Shortest Path Algorithms Verification with Idris

Yazhe Feng

February 21, 2019

1 Introduction

Shortest path problems deal with finding the path with minimum distance value between two nodes in a given graph. One variation of shortest path problem is single-source shortest path problem, which focuses on finding the path with minimum distance value from one source to all other vertices within the graph. Dijkstra's and Bellman-Ford are the most renowned single-source shortest path algorithms, and are implemented by software concerning various fields in real-life applications, such as finding the shortest path in road map, or routing path with minimum cost in networks.

Given the importance of Dijkstra's and Bellman-Ford in real-life applications, we are interested in verifying the implementation of both algorithms. We start with providing a concrete implementation for both algorithms. Then with respect to our programs, we will define functions describing properties that should hold for any correct implementation of Dijkstra's and Bellman-Ford, for instance returning the minimum distance value from the source to each node in the graph. Having these functions type checked will then ensure the correctness of our algorithm implementation, and that any program with problematic implementation will fail to compile at the type checking level. Our work will be done with the Idris functional programming language, which embraces powerful tools and features that are siginificantly helpful in program verification.

Existing work on verifying Dijkstra's algorithm is relatively limited, and few resources are found for the verification of Bellman-Ford algorithm. Robin Mange and Jonathan Kuhn [3] demonstrates an implementation that verifies Dijkstra's algorithm with the Jahob verification system in their report on efficient proving of Java implementations[6]. Although few resource has been found on the concrete implementation of this work, the report illustrates that as Jahob allows programmers to provide specification of their function's behaviors in high-level logic(HOL), program verification can be reduced to the problem of the validity of HOL formulas.

Klasen et. al. [1] from the University of Koblenz and Landau present a concrete implementation of Dijkstra's algorithm in Java and its verification with the KeY system. The verification process involves specifying the behavior of each function with preconditions, postconditions, and invariants, and the KeY system checks a function as correct with respect to its specifications if the postconditions hold after execution. Jean-Christophe Filliâtre[4], a senior researcher from the National Center for Scientific Research(CNRS), offers an implementation of Dijkstra's algorithm along with its verification in Why3, a deductive program verification platform that relies on external theorem provers[5]. Both verifications above are largely dependent on theorem proving systems. Unlike Filliâtre and Klasen et. al., our work relies on a significantly smaller trusted code base, indicating that considerable amount of proofs will be presented explicitly in our implementation rather than provided by external theorem provers.

Contribution

(To be finished.)

The structure of the paper is as follows. Section 2 describes the significance and value of algorithm verification, and reasons of choosing Idris as the language for verifying programs. Section 3 provides some background on Dijkstra's and Bellman-Ford algorithms, follows up by briefly introduction on the Idris functional programming language. Section 4 includes an overview of our verification program, including definition of key concepts, assumptions made by our program, and details on the pseudocode and theoretical proof of Dijkstra's and Bellman-Ford, which serves as important guideline in implementation our verification program. Section 5 covers more details of our verification program, including function type signatures and code of the proof for key lemmas. Section 6 is discussion of our work. Section 7 presents and compares related work, and section 8 gives a breif conclusion.

2 Motivation

Verifying the correctness of programs is important, however in most real-life applications, the correctness of software is never verified directly, rather, it relies on the correctness of the algorithms it implements. This raises an issue concerning the gap between the expected and actual behavior of programs, that theoretical proof of algorithms can never validate the actual behavior of programs. The significance and value of verification, therefore, lies on the fact that it allows us to verify programs themselves rather than the algorithms behind them.

Dijkstra's and Bellman-Ford algorithms are two of the most renowned and widely-applied shortest path algorithms, however existing resource on verifying both algorithms are relatively limited. In this thesis, we offer verifications for the implementations of both algorithm. In additional, we aim to present verification as a programming issue. We want to show that with certain programming languages, verifying the correctness of programs can be achieved with type checking, that if the program's correctness is not guaranteed, then our verification program will fail to be type checked.

Based on the above motivations, the Idris programming language is chosen over other verification tools and proof management systems. Idris is a functional programming language with dependent types, which allows programmers to provide more specification on function's behaviors in its type signature. As we plan to achieve verification with type checking, this feature of Idris can be significantly helpful as often times it is important to establish tight connection between functions and its input data in a verification program. In addition, Idris's compiler-supported interactive editing feature provides precise description of functions' behaviors according to their types, allowing programmer to use types as guidance for writing program, which offers considerable assistance during our implementation. Section 3 covers more backgrounds on the Idris programming language.

3 Background

3.1 Introduction of Idris

Idris is a general-purpose functional programming language with dependent types. Many aspects of Idris is influenced by Haskell and ML. Features of Idris include but not limit to dependent types, `with` rule, `case` expressions, lambda binding, as well as interactive editing.

Data Declaration

As an example of data declaraction in Idris, below shows the definition of natural numbers in Idris standard library:

```
  -- natural number can be either zero(Z) or plus one of another
natural number (S Nat)
  data Nat = Z | S Nat
```

Another syntax similar to that of GADT in Haskell for data declaration is also allowed:

```
-- declaration of List data type in Idris standard library
data List : (elem : Type) -> Type where
  Nil : List elem
  (::) : (x : elem) -> (xs : List elem) -> List elem
```

Dependent Types

Dependent types are types that depend on elements of other types[2]. It allows programmers to specify certain properties of data types explicitly in their type signature. Consider the following definition of a vector data type, where `len` specifies the length of the vector, and `elem` gives the element type:

```
-- declaration of Vect data type in Idris standard library
data Vect : (len : Nat) -> (elem : Type) -> Type where
  Nil  : Vect Z elem
  (::) : (x : elem) ->
         (xs : Vect len elem) ->
         Vect (S len) elem
```

The type `Vect len elem` is dependent on the value of type variables `len` and `elem`, specifying that `Vect` is a vector of length `len` containing element of type `elem`. With dependent types, programmers can ensure the behaviors of functions through their type signatures by defining more precise types. Consider the function `concat` below that concantenates two vectors. Given two input vectors V_1 and V_2, the output value of `concat` should be a vector of length $|V_1| + |V_2|$.

```
concat : Vect n Nat -> Vect m Nat -> Vect (n+m) Nat
```

The type siganture of `concat` establishes that the resulting vector of concatenating two vectors of length n and m must be of length $(n + m)$, otherwise `concat` will fail to type check.

Pattern Matching and Totality Checking

Pattern matching is the process of matching values against specific patterns. In Idris, functions are implemented by pattern matching on possible values of inputs. Continuing with the above example of `concat` function that concatenates two vectors, to define `concat`, we need to provide definitions on all possible values of `Vect`, which can either be `Nil`, i.e., a vector of length zero, or a non-empty vector of the pattern `(x :: xs)`.

```
concat : Vect n Nat -> Vect m Nat -> Vect (n+m) Nat
concat Nil v2 = v2
concat v1 Nil = v1
concat (x :: xs) v2 = x :: concat xs v2
```

Functions defined for all possible values of input are total functions, and are guaranteed to produce a result in finite time given well-typed inputs. Partial functions are not total, and hence might crash for some inputs. To secure the termination of programs, every function definition in Idris are checked for totality after type checking. Specifically, Idris decides whether a function terminates based on two aspects: first, function must be defined for all possible inputs; and second, if a function definition includes a recursive call, then there must be an argument that strictly decreases over each recursion, and converges towards a base case. An error or warning will be given for any function that fails totality checking.

3.2 Dijkstra's and Bellman-Ford algorithms

Dijkstra's Algorithm

Dijkstra's algorithm is a greedy algorithm that finds the shortest path from a given source to all other nodes in a directed graph with weighted edges. It was first introduced in 1959 by Edsger Wybe Dijkstra, and it is widely applied in many real-life applications, including shortest path finding in road map, or Internet routing protocols such as the Open Shortest Path First protocol.

Dijkstra's algorithm takes in a directed graph with non-negative edge weights, and computes the shortest path distance from one single source node to all other reachable nodes in the graph. The algorithm maintains a list of unexplored nodes and their distance values to the source node. Initially, the list of unexplored nodes contains all nodes in the input graph, and the distance value of all node are set as infinity except for the source node itself, which is set to zero. The algorithm extracts the node v with minimum distance value from the unexplored list during each iteration, and for each neighbor v' of v, if the path from source to v' via v contributes a smaller distance value, then the distance value of v' is updated.

Bellman-Ford Algorithm

4 High-Level Contribution

4.1 Dijkstra's Algorithms

We implemented Dijkstra's algorithm with the Idris functional programming languages. Below presents the pseudocode of our implementation.

4.1.1 Pseudocode

Given input graph g and source node s with types:

g : Graph gsize weight
s : Node gsize

We denote (u, v) as an edge from node u to v, $weight(u, v)$ as the weight of edge (u, v). We define $unexplored$ as the list of unexplored nodes, and $dist$ as the list storing distance from s to each node $n \in g$

(initially $unexplored$ contains all nodes in graph g)
$unexplored : List(Node\ \ gsize)$
$unexplored = \{v : v \in g\}$

(node value is used to index $dist$, initially distance of all nodes are infinity except the source node)
$dist : List\ \ weight$
$dist[s] = 0, dist[a] = infinity, \forall a \in g, a \neq s$

The Dijkstra's Algorithm runs as follows:

```
while (unexplored is not Nil)      {
     (At the $k^{th}$ iteration of the while loop)
     choose $u \in unexplored$ s.t.$\forall u' \in unexplored, dist[u] \leq dist[u']$
     let $unexplored'$ be the list after removing $u$ from $unexplored$
     for($\forall v \in g$ s.t.$(u, v) \in g$) {
          (At the $p^{th}$ iteration of this for loop)
          if($dist[u] + weight(u, v) < dist[v]$) {
               let $dist' = dist$ with $dist'[v] = dist[u] + weight(u, v)$
          }
          input the new $dist'$ to the $(p + 1)^{th}$ iteration of the for loop
     }
     input the new $unexplored'$ and $dist'$ to the $k^{th}$ iteration of the while
loop
}
```

4.1.2 Assumption

1. Weight of edges are positive

2. Distance value can only be zero, infinity, or summation of edge weights

3. All nodes n and edge e are valid: $n, e \in g$

4.1.3 Correctness Proof

We first provide definitions of key concepts used in our proof.

Definition 4.1. Path
(We adopt the definition of $path$ presented in the `Discrete Mathematics with Applications` book by `SUSANNA S. EPP`.)

A path from node v to w is a finite alternating sequence of adjacent vertices and edges of G, which does not contain any repeated edge or vertex. A path from v to w has the form:

$$ve_0v_0e_1v_2....v_{n-1}e_nw$$

where e_i is an edge in g with endpoints v_{i-1}, v_i. We denote the set of paths from v to w as $path(v, w)$.

Definition 4.2. Length of Path
The length of a path $p = ve_0v_0e_1v_2....v_{n-1}e_nw$ is the sum of the weights of all edges in p. We write:

$$length(p) = \sum weight(e_i), \forall e_i \in p.$$

Definition 4.3. Shortest Path
Denote $\Delta(s, v)$ as the shortest path from s to v, and $\delta(v)$ as the length of $\Delta(s, v)$. $\Delta(s, v)$ must fulfills:

$$\Delta(s, v) \in path(s, v)$$
and
$$\forall p' \in path(s, v),\ \delta(v) = length(\Delta(s, v)) \leq length(p')$$

4.1.4 Proof of Correctness

Proof of Termination
The inner for loop is guaranteed to terminate as the algorithm goes through each adjacent node exactly once. As the size of list `unexplored` decreases by one during each iteration of the while loop, the algorithm is guaranteed to terminate.

Proof of Correctness
Given graph g and source node s, $dist$ stores the distance value from s to all nodes in g calculated by the Dijkstra's algorithm, $dist[v]$ gives the corresponding distance value of v from s. Denote $explored$ as the list of nodes in g but not in $unexplored$, i.e., $explored$ stored all nodes whose neighbors have been updated by the algorithm, and $dist_k[v]$ as the value of $dist[v]$ during the k^{th} iteration of the algorithm.

Lemma (1). During the n^{th} iteration of the algorithm for $n \geq 1$, forall node $v \in explored$, we have:

1. $\delta(v) \leq \delta(v'), \forall v' \in unexplored.$
2. $dist_n[v] = \delta(v)$

Proof. We will prove this by inducting on the number of iterations.

Let P(n) be: during the n^{th} iteration of the algroithm for $n \geq 1$, forall node $v \in explored$:

(1) $\delta(v) \leq \delta(v'), \forall v' \in unexplored$; and (2) $dist_n[v] = \delta(v)$.

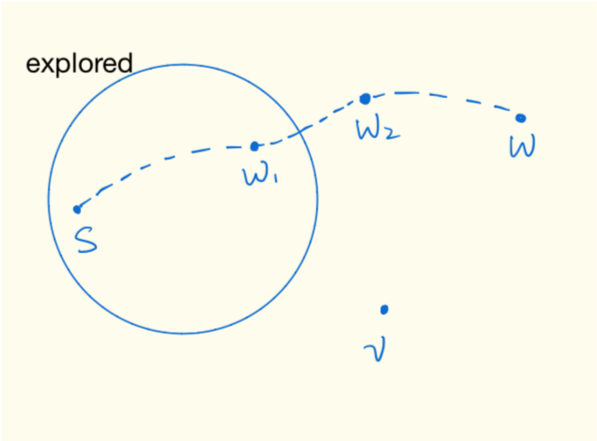
Base Case: We shall show P(1) holds
Based on the algorithm, during the first iteration, the node with minimum distance value is the source node s with $dist_1[s] = 0$. Hence during the first iteration, only s is removed from $unexplored$ and added to $explored$. Since all edge weights are positive, then the shortest distance value from s to s is indeed 0, hence $dist_1[s] = 0 = \delta(s)$ and $\delta(s) \leq \delta(v')$, $\forall v' \in unexplored$. P(1) holds.

Inductive Hypothesis: Suppose P(i) is true for all $1 < i \leq k$. That is, during the i^{th} iteration forall $1 < i \leq k$, forall node $v \in explored$: (1) $\delta(v) \leq \delta(v')$, $\forall v' \in unexplored$; and (2) $dist_i[v] = \delta(v)$.

Inductive Step: We shall show P(k+1) holds.
Suppose v is the node added into $explored$ during the $(k+1)^{th}$ iteration. We need to show (1) $\delta(v) \leq \delta(v')$, $\forall v' \in unexplored$, and (2) $dist_{k+1}[v] = \delta(v)$.

1. $\delta(v) \leq \delta(v'), \forall v' \in unexplored, v' \neq v$

   We will prove (1) by contradiction. Suppose there exists $w \in unexplored$, such that $\delta(v) > \delta(w)$.
   Based on the definition of shortest path, $\delta(v) \leq dist_{k+1}[v]$. Since $\delta(v) > \delta(w)$ and $\delta(v) \leq dist_{k+1}[v]$, then we have $\delta(w) < dist_{k+1}[v]$([1]).
   Let p_w be the shortest path from s to w, i.e., $\delta(w) = length(p_w)$. Since $w \notin explored$ during the $(k+1)^{th}$ iteration, then there must exists some node along p_w that are not in $explored$. Suppose during the $(k+1)^{th}$ iteration, the first node in p_w that is not in the $explored$ list is w_2, and the node right before w_2 in the s to w_2 subpath is w_1, thus $w_1 \in explored$ during the $(k+1)^{th}$ iteration. The image below illustrates this construction:



   Denote the subpath from s to w_1 in p_w as $p(s, w_1)$, subpath w_1 to w_2 as $p(w_1, w_2)$, and subpath w_2 to w as $p(w_2, w)$. Since w_1 is right before w_2 in p_w, then $length(p(w_1, w_2)) = weight(w_1, w_2)$. Then:

$$\delta(w) = length(p_w) = length(\Delta(s, w_1)) + weight(w_1, w_2) + length(w_2, w)$$

Since all edge weights are positive, then:

$length(\Delta(s, w_1)) + weight(w_1, w_2) \leq \delta(w)$
i.e., $\delta(w_1) + weight(w_1, w_2) \leq \delta(w)$

Since during the $(k+1)^{th}$ iteration, $w_1 \in explored$, then w_1 must be added into $explored$ with all neighbors of w_1 updated during the i^{th} iteration for some $i < k+1$. Then based on our inductive hypothesis, $dist_{k+1}[w_1] = \delta(w_1)$. Since the value of $dist[w_1]$ remains unchanged after adding w_1 into $explored$, then $dist_i[w_1] = dist_{k+1}[w_1] = \delta(w_1)$.
Since w_1 is the node right before w_2 in p_w during the $(k+1)^{th}$ iteration, then $dist_{k+1}[w_2] = dist_i[w_1] + weight(w_1, w_2) = \delta(w_1) + weight(w_1, w_2)$. Since $\delta(w_1) + weight(w_1, w_2) \leq \delta(w)$, then $dist_{k+1}[w_2] \leq \delta(w)$([2]).
Combining [1] and [2], we have:

$$\delta(w) < dist_{k+1}[v]([1])$$
$$dist_{k+1}[w_2] \leq \delta(w)([2])$$

Hence $dist_{k+1}[w_2] < dist_{k+2}[v]$([3]).
Based on our assumption, during the $(k+1)^{th}$ generation, $w_2 \notin explored$ and v is selected by the algorithm, then we must have $dist_{k+1}[w_2] \geq dist_{k+1}[v]$, which contradicts with [3]. Hence by the principle of prove by contradiction, there does not exsist $w \in unexplored$, such that $\delta(v) > \delta(w)$. (1) holds for the $(k+1)^{th}$ iteration.

(Proof below are not modified yet)

2. $dist[v] = \delta(v)$

Suppose $dist[v]$ is associates with path $p \in path(s, v)$ during the k^{th} iteration, and assume the shortest path from s to v is some path $p' \in path(s, v)$ different than p, $length(p') = \delta(v) < dist[v]$([b]). Suppose v' is the node just before v in p'.

$$\delta(v) = dist[v'] + weight(v', v)$$

Since all edge weights are non-negative, then $dist[v'] < \delta(v)$. Based on (1), since $\delta(v) < \delta(w) \forall w \in unexplored$, then v' must be in $explored$. Since v' is in $explored$ and has an edge to v, then the algorithm must have compared $dist[v'] + weight(v', v)$ to the current $dist[v]$ and chose $dist[v]$. Hence it must be $dist[v'] + weight(v', v) \geq dist[v]$, i.e. $\delta(v) \geq dist[v]$, which contradicts with [b]. Hence by the principle of prove by contradiction, p is the shortest path from s to v, and that $dist[v] = \delta(v)$.

Since we proved both (1) and (2) for the k^{th} iteration, forall $k \leq 1$, we have proved that Lemma (1) holds. □

Proof. **Prove of Correctness**

By applying Lemma (1) to the last iteration of the algorithm, we obtained that for all nodes n in the explored list, $dist[n]$ is indeed the shortest path distance value from source s to n, hence Dijkstra's algorithm indeed calculates the shortest path distance value from the source s to each node $n \in g$. □

5 Low-Level Contribution

6 Discussion

7 Related Work

8 Conclusion

References

[3] https://lara.epfl.ch/w/_media/dijkstra.pdf

[2] http://www.cse.chalmers.se/~peterd/papers/DependentTypesAtWork.pdf

[1] https://kola.opus.hbz-nrw.de/opus45-kola/frontdoor/deliver/index/docId/420/file/DA_KLASEN.pdf

[4] http://toccata.lri.fr/gallery/dijkstra.en.html

[5]http://why3.lri.fr/#provers

[6] http://lara.epfl.ch/w/jahob_system

Appendix

Statutory Declaration