Dijkstra's Algorithm without the distributivity axiom: a proof of correctness in Agda

Leonhard D. Markert and Dominic P. Mulligan

Computer Laboratory, University of Cambridge

Abstract Dijkstra's Algorithm is typically presented as operating on graphs with numeric arc weights, but a more general form of the algorithm exists that operates on graphs where arc weights are drawn from a large class of semirings. In this setting, standard correctness proofs rely on the distributivity property of semirings to establish that the algorithm computes globally optimal path weights over every path from a single source node to all potential destinations, a process which can be interpreted as finding a fixed point for a certain matrix equation. We present a mechanised proof that Dijkstra's Algorithm can solve these matrix equations even after weakening the semiring axioms so that distributivity does not hold. Fixed points to matrix equations for non-distributive algebras can be interpreted as representing 'locally optimal' path weights. We use Agda for our implementation and proof, making use of dependent types and some of Agda's more cutting edge features—such as induction-recursion—to structure our algorithm and correctness proof.

Keywords: Dijkstra's Algorithm, semirings, interactive theorem proving

1 Introduction

Few algorithms are better known than Dijkstra's Algorithm [7] amongst working Computer Scientists, underlining the importance of shortest path algorithms to the field. Most textbook presentations of Dijkstra's Algorithm (see, for example, [5, Chapter 24]) are particularly concrete, presenting the algorithm as operating on directed graphs with non-negative numeric arc weights—typically positive reals or naturals. Interestingly, though potentially less well known, the algorithm exists in a more general form where path weights are not numeric but drawn from a large class of *semirings* [14].

Recall that a semiring $\langle S, \oplus, \otimes, 0, 1\rangle$ is composed of a commutative monoid $\langle S, \oplus, 0\rangle$ and a monoid $\langle S, \otimes, 1\rangle$ such that two *distributivity* and an *annihilation* axiom hold, connecting the two substructures:

$$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c) \qquad (b \oplus c) \otimes a = (b \otimes a) \oplus (c \otimes a) \qquad 0 \otimes a = 0$$

Now, given an adjacency matrix over a semiring, $\mathbf{A}$, representing a weighted, directed graph, the generalised shortest-path problem is the task of finding a

matrix $\mathbf{A}^*$ such that

$$\mathbf{A}^*(i,\ j) = \bigoplus_{p \in \mathcal{P}(i,\ j)} w_{\mathbf{A}}(p)$$

where $\mathcal{P}(i,\ j)$ represents the set of all paths from node i to node j and $w_{\mathbf{A}}(p)$ is the *weight* of a path. Given a path $p = v_0, v_1, \ldots, v_k$—a sequence of nodes—this is defined as

$$w_{\mathbf{A}}(p) = \mathbf{A}(v_0,\ v_1) \otimes \mathbf{A}(v_1,\ v_2) \otimes \cdots \otimes \mathbf{A}(v_{k-1},\ v_k)$$

with the empty path assigned the weight 1. This matrix, $\mathbf{A}^*$, need not always exist but if it does then it is a solution for $\mathbf{R}$ in the following right equation[1]

$$\mathbf{R} = (\mathbf{R} \otimes \mathbf{A}) \oplus \mathbf{I}$$

where $\mathbf{I}$ is the identity matrix, and the multiplication and addition operations of the semiring have been lifted to matrix multiplication and addition in the 'obvious' manner. Solutions to this equation encode the path weights of shortest paths between nodes of the graph. Dijkstra's Algorithm, a single-source shortest path algorithm, may then be seen as computing one row of a solution to this equation—the i^{th} row for source node i.

A compelling advantage of working in this algebraic setting is that the semiring parameterising the algorithm can be varied. Instantiating the algorithm with particular semirings causes the algorithm to compute shortest paths through the graph, *à la* classical presentations of Dijkstra, whereas others cause the algorithm to compute maximum bottleneck capacity, and so on. The key idea here is that a generic, parametric implementation of a shortest path algorithm can be developed and then instantiated with different semirings, and thereafter uniformly used to calculate different properties of graphs. Space is limited here, therefore the interested reader can find a more thorough introduction to this material in [10].

Further, in this abstract, algebraic setting, we may also ask what happens when we vary the structure parameterising our algorithm, for example by weakening or otherwise varying the axioms of a semiring. In particular, it can be seen that Dijkstra's Algorithm relies crucially on distributivity to compute *globally optimal* paths. However, recent work on mathematically modelling the Border Gateway Protocol (BGP), the Internet routing protocol which maintains connectivity between Internet Service Providers, has led to the idea of *locally* optimal paths (also called *equilibrium solutions*), where the underlying algebraic structure lacks the distributivity property [19]. In this relaxed setting, Dijkstra's Algorithm also computes solutions to the right fixpoint equation above, but these solutions now encode the path weights of locally optimal paths, rather than the globally optimal paths its semiring-oriented cousin computes.

In this paper, we present a purely-functional implementation of a generalised, non-distributive variant of Dijkstra's Algorithm, along with a mechanised proof of

[1] And also a solution for an analogous left equation.

its correctness (see Section 4). Our algorithm is a variant of Dijkstra's Algorithm in the sense that it maintains the queue discipline characteristic of that algorithm: the next node to be visited is always the unvisited node with the minimum distance estimate from the source node. Further, our algorithm is parameterised by a non-distributive algebraic structure—a 'Sobrinho Algebra', after João Luís Sobrinho—which distills the algebraic properties assumed, but left implicit, in previous work on this topic into a concrete structure whose properties can be explored (see Section 3). Our correctness proof establishes that our algorithm does indeed compute the Right Local Solution to the fixpoint equation mentioned above, given a graph's adjacency matrix over a Sobrinho Algebra. In particular, we believe that this is the first mechanised, axiom-free correctness proof of a shortest path algorithm using the 'algebraic approach', rather than following more traditional proof strategies. We use Agda for our implementation and proof of correctness, and we make extensive use of Agda's dependent types, and some more cutting edge features of the language—such as induction-recursion—to structure and complete the proof.

Implementation Agda [17] is a dependently-typed programming language *cum* proof assistant for higher-order intuitionistic logic. In contrast to similar systems [1, 2] proof terms are hand constructed via a type-directed refinement process, rather than constructed via tactic-oriented metaprogramming.

Agda has a uniform syntax, though one syntactic novelty is a flexible system of user-declared Unicode mixfix identifiers [6] with 'holes' in an identifier being denoted by underscores. We write $(x : A) \to B$ for the dependent function space where x may occur in B, write $A \to B$ when x does not occur in B, and write $\{x : A\} \to B$ (sometimes making use of the shorthands $\forall\ x \to B$ and $\forall\ \{x\} \to B$), when types can be inferred. We write $\Sigma\ A\ B$ for the dependent sum type whose first projection has type A, and write $A \times B$ when the second projection does not depend on the first, and write $\exists\ \lambda\ x \to P$ for the dependent sum type when the type of the first projection can be inferred. Dependent sums are constructed using the comma constructor: $x\ ,\ y$. Propositional equality between two types is written $A \equiv B$ and has a single canonical inhabitant, refl. Lastly, we write $A \uplus B$ for disjoint union, with constructors inj_1 and inj_2, and $\neg\ A$ for negation.

2 Basic definitions

Matrices and graph nodes We write Vec $A\ n$ for a length-indexed list containing elements of type A with length n. We write Matrix $A\ m\ n$ for the type of $m \times n$-dimensional matrices containing elements of type A, implemented as a vector of vectors. We use finite sets, where Fin n is intuitively the type of natural numbers 'of at most n', to index into matrices and represent graph nodes—this type has a decidable equality for all n. We write Subset n for a fixed-length list of length n, which partitions a finite set into elements that lie 'inside' and 'outside' of the set, to implement sets of nodes. At each index i of the vector are one of two flags—inside or outside—denotating whether the i^{th} element of the finite set in

question is inside or outside the described subset, i.e. a partitioning of a finite set into two new sets.

Assume an algebraic structure with carrier type Carrier, a decidable equality _≈_ and left multiplicative identity 1# (structures of this form will be further discussed in Section 3). We define an n-dimensional adjacency matrix over this structure as a record Adj $(n : \mathbb{N})$ parameterised by the dimension, and with two fields: matrix, the underlying adjacency matrix of type Matrix Carrier n n, and diag, a proof that diagonal elements of matrix are all equivalent to 1#.

Big sums We introduce a notion of lightweight 'big sum' [3] that will be used in our algorithm and proof of correctness when calculating path weights. Though in the rest of the paper they will be used over Sobrinho Algebras, we here define path weight sums over commutative monoids for convenience as they are well supported by the Standard Library, and Sobrinho Algebras subsume commutative monoids. We explicitly require a proof of idempotency whenever needed.

We use the function fold to define sums over subsets of finite sets using the underlying monoid's identity element ε and binary operator _•_:

```
fold : ∀ {n} → (Fin n → Carrier) → Subset n → Carrier
fold f []              = ε
fold f (inside  :: xs) = f zero • fold (f ∘ suc) xs
fold f (outside :: xs) =          fold (f ∘ suc) xs
```

Intuitively, for a subset of a finite set of size n, the function call fold f xs enumerates all n possible elements of the set, testing each in turn whether it is an element of the subset described by xs, acting on the element if so, ignoring it otherwise. For convenience we provide a syntax declaration for fold, so that the notation ⨁[$x \leftarrow v$] e denotes the application fold $(\lambda\ x \to e)$ v.

Trivially, we have that folding over an empty set (written ⊥) is equivalent to the neutral element of the monoid, and folding over a singleton set containing an element, written ⟦ i ⟧ for each element i, is equivalent to applying the function f to i. These facts are expressed as the lemmas fold-⊥ and fold-⟦i⟧, respectively, which we omit here. Folding a function f over a union of two subsets, xs and ys, is equivalent to folding over xs and ys separately and combining the two results with the commutative monoid's binary operator, _•_, whenever the operator is idempotent, as expressed by the following lemma, fold-∪:

```
fold-∪ : ∀ {n} (idp : Idempotent _•_) f (xs : Subset n) (ys : Subset n) →
         fold f (xs ∪ ys) ≈ fold f xs • fold f ys
```

The proof proceeds by simultaneous induction on both subsets. For each element of the two sets we must consider whether it lies inside or outside of the subsets being described by xs and ys.

Finally, we have an extensionality property, namely that folding two different functions across the same set results in equivalent values if the functions agree pointwise on all elements in the set. This is expressed in the lemma fold-cong:

```
fold-cong : ∀ {n} f g (xs : Subset n) → (∀ i → i ∈ xs → f i ≈ g i) →
            fold f xs ≈ fold g xs
```

The proof proceeds by induction on xs and is omitted.

Sorted vectors We define an indexed family of types of sorted vectors that we will use in Section 4 to implement a priority queue of unvisited nodes. Here, for generality we keep the particular type used to implement priorities abstract, and any type with a decidable total order structure defined over them will suffice.

Note that we prefer working with a linear sorted data structure, compared to a balanced binary tree such as Agda's existing implementation of AVL trees in Data.AVL, to simplify proofs. Using a length-indexed data structure also allows us to straightforwardly statically assert the non-emptiness of our priority queue by mandating that the queue's length must be of the form suc n, for some n.

Throughout this Section we fix a decidable total order record, DecTotalOrder and write Carrier, $\leq$ and $\leq?$ for the ordering's carrier set, ordering relation, and proof that the ordering relation is decidable, respectively. Assuming this, we define a type of sorted vectors, or sorted lists indexed by their length:

```
mutual
  data SortedVec : ℕ → Set (ℓ₂ ⊔ a) where
    []          : SortedVec 0
    _::_⟨_⟩ : ∀ {n} → (y : Carrier) → (ys : SortedVec n) →
                      (y≼ys : y ≼ ys) → SortedVec (ℕ.suc n)

  _≼_ : ∀ {n} → Carrier → SortedVec n → Set ℓ₂
  x ≼ []                  = Lift ⊤
  x ≼ (y :: ys ⟨ prf ⟩)  = (x ≤ y) × (x ≼ ys)
```

Our 'cons' constructor, _::_⟨_⟩, takes a proof that the head element *dominates* the tail of the list. The domination relation, _≼_, is defined mutually with our type definition via induction-recursion [8] making it impossible to construct a vector that is not sorted. The relation is decidable and also *quasi-transitive* in the sense that if x dominates xs and y is less than x according to our total order then y also dominates xs (proof omitted):

```
≼-trans : ∀ {n y x} → (xs : SortedVec n) → x ≼ xs → y ≤ x → y ≼ xs
```

The insertion of an element into a sorted vector is defined by mutual recursion between two functions insert and ≼-insert. The function insert places the inserted element in the correct position in the vector, modifying the length index, whilst ≼-insert constructs the required domination proof for the new element:

```
mutual
  insert : ∀ {n} → Carrier → SortedVec n → SortedVec (ℕ.suc n)
  insert x []  = x :: [] ⟨ lift tt ⟩
  insert x (y :: ys ⟨ prf ⟩) with x ≤? y
  ... | yes lt  = x :: y :: ys ⟨ prf ⟩ ⟨ lt , ≼-trans ys prf lt ⟩
  ... | no ¬lt  = y :: insert x ys ⟨ ≼-insert ys (¬x≤y→y≤x ¬lt) prf ⟩
```

```
≼-insert : ∀ {n x y} → (ys : SortedVec n) → y ≤ x →
           y ≼ ys → y ≼ (insert x ys)
≼-insert {zero}  {x} []                y≤x  dom = y≤x , lift tt
≼-insert {suc n} {x} (z :: zs ⟨ prf ⟩) y≤x  (y≤z , zsDomy) with x ≤? z
... | yes lt  = y≤x , y≤z , zsDomy
... | no ¬lt  = y≤z , ≼-insert zs y≤x zsDomy
```

Here, ¬x≤y→y≤x is a proof that $x \not\leq y$ implies $y \leq x$ in a total order. We use ≼-trans to construct the domination proof in the 'cons' case of insert.

Typical list functions may be given the precise types one usually expects when working with vectors. Vector membership, _∈_, used throughout the paper, is defined using an inductive relation with two constructors as usual, complicated only slightly by the need to quantify over explicit domination proofs:

```
data _∈_ (x : Carrier) : ∀ {n} → SortedVec n → Set (ℓ₁ ⊔ a ⊔ ℓ₂) where
  here  : ∀ {n} → (xs : SortedVec n) → ∀ prf → x ∈ (x :: xs ⟨ prf ⟩)
  there : ∀ {n} → (y : Carrier) → (ys : SortedVec n) →
                  ∀ prf → x ∈ ys → x ∈ (y :: ys ⟨ prf ⟩)
```

Using this definition, we may show by case analysis that the head of a vector is indeed the smallest element contained therein:

```
head-≤ : ∀ {m} {x} {xs : SortedVec (ℕ.suc m)} → x ∈ xs → head xs ≤ x
```

3 Sobrinho Algebras, their properties and models

Fix a carrier set S and call a binary operation *selective* when $x \bullet y = x$ or $x \bullet y = y$ for any $x, y \in S$. Intuitively, a selective binary operation denotes a 'choice' between elements. We call $\langle S, \oplus, \otimes, 0, 1 \rangle$ a 'Sobrinho Algebra' whenever:

- $\langle S, \oplus, 0 \rangle$ forms a commutative monoid,
- 1 is a left identity for multiplication, and a left- and right zero for addition,
- addition is selective, and addition absorbs multiplication.

All closure and congruence properties for the operations apply as usual. Following convention, we capture the notion of a Sobrinho Algebra as an Agda record, SobrinhoAlgebra. We use Carrier for the carrier type of a Sobrinho Algebra, corresponding to the carrier set S above, obtaining the closure properties mentioned above for 'free' as a side-effect of Agda's typing discipline, assuming that there exists a decidable setoid equivalence relation on elements of this type, _≈_. We use 1# and 0# for our two identity elements, in order to avoid clashing with Agda's in-built numeral parsing notation for natural numbers.

Models We present three inhabitants of the SobrinhoAlgebra record to demonstrate both that they exist and are not categorical (i.e. are not inhabitable by only one structure up to isomorphism). We will also use the inhabitants later in Section 5 where we provide an example execution of our algorithm.

Trivially, the axioms of a SobrinhoAlgebra are satisfied by the unit type, ⊤, defining a degenerate 'addition' and 'multiplication' operation on ⊤. Inhabiting the SobrinhoAlgebra record is henceforth straightforward.

To obtain more useful models, we first consider the natural numbers with a distinguished element, intuitively taken to be a 'point at infinity':

```
data ℕ∞ : Set where
  ↑  : ℕ → ℕ∞
  ∞  : ℕ∞
```

The constructor ↑ can be used to embed the natural numbers into ℕ∞. Define addition, multiplication, minimum and maximum functions, _+_, _*_, _⊓_, and _⊔_, respectively, so that ∞ is fixed as the largest element of ℕ∞, and the following properties of addition and multiplication hold for all m: $\infty + m \equiv \infty \equiv m + \infty$, and $\infty * m \equiv \infty \equiv m * \infty$, behaving in the 'obvious way' in all other cases.

Using these definitions we can define the *shortest path* algebra by taking the algebra's addition and multiplication functions to be _⊓_ and _+_ on ℕ∞, respectively, the unit for addition to be ∞, and the unit for multiplication to be ↑ 0. We may also define the *widest path* algebra by taking the algebra's addition and multiplication functions to be _⊓_ and _⊔_ on ℕ∞, respectively, the unit for addition to be ∞, and the unit for multiplication to be ↑ 0.

Properties Throughout this section we fix an inhabitant of CommutativeMonoid, and use Carrier, _≈_, ε, and _•_ to denote the monoid's underlying carrier type, supplied equivalence relation, neutral element, and binary operation, respectively.

We introduce the Left and Right Canonical Orders of commutative monoids and show some of their properties, and culminate in a proof that the Left and Right Canonical Orders are both total orders whenever the monoid's binary operation is selective. For reasons of brevity, we only present cases for the Left Canonical Order, leaving aside the obvious analgous proofs and definitions for the Right Canonical Order.[2] The Left and Right Canonical Orders (_≤ᴸ_ and _≤ᴿ_, respectively) are defined as follows:

$$a \leq^{\mathsf{L}} b \;=\; \exists\, \lambda\, c \to a \approx (b \bullet c) \qquad a \leq^{\mathsf{R}} b \;=\; \exists\, \lambda\, c \to b \approx (a \bullet c)$$

Recall that ∃ is defined in Agda's Standard Library as a shorthand for a dependent pair where the type of the first element (Carrier in this case) is inferred automatically. Both Left and Right Canonical Orders are reflexive:

```
≤ᴸ-reflexive : ∀ {a b} → a ≈ b → a ≤ᴸ b
≤ᴸ-reflexive {a} {b} a≈b = ε , sym (trans (proj₂ identity b) (sym a≈b))
```

Here, our existential witness is ε, the monoid's unit, and the second component of the dependent pair is a proof that given $a \approx b$, the equivalence $a \approx (b \bullet \varepsilon)$ holds. By definition this is equivalent to $a \leq^{\mathsf{L}} b$. We also have transitivity:

[2] Note, the wider algebraic routing literature variously refers to either of the two definitions we will introduce below as *the* Canonical Order; Gondran and Minoux [10, p. 18], for example, exclusively use the Right Canonical Order in their work.

```
≤ᴸ-transitive : ∀ {a b c} → a ≤ᴸ b → b ≤ᴸ c → a ≤ᴸ c
≤ᴸ-transitive {a} {b} {c} (x , a≈b•x) (y , b≈c•y) = x • y , eq
  where
    eq = begin
      a           ≈⟨ a≈b•x ⟩
      b • x       ≈⟨ •-cong b≈c•y refl ⟩
      (c • y) • x ≈⟨ assoc _ _ _ ⟩
      c • (y • x) ≈⟨ •-cong refl (comm _ _) ⟩
      c • (x • y) □
```

The proof of transitivity is slightly more involved. Using the monoid's associative and commutative laws, we show that $a \approx c \bullet (x \bullet y)$ which implies $a \trianglelefteq^{L} c$. We use the Agda Standard Library's equational reasoning constructs—`begin_`, `_≈⟨_⟩_` and `_□`—here and in the rest of the paper to structure proofs.

The Left Canonical Order is also total—that is, for any a and b, $a \trianglelefteq^{L} b$ or $b \trianglelefteq^{L} a$—whenever `_•_` is selective. We remind the reader that `_•_` is *selective* when $a \bullet b$ is equivalent to either a or b. Accordingly, our proof proceeds by a case split on the two possible results of $a \bullet b$:

```
≤ᴸ-total : Selective _•_ → Total _≤ᴸ_
≤ᴸ-total selective a b with selective a b
... | inj₁ a•b≈a = inj₁ (a , (sym (trans (comm _ _) a•b≈a)))
... | inj₂ a•b≈b = inj₂ (b , (sym a•b≈b))
```

Whenever `_•_` is selective we have that `_≤ᴸ_` is antisymmetric. Again, we proceed by a case split on the results of $a \bullet y$ and $b \bullet x$:

```
≤ᴸ-antisym : Selective _•_ → Antisymmetric _≈_ _≤ᴸ_
≤ᴸ-antisym selective {a} {b} (x , a≈b•x) (y , b≈a•y) with selective a y | selective b x
... | _          | inj₁ b•x≈b = trans a≈b•x b•x≈b
... | inj₁ a•y≈a | _          = sym (trans b≈a•y a•y≈a)
... | inj₂ a•y≈y | inj₂ b•x≈x = a≈b
  where
    a≈x = trans a≈b•x b•x≈x
    b≈y = trans b≈a•y a•y≈y
    a≈b = begin
      a       ≈⟨ a≈x ⟩
      x       ≈⟨ sym b•x≈x ⟩
      b • x   ≈⟨ •-cong b≈y refl ⟩
      y • x   ≈⟨ comm _ _ ⟩
      x • y   ≈⟨ •-cong (sym a≈x) refl ⟩
      a • y   ≈⟨ a•y≈y ⟩
      y       ≈⟨ sym b≈y ⟩
      b □
```

We therefore have that the Left Canonical Order on a selective commutative monoid is a total order. We next show that the Left Canonical Order of a Sobrinho Algebra's addition operator is a decidable total order. From this point on we fix •-selective, a proof that the monoid's binary operation is selective, and

$\_\stackrel{?}{=}\_$, a proof that the monoid's equivalence relation is decidable. Any Sobrinho Algebra possesses both of these properties, so assuming them here is 'safe' for our purposes. Further, as selectivity implies idempotence, we also have •-idempotent, a proof that the monoid's binary operation is idempotent whenever it is selective.

Before demonstrating decidability, we require two auxiliary lemmas. The first, •-selective′, is a direct consequence of selectivity, stating that, given $a \approx b \bullet c$, one of $a \approx b$ or $a \approx c$ must hold:

```
•-selective′ : ∀ {a b c} → a ≈ b • c → a ≈ b ⊎ a ≈ c
•-selective′ {a} {b} {c} a≈b•c with •-selective b c
... | inj₁ b•c≈b = inj₁ (trans a≈b•c b•c≈b)
... | inj₂ b•c≈c = inj₂ (trans a≈b•c b•c≈c)
```

The second, $\not\approx\Rightarrow\not\trianglelefteq^{\mathsf{L}}$, states that if $b \bullet a \approx a$ does *not* hold, then $a \trianglelefteq^{\mathsf{L}} b$ also does not hold, neither:

```
≉⇒⋬ᴸ : ∀ {a b} → ¬ b • a ≈ a → ¬ a ⪯ᴸ b
≉⇒⋬ᴸ {a} {b} ¬b•a≈a (x , a≈b•x) with •-selective′ a≈b•x
... | inj₁ a≈b = ¬b•a≈a (trans (•-cong (sym a≈b) refl) (•-idempotent a))
... | inj₂ a≈x = ¬b•a≈a (trans (•-cong refl a≈x) (sym a≈b•x))
```

Using these we may now prove decidability of the Left Canonical Order, proceeding by splitting on whether $b \bullet a$ is equivalent to a, or not, with the interesting case being the second, where we make use of both of our auxiliary lemmas above:

```
_⪯ᴸ?_ : Decidable _⪯ᴸ_
a ⪯ᴸ? b with (b • a) ≟ a
... | yes b•a≈a  = yes (a , sym b•a≈a)
... | no ¬b•a≈a  = no (≉⇒⋬ᴸ ¬b•a≈a)
```

We therefore have that the Left and Right Canonical Orders form a decidable total order in an arbitrary commutative monoid whenever the monoid's binary operation is selective and its equivalence relation is decidable. As Sobrinho Algebras are a superstructure of commutative selective monoids with a decidable equivalence relation, we have that the Left and Right Canonical Orders in an arbitrary Sobrinho Algebra are decidable total orders.

4 Dijkstra's Algorithm and its correctness

Our purely functional implementation in Agda consists of nine mutually recursive definitions, the most important of which are order, estimate, seen and queue. Throughout this section we use i to denote the start node of the search, and use the suggestive name Weight to refer to the carrier set of our Sobrinho Algebra.

At each *step* of the algorithm graph nodes are totally ordered. This total order is constructed using the order function, which is parameterised by the *step* of the algorithm:

```
order : (step : ℕ) → {s≤n : step ≤ n} → DecTotalOrder _ _ _
order step {s≤n} = estimateOrder $ estimate step {s≤n}
```

The function estimateOrder lifts a mapping from nodes to weights into a decidable total order on nodes. The function estimate provides an estimate of the distance from the start node i to every other node in the graph:

```
estimate : (step : ℕ) → {s≤n : step ≤ n} → Fin (suc n) → Weight
estimate zero                 j = A[ i , j ]
estimate (suc step) {step≤n}  j = r j + r q * A[ q , j ]
  where
    q  = Sorted.head (order step {≤-step′ step≤n}) (queue step {step≤n})
    r  = estimate step {≤-step′ step≤n}
```

The base case for the estimate function is a lookup in the adjacency matrix of the graph. Note that since the addition operation, _+_, of a Sobrinho Algebra is selective, the inductive case of estimate encodes a *choice* between r j and r q * A[q , j]. The former is simply the previous distance estimate to j, whilst the latter represents the option of going from the start node to q via the best known path from the previous step, and then directly from q to j (where q is the head of the priority queue of nodes that have not yet been visited).

The set of visited nodes at a given *step* is computed by the function seen:

```
seen : (step : ℕ) → {s≤n : step ≤ n} → Subset (suc n)
seen zero                 = ⟦ i ⟧
seen (suc step) {step≤n}  = seen step {≤-step′ step≤n} ∪
  ⟦ Sorted.head (order step {≤-step′ step≤n}) (queue step {step≤n}) ⟧
```

Here, ⟦ i ⟧ is a singleton set containing only the start node, i. The inductive case of seen unions together all visited nodes from previous steps of the algorithm with the next node to be visited. Once a node has been visited, its distance estimate stays constant and is optimal—this important invariant will be proved and used later in the proof of correctness of the algorithm in the remainder of the paper.

The following is an auxiliary definition needed to define the function queue, computing the queue of nodes that have not yet been visited by the algorithm:

```
queue′ : (step : ℕ) {s≤n : step ≤ n} → Sorted.Vec _ (size $ ∁ $ seen step {s≤n})
queue′ step {s≤n} = Sorted.fromVec (order step {s≤n}) $ toVec $ ∁ $ seen step
```

Here the function ∁ is setwise complement, with the expression ∁ $ seen *step* {s≤n} corresponding to the set of *unseen* graph nodes. The function queue′ is a direct definition of the priority queue of unvisited nodes at a given step of the algorithm: we take the complement set of the set of nodes that have been visited thus far and order them using our total order, order, at the given algorithm step. Whilst straightforward to understand, unfortunately, this definition is awkward to use in practice due to a problem with the type of queue′: the priority queue's only use is to provide the node with the smallest estimate that has not yet been visited, which is always at the head of the queue, but to extract the head of a queue, its type must guarantee that it contains at least one element. This fact is expressed by mandating that the length index of the vector whose head is being

examined must be of the form suc n for some n. Therefore, in order to provide a queue with a more usable length index, we prove the following lemma which we will use to 'massage' the type of queue′ into something more amenable:

```
queue-size : (step : ℕ) → {s≤n : suc step ≤ n} →
             size (C $ seen step {≤-step′ s≤n}) ≡ suc (n ∸ suc step)
```

Using queue′ and queue-size, we can then give the following more useful definition of the priority queue of previously unvisited nodes, with a suc in head position in the vector's length index, with the function queue:

```
queue : (step : ℕ) → {s<n : suc step ≤ n} → Sorted.Vec _ (suc (n ∸ (suc step)))
```

Correctness We now prove that our algorithm computes a Right Local Solution to the matrix fixpoint equation described in the Introduction. Henceforth, we fix *alg*, an arbitrary inhabitant of SobrinhoAlgebra, and *adj*, an arbitrary $n \times n$ adjacency matrix describing a graph whose coefficients are taken from the Carrier type of *alg*. Ultimately we aim to show the following statement of correctness:

```
correct : ∀ j → RLS n {≤-refl} j
```
(1)

That is, our algorithm is correct if, after n iterations of the algorithm on the adjacency matrix *adj*, a Right Local Solution to the matrix fixpoint equation has been found. Above, we make use of RLS, a predicate over graph nodes and steps of the algorithm, which captures the notion of a Right Local Solution. An estimate $r_j^{(n)}$ for node j at step n is a *Right Local Solution* iff the equation

$$r_j^{(n)} \approx I_{i,j} + \bigoplus_{k \in V} r_k^{(n)} * A_{k,j}$$

holds, where V is the set of all nodes in the graph described by *adj* (expressed as $\top$ in Agda). Concretely, in Agda we define this as follows:

```
RLS step {s≤n} j =
  let r = estimate step {s≤n} in
    r j ≈ I[ i , j ] + (⨁[ k ← ⊤ ] r k * A[ k , j ])
```

To prove Property 1 above, we define an auxiliary, weaker predicate, capturing the notion of a *Partial Right Local Solution*. In particular, the estimate $r_j^{(n)}$ for node j at step n is a Partial Right Local Solution if and only if the equation

$$r_j^{(n)} \approx I_{i,j} + \bigoplus_{k \in S_n} r_k^{(n)} * A_{k,j}$$

holds (S_n is the set of visited nodes at step n), expressing this in Agda as:

```
pRLS step {s≤n} j =
  let r = estimate step {s≤n} in
    r j ≈ I[ i , j ] + (⨁[ k ← seen step {s≤n} ] r k * A[ k , j ])
```

This definition of a Partial Right Local Solution, as captured by pRLS, is central to our proof, as we will prove by induction on the number of algorithm steps taken that the predicate pRLS holds for any *step* and j. We then show that pRLS n j, and the fact that at step n the algorithm has visited all graph nodes, implies RLS n j. Correctness will follow. The following lemma

```
pcorrect : (step : ℕ) {s≤n : step N≤ n} → ∀ j → pRLS step {s≤n} j
```

implements the central argument of our correctness proof, as previously described. We step through its proof, which proceeds by induction on *step*, the number of steps of the algorithm so far completed.

Base case. In the base case (*step* = zero), we case split on whether the node j is equal to the start node, i, using the following shorthands to conserve space:

- r = estimate zero {z≤n} : Fin (suc n) → Weight. For any node j, r j stands for the initial distance estimate from the start node to j.
- Given that ¬ i ≡ j, I[i,j]≡0 : lookup i j (tabulate (diagonal 0# 1#)) ≡ 0# shows that looking up an element of the identity matrix of the Sobrinho Algebra that is not on the diagonal is propositionally equal to the algebra's multiplicative unit.
- fold : ⨁[k → r k * A[k , j]] ⟦ i ⟧ ≈ r i * A[i , j] proves a particular case of the fact that a fold over a singleton set is just the inner expression of the fold with the only element of the singleton set as the bound variable.

First, assume that $i = j$. By definition, we have estimate zero j is A[i , j], which equals A[i , i], by assumption. This, in turn, is equivalent to 1# by the adjacency matrix diagonal property. The result follows by the identity matrix' diagonal property and the fact that 1# is a zero element for _+_:

```
pcorrect zero {s≤n} j with i FP.≟ j
... | yes i≡j = begin
  r j             ≡⟨⟩
  A[ i , j ]      ≡⟨ P.cong₂ A[_,_] (P.refl {x = i}) j≡i ⟩
  A[ i , i ]      ≈⟨ Adj.diag adj i ⟩
  1#              ≈⟨ sym (proj₁ +-zero _) ⟩
  1#      + _     ≈⟨ +-cong (sym (Adj.diag I j)) refl ⟩
  I[ j , j ] + _  ≡⟨ P.cong₂ _+_ (P.cong₂ I[_,_] j≡i (P.refl {x = j})) P.refl ⟩
  I[ i , j ] + _ □
```

Next, assume that $i \neq j$. We expand the definition of estimate and use the identity property of _+_ to show that estimate zero j is equivalent to 0# + A[i , j]. The left-hand side (0#) is equal to I[i , j] by the definition of the identity matrix and the assumption $i \neq j$. Further, the right-hand side (A[i , j]) can be massaged into $\bigoplus_{k \in \{i\}} r_k * A_{k,j}$ using the left-identity property of * and the adjacency matrix diagonal property, as follows:

```
... | no ¬i≡j = begin
  r j                          ≡⟨⟩
  A[ i , j ]                   ≈⟨ sym (proj₁ +-identity _) ⟩
  0#      + A[ i , j ]         ≡⟨ P.cong₂ _+_ (P.sym I[i,j]≡0) P.refl ⟩
  I[ i , j ]  + A[ i , j ]     ≈⟨ +-cong refl (sym (*-identityˡ _)) ⟩
  I[ i , j ]  + 1# * A[ i , j ] ≈⟨ +-cong refl (*-cong (sym (Adj.diag adj i)) refl) ⟩
  I[ i , j ]  + r i * A[ i , j ] ≈⟨ +-cong refl (sym fold) ⟩
  I[ i , j ]  + (⨁[ k ← ⟦ i ⟧ ] r k * A[ k , j ]) □
```

Induction step. Next, we have the induction step case (*step* = suc *step*) of the partial correctness proof, using the following shorthands to conserve space:

- r = estimate *step* {≤-step′ *s*≤*n*} : Fin (suc *n*) → Weight, so r *j* stands for the distance estimate from the start node to node *j* at step '*step*'.
- r′ = estimate (suc *step*) {*s*≤*n*} : Fin (suc *n*) → Weight, so r *j* stands for the distance estimate to node *j* at step 'suc *step*'.
- q = Sorted.head _ (queue *step* {*s*≤*n*}) : Fin (suc *n*), is the node whose current estimated distance from the start node is the smallest of all unvisited nodes.
- f = λ *k* → *r k* * A[*k* , *j*] : Fin (suc *n*) → Weight.
- f′ = λ *k* → *r*′ *k* * A[*k* , *j*] : Fin (suc *n*) → Weight.
- vs = seen *step* {≤-step′ *s*≤*n*} : Subset (suc *n*), the list of nodes that have been visited at step *step*.
- fold = fold-cong f f′ *vs* (λ *k k*∈*vs* → lemma *k k*∈*vs*), with type ⨁[*k* ← *vs*] f *k* ≈ ⨁[*k* ← *vs*] f′ *k* is a special case of the theorem that given f *i* ≈ f′ *i* for all *i* ∈ *xs* it follows that the fold over *xs* using f is equivalent to the fold over *xs* using f′ as the fold expresssion (see Section 2).

Note, in the definition of fold, we make use of a small lemma, with type ∀ *k* → *k* ∈ *vs* → f *k* ≈ f′ *k*, which shows that f and f′ agree on all visited graph vertices. Below we present the formal proof of the inductive step case, using Agda's equational reasoning mechanism, with explicative comments describing each equational reasoning step to aid the reader:

```
pcorrect (suc step) {s≤n} j = begin
  r′ j
  {- Definition of 'estimate' -}
    ≡⟨⟩
  r j + r q * A[ q , j ]
  {- Induction Hypothesis -}
    ≈⟨ +-cong (pcorrect step {≤-step′ s≤n} j) refl ⟩
  (I[ i , j ] + (⨁[ k ← vs ] r k * A[ k , j ])) + r q * A[ q , j ]
  {- Associativity of _+_ -}
    ≈⟨ +-assoc _ _ _ ⟩
  I[ i , j ] + ((⨁[ k ← vs ] r k * A[ k , j ]) + r q * A[ q , j ])
  {- Absorptivity -}
    ≈⟨ +-cong refl (+-cong fold (*-cong (sym (+-absorbs-* _ _)) refl)) ⟩
  I[ i , j ] + ((⨁[ k ← vs ] r′ k * A[ k , j ]) + r′ q * A[ q , j ])
```

```
  {- Singleton Fold -}
    ≈⟨ +-cong refl (+-cong refl (sym (fold-⟦i⟧ f′ q))) ⟩
  I[ i , j ] + ((⨁[ k ← vs ] r′ k * A[ k , j ]) + (⨁[ k ← ⟦ q ⟧ ] r′ k * A[ k , j ]))
  {- Commutativity and Associativity of _+_ -}
    ≈⟨ +-cong refl (sym (fold-∪ +-idempotent f′ (seen step) ⟦ q ⟧)) ⟩
  I[ i , j ] + (⨁[ k ← vs ∪ ⟦ q ⟧ ] r′ k * A[ k , j ])
  {- Definition of ‘seen‘ -}
    ≡⟨⟩
  I[ i , j ] + (⨁[ k ← seen (suc step) {s≤n} ] r′ k * A[ k , j ]) □
```

This completes the proof of pcorrect. Now, after n iterations all n of the graph's nodes have been visited, so seen $n \equiv \top$. We omit the straightforward proof of this fact, which we refer to as lemma in the following proof that a Partial Right Local Solution after n steps is the same as a Right Local Solution:

```
correct : ∀ j → RLS n {≤-refl} j
correct j = begin
  r j
    ≈⟨ pcorrect n j ⟩
  I[ i , j ] + (⨁[ k ← seen n {≤-refl} ] r k * A[ k , j ])
    ≡⟨ P.cong₂ _+_ P.refl lemma ⟩
  I[ i , j ] + (⨁[ k ← ⊤ ] r k * A[ k , j ]) □
```

Above, we have omitted the definition of lemma, which proves that seen n can be replaced by $\top$ (the set containing all naturals up to n) as the index set expression of the fold: lemma : ⨁[$k \leftarrow$ seen n] r k * A[k , j] $\approx$ ⨁[$k \leftarrow \top$] r k * A[k , j] using the fact that after n steps, all nodes have been visited, that is, seen $n \equiv \top$. With this, we have Property 1, and have the correctness proof of the algorithm.

5 Example

We demonstrate our algorithm in action by executing it within Agda, showing that a computed Right Local Solution matches a precomputed matrix of weights of shortest paths. All matrix coefficients are taken from the shortest path algebra—the algebra over $\mathbb{N}\infty$ with _⊓_ as addition and _+_ as multiplication—described in Section 3. We will suggestively refer to the carrier of this algebra as Weight. The two matrices are:

$$\text{Adjacency} = \begin{pmatrix} 0 & 4 & 1 \\ \infty & 0 & 2 \\ 1 & 2 & 0 \end{pmatrix} \qquad \text{Expected} = \begin{pmatrix} 0 & 3 & 1 \\ 3 & 0 & 2 \\ 1 & 2 & 0 \end{pmatrix}$$

The fact that the right-hand matrix is correct can easily be established by hand. We implement both matrices using our matrix library, calling the first matrix adj and the second rls-expected. For convenience we define the following function rls that computes the entire Right Local Solution for a given adjacency matrix:

```
rls : ∀ {n} → Adj (suc n) → Matrix Weight (suc n) (suc n)
rls adj = M.tabulate (λ i → let open Algo alg i adj in estimate _ {≤-refl})
```

The computed Right Local Solution and the expected result are pointwise equal, with the execution time (within Agda) being on the order of seconds:

```
rls-correct : Pointwise _≡_ (rls adj) rls-expected
rls-correct = λ r c → refl
```

6 Conclusions

In this paper we have presented a purely functional implementation of a generalised shortest path algorithm, and proved it correct using an algebraic method. We have made extensive use of dependent types, and some of Agda's more advanced features, such as induction-recursion, to structure the implementation and proof. All implementation files, and supporting documentation, are available anonymously from a public `git` repository.[3] Our implementation consists of approximately 2,400 lines of Agda, and was developed with Agda 2.4.2.1 and 2.4.2.2 and Standard Library version 0.9.

Related work Despite the algorithm's notability, Chen seems to have been the first to verify the correctness of Dijkstra's Algorithm in any proof assistant, producing a Mizar implementation in 2003 [4]. Later, Moore and Zhang verified Dijkstra's Algorithm in ACL2 [15]. Gordon, Hurd, and Slind verified Dijkstra's reachability algorithm in HOL4 as part of a wider formalisation of Accellera [11]. Fleury verified Floyd's all-pairs shortest path algorithm in Coq [9], and in unpublished work, Paulin and Fillîatre later verified Floyd's algorithm again in imperative form with the aid of an additional tool, also in Coq. Nordhoff and Lammich verified Dijkstra's algorithm in Isabelle/HOL as a showcase of the Isabelle refinement and collections frameworks [16]. Lammich also later implemented and verified an imperative version of Dijkstra's Algorithm in Isabelle/HOL [13]. These prior formalisations take a classical approach to shortest path algorithms and their proofs of correctness in contrast to our work presented in this paper. We believe that we are the first to present a verified, axiom-free implementation and proof of correctness of a shortest path algorithm employing the algebraic method.

The idea that Dijkstra's algorithm can be generalised to an algorithm that solves a matrix fixpoint equation was first explored by Sobrinho [18], with Mohri later presenting a general semiring framework for shortest path algorithms [14] where the underlying semiring and queuing discipline used by the algorithm are abstracted over. Dijkstra's algorithm, and other shortest path algorithms, were recovered as special cases. Griffin and Sobrinho explored the solutions found by the generalised algorithm whenever distributivity is not assumed [19], with Griffin later producing an unpublished, incomplete formalisation in Coq of some of these ideas, with 10 axiomatised statements [12]. Our work builds on these latter ideas, but goes beyond it in several ways: we fully specify the properties of

[3] https://bitbucket.org/curiousleo/path-algebra

the algebraic structure assumed obtaining the notion of a 'Sobrinho Algebra', explore properties of these algebras and present some of their models, give a concrete implementation of the algorithm, and mechanically verify its correctness with a clean-slate axiom-free proof.

Acknowledgements We thank Timothy G. Griffin for helpful comments when preparing this paper, including the suggestion of the name 'Sobrinho Algebra'. The second author is employed on the EPSRC grant EP/K008528 (REMS).

References

1. Asperti, A., Ricciotti, W., Coen, C.S., Tassi, E.: The Matita interactive theorem prover. In: 23rd International Conference on Automated Deduction (CADE). pp. 64–69 (2011)
2. Bertot, Y.: A Short Presentation of Coq. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) Theorem Proving in Higher Order Logics, pp. 12–16. No. 5170 in Lecture Notes in Computer Science, Springer Berlin Heidelberg (2008)
3. Bertot, Y., Gonthier, G., Biha, S.O., Pasca, I.: Canonical Big Operators. In: Mohamed, O.A., Muñoz, C., Tahar, S. (eds.) Theorem Proving in Higher Order Logics, pp. 86–101. Springer, Berlin; Heidelberg (Jan 2008)
4. Chen, J.C.: Dijkstra's shortest path algorithm. Journal of Formalized Mathematics 15 (2003)
5. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to algorithms. MIT Press, 2 edn. (2001)
6. Danielsson, N.A., Norell, U.: Parsing Mixfix Operators. In: Scholz, S.B., Chitil, O. (eds.) Implementation and Application of Functional Languages, pp. 80–99. No. 5836 in Lecture Notes in Computer Science, Springer, Berlin; Heidelberg (2011)
7. Dijkstra, E.W.: A note on two problems in connexion with graphs. Numerische Mathematik 1, 269–271 (1959)
8. Dybjer, P.: A general formulation of simultaneous inductive-recursive definitions in type theory. Journal of Symbolic Logic 65(02), 525–549 (2000)
9. Fleury, E.: Implantation des algorithmes de Floyd et de Dijkstra dans le Calcul des Constructions (1990), rapport de Stage
10. Gondran, M.: Graphs, Dioids and Semirings: New Models and Algorithms. No. 41 in Operations Research/Computer Science Interfaces, Springer, New York (2008)
11. Gordon, M.J.C., Hurd, J., Slind, K.: Executing the formal semantics of the Accellera Property Specification Language by mechanised theorem proving. In: Proceedings of the 12th IFIP WG 10.5 Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME). pp. 200–215 (2003)
12. Griffin, T.G.: A partial Coq formalisation of results contained in 'Routing in Equilibrium' (2012), http://www.cl.cam.ac.uk/~tgg22/metarouting/rie-1.0.v
13. Lammich, P.: Refinement to Imperative HOL. In: Proceedings of the 6th International Conference on Interactive Theorem Proving (ITP). pp. 253–269 (2015)
14. Mohri, M.: Semiring frameworks and algorithms for shortest-distance problems. Journal of Automata, Languages and Combinatorics 7, 321–350 (2002)
15. Moore, J.S., Zhang, Q.: Proof pearl: Dijkstra's shortest path algorithm verified with ACL2. In: Proceedings of the 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLS). pp. 373–384 (2005)

16. Nordhoff, B., Lammich, P.: Dijkstra's shortest path algorithm. In: The Archive of Formal Proofs (2012)
17. Norell, U.: Dependently Typed Programming in Agda. In: Koopman, P., Plasmeijer, R., Swierstra, D. (eds.) Advanced Functional Programming, pp. 230–266. No. 5832 in Lecture Notes in Computer Science, Springer, Berlin; Heidelberg (2009)
18. Sobrinho, J.L.: Algebra and algorithms for QoS path computation and hop-by-hop routing in the internet. In: IEEE INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. vol. 2, pp. 727–735 vol.2
19. Sobrinho, J.L., Griffin, T.G.: Routing in Equilibrium. In: Proceedings of the 19th International Symposium on Mathematical Theory of Networks and Systems (2010)