}, and *G*₁ = (*V*₁, *E*₁).

(Khoussainov and Khoussainova 2012, p. 172)

```java
public LazyPrimMST(EdgeWeightedGraph G)
    EdgeWeightedGraph is an abstract data type defined elsewhere
    G is represented by an array of sets of edges
    
    {{{{0, 1}, 1}, {{0, 2}, 1}}
     {{{0, 1}, 1}, {{1, 2}, 2}, {{1, 3}, 2}}
     {{{0, 2}, 1}, {{1, 2}, 2}, {{2, 3}, 3}}
     {{{1, 3}, 2}, {{2, 3}, 3}}
    }

    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
```
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.
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

- 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 $\tau : S \rightarrow S$
  - 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 $t \in T$, $\tau$ 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
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.
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 \( e = \{x, y\} \) 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 \((V, E_1)\).

(Khoussainov and Khoussainova 2012, p. 175)
Trace Sets

Trace set

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

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

Trace set

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

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

Trace set

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

1. Each $\sigma \in \Delta(a)$ is a concatenation of assumed operations for $F$ in $c$;
2. If $\sigma \in \Delta(a)$ and $\tau \in \Delta(b)$ follow a different sequence of operations, then some assumed operation must distinguish them.
Trace Sets

Trace set

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

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

**Trace set**

Let \( \mathcal{T} \) be the set of all traces.

An \( n \)-ary trace set \( A \) for task \( F \) in context \( c \) is a function \( A : \mathcal{D}^n \to 2^\mathcal{T} \) such that

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

<table>
<thead>
<tr>
<th>Trace set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Let $\mathcal{T}$ be the set of all traces. An $n$-ary trace set $A$ for task $F$ in context $c$ is a function $A : \mathcal{D}^n \rightarrow 2^\mathcal{T}$ such that</td>
</tr>
<tr>
<td>1 Each $\sigma \in A(\vec{a})$ is a concatenation of assumed operations for $F$ in $c$;</td>
</tr>
<tr>
<td>2 If $\sigma \in A(\vec{a})$ and $\tau \in A(\vec{b})$ follow a different sequence of operations, then some assumed operation must distinguish them.</td>
</tr>
</tbody>
</table>
Trace Sets

Goals

- Keep things abstract
- Allow for under-determined steps
- Connect back to computability theory
What is computable in theory by a finite agent?
- What is an effective procedure?
- Models of computation: Turing machines, $\mu$-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 \( \sigma[\alpha] \Leftrightarrow \tau[\beta] \) then either

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

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

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

Theorem

Let $a$ be a trace set using Turing machine operations and $\Gamma$ a finite alphabet such that

1. $a(v) \neq \emptyset$ iff $v \in \Gamma^*$;

2. $|a(v)| = 1$ for all $v \in \Gamma^*$;

3. $\sigma \in a(v)$ is a sequence of Turing machine configurations;

4. $a$ has a finite control equivalence $\leftrightarrow$.

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

Let $\mathcal{M} = \langle S, \Sigma, \Gamma, s_0, s_a, s_r \rangle$ be any Turing machine. Then $\text{RUN}_{\mathcal{M}}$ has a finite control equivalence.
Finite Control

Theorem

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

Theorem

Let $A$ be a trace set using Turing machine operations and $\Gamma$ a finite alphabet such that

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

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

- Algorithms ≠ 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


“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