VERIFICATION OF SHORTEST PATH ALGORITHMS IN IDRIS

Yazhe Feng

A thesis submitted in partial fulfillment of the requirements for the degree of Bachelor of Arts

in the

Department of Computer Science

at Bryn Mawr College

Advisor: Richard A. Eisenberg

Acknowledgments

(To be finished...)

Abstract

Contents

1 Introduction

Shortest path problems are concerned 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[6] and Bellman-Ford[7] are the most well-known single-source shortest path algorithms, and are implemented in various real-life applications, for instance a variant of Bellman-Ford algorithm is used in Routing Information Protocol, which determines the best routes for data package transportation based on distance.

Given the importance of Dijkstra's and Bellman-Ford in real-life applications, we are interested in verifying the implementation of both algorithms. We provide concrete implementations for both algorithms. Based on the specific implementation, we then define functions with precise type signatures which carry specifications that should hold for the correct implementations of Dijkstra's and Bellman-Ford algorithms, 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. Our implementation uses the Idris functional programming language, which embraces powerful tools and features that makes program verification possible.

Contributions

(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 discusses future work. Section 7 presents and compares related work, and section 8 gives a brief 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 are influenced by Haskell and ML. Features of Idris include but not limit to dependent types, `with` rule, `case` expressions, lambda binding, and interactive editing.

Variables and Types

Idris requires type declarations for all variables and functions defined. To define a variable, we provide the type on one line, and specify the value on the next line. Below presents the syntax for variable declaration.

```
<variable_name> :<type>
<variable_name> = <value>
```

The example below defines a variable `n` of type `Int` with value 37.

```
n : Int
n = 37
```

Types in Idris are first-class values, which means types can be operated as any other values. Type declaration is the same as declaring any other variables, with exactly the same syntax, except that the type of all types is `Type`. By convention, variables that represent types are capitalized. Below example declares a type `CharList`, which denotes the type of list of characters.

```
CharList : Type
CharList = List Char
```

Function

To define a function a Idris, the types for all input values and output values must be specified in the function type signature, connecting by right arrows. Specifically, function type is of the form:

```
<func_name> : x_1 -> x_2 -> ... ->  x_n
```

where $x_1, x_2, ..., x_{n-1}$ are types for the input values, and x_n is the output type of the function. Input values can be annotated to provide more information, and also allows each input to be referred to easily later. For instance the type of the `reverse` function below names the first input as `elem`, which specifies that the input and output lists contain elements of same type.

```
-- "reverse" reverse a list
reverse : (elem : Type) -> List elem -> List elem
```

A function definition is provided on the line below the function type. In Idris, functions are defined by pattern matching, which will be elaborated on later. Here we provide an example for function definition that requires little experience with pattern matching, only aiming to illustrate the syntax for defining functions. The `mult` function defined below multiplies the two input integers.

```
-- "mult" calculates the multiplication of two input integers 'n'
and 'm'
mult : Int -> Int -> Int
mult n m = n * m
```

Data Types

User defined data types are supported in Idris. To define a data type, we provide the name and type of the data type on one line, starting with the keyword `data`, followed by the id of the data type, a colon `:`, the type of the data type, and the keyword `where`. On the next few lines we define the constructors for this data type. Below provides the definition of the natural number type `Nat` in Idris.

```
-- natural number can be either zero(Z) or plus one of another
natural number (S Nat)
data Nat : Type where
  Z : Nat
  S : (n : Nat) -> Nat
```

Idris allows data types to be parameterized over other types. The `List` data type below takes the parameter `elem` of type `Type`, which stands for the type of elements in the list.

```
-- 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]. They allow programmers to specify certain properties of data types explicitly in their type signature. The following example provides a definition of a vector data type, which is indexed by the vector length `len` and parameterized over the element type `elem`.

```
-- 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`, which means a `Vect` of length 3 and 4 are considered as different types. With dependent types, programmers can ensure the behaviors of functions through their type signatures by defining more precise types. Consider a function `concate` that concatenates two `Vect`. As `concate` takes in two vectors, then we have the function type for `concate` as follows:

```
concate : Vect n elem -> Vect m elem -> resultType
```

The output value of `concate` should be the result of concatenating two vectors, which means the resulting vector should have length (n+m), hence `resultType` should be of type `Vect (n+m) elem`. With dependent types, Idris can help to ensure the function correctness of `concate` with the Idris type checker. By providing a function type for `concat` that specifies the length of the output `Vect`, if the definition of `concate` does not return a vector of length (n+m), `concate` would fail type check. Take the following definition of `concate` as an example.

```
concate : Vect n elem -> Vect m elem -> Vect (n+m) elem
concate v1 v2 = v1
```

The type of the above `concate` function specifies that the output value should be a `Vect` of length `(n+m)`, where `n`, `m` are the length of the two input `Vect`, however the function is defined to return the first input vector of length `n`. Idris gives the following error message when compiling this function definition:

```
  |
  | concate v1 v2 = v1
  |                 ~~
When checking right hand side of Haha.concat with expected type
        Vect (n + m) elem

Type mismatch between
        Vect n elem (Type of v1)
and
        Vect (plus n m) elem (Expected type)

Specifically:
        Type mismatch between
                n
        and
                plus n m
```

The error message indicates that the type of the return value of `concate`, i.e., type of `v1`, does not match with the output type specified in the function signature, which is `Vect (n+m) elem`. Idris type checker will only accept definitions for `concate` that return a vector of length `(n+m)`, which helps to ensure the correct behaviors of functions defined. A correct implementation of `concate` is provided below.

```
concate : Vect n Nat -> Vect m Nat -> Vect (n+m) Nat
concate v1 v2 = v1 ++ v2
```

The example above illustrates that dependent types in Idris allow programmers to provide more precise description of function behaviors through function type signatures, which helps to ensure function correctness with the Idris type checker. In verification, dependent types be used to specify program behaviors, and thus allowing us to verify the correctness of program through the Idris type checker.

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 `concate` function that concatenates two vectors, to define `concate`, 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[6], 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

Bellman-Ford algorithm was first introduced by Alfonso Shimbel in 1955, and was published by Richard Bellman and Lester Ford, Jr in 1958 and 1956 respectively[7]. The algorithm solves the issue of calculating the minimum distance value from a single source to all other nodes in a given graph, and different from Dijkstra's algorithm, Bellman-Ford algorithm allows negative edge weights in the input graph, and is capable of detecting the existence of negative cycle(a cycle whose edge weights sum up to a negative value). Applications of Bellman-Ford includes routing protocols such as the Routing Information Protocol.

4 Overview of Algorithms Implementations and Proofs of Correctness

4.1 Dijkstra's Algorithm

4.1.1 Data Structures

Dijkstra's algorithm requires non-negative edge weights and valid input graph, and the data structures in our implementation are designed to ensure these properties of input values. An overview of data structures in our implementation is presented below, and a detailed description is provided under Section 5.
Denote `gsize` as the size of graph, i.e. the number of vertices in a graph. A graph g is defined as a vector containing `gsize` number of adjacent lists, one for each node in the graph, and a node is defined as a data structure carrying a value of type `Fin gsize`. An adjacent list for a node $n \in g$ is defined as a list of tuples $(n', edge_w)$, where the first element n' in each tuple is a neighbor of n in g, and the second element $edge_w$ is the weight of the edge (n, n') in g. To access the adjacent list for a particularly node, the `Fin gsize` type value carried by this node is used to index the graph g. As the graph is defined as a vector of length `gsize`, the definition of node data type ensures that every well-typed node is a valid vertex in the graph, and that each indexing to the graph data structure are guaranteed to be in-bound.
The type of edge weight is user-defined in our implementation. Specifically, we define a `WeightOps` data type, which carries a user-specified type for the edge weight, along with operators and properties proofs for this type, which includes arithmetic operators, proof of non-negative value, and proof of plus associativity. The definition of `Distance` data type is then parameterized over the user-defined edge weight data type. Since all edge weight are non-negative, the value of `Distance` can only be zero, infinity, or sum of edge weights.

4.1.2 Definition

Our implementation and correctness proof are based on the following definitions of key concepts used in Dijkstra's algorithm.

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. Prefix of Path
Given a path from node v to $w : p(v,w) = ve_0v_0e_1v_2....v_{n-1}e_nw$, the prefix of this $v-w$ path is defined as the subsequence of $p(v,w)$ that starts with v and ends with some node $w' \in p(v,w)$ (w' is a vertex in the sequence $p(v,w)$).

Definition 4.3. 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 4.4. 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
$$\forall p' \in path(s,v),\ \delta(v) = length(\Delta(s,v)) \leq length(p')$$

4.1.3 Pseudocode

We denote (u,v) as an edge from node u to v, $weight(u,v)$ as the weight of edge (u,v). Let `gsize` denote the size of the input graph, i.e., the number of nodes in the graph. The type `Graph gsize weight` specifies a graph with `gsize` nodes and edge weight of type `weight`.
Given input graph g and source node s with types:

```
g : Graph gsize weight
s : Node gsize
```

Define $unexplored$ as the list of unexplored nodes, and $dist$ as a list storing 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.

```
(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)
$dist : List\ \ distance$
$dist[s] = 0, dist[a] = \infty, \forall a \in g, a \neq s$
```

We index $unexplored$ and $dist$ by the number of iterations. Specifically, denote u_i as the node being explored at the i^th iteration, and denote $dist_i$, $unexplored_i$ as the value of distance list and unexplored list at the beginning of the i^{th} iteration. Then during each iteration the Dijkstra's Algorithm calculates $dist, unexplored, explored$ as follows:

```
choose $u_k \in unexplored_k$ and $\forall u' \in unexplored_k, dist_k[u_k] \leq dist_k[u']$
$unexplored_{k+1} = unexplored_k - \{u_k\}$
for($\forall v \in g$) {
    $dist_{k+1}[v] = \begin{Bmatrix} min(dist_k[v], (dist_k[u_k] + weight(u_k, v))), & (u_k, v) \in g \\ dist_k[v] & otherwise \end{Bmatrix}$
}
```

4.1.4 Proof of Correctness

This section provides a theoretical proof for our Dijkstra's implementation, which includes proof of program termination and proof of correct program behavior.

4.1.4.1 Lemmas

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. We index $explored$ by the number of iterations, such that $explored_i$ denotes the value of $explored$ at the beginning of the i^{th} iteration.

Lemma 4.1. Given any two nodes v, w, the prefix of the shortest path $\Delta(v, w)$ is also a shortest path.

Proof. We will prove Lemma 4.1 by contradiction.
Consider any node q in the sequence of $\Delta(v, w)$, we have $\Delta(v, w) = ve_0v_0e_1v_2...v_iqv_j....v_{n-1}e_nw$. Suppose the prefix of $\Delta(v, w)$ from v to q, denote as $p(v, q)$, is not the shortest path from v to q. Then we know $p(v, q) = ve_0v_0e_1v_2...v_iq$ is a path from v to q and $length(p(v, q)) > length(\Delta(v, q))$.
Based on the definition of shortest path, we know:

$$length(\Delta(v, w)) \leq length(p), \forall p \in path(v, w)$$

Fenote the path after the node q as $p(q, w) = qv_j....v_{n-1}e_nw$, since $\Delta(v, w) = ve_0v_0e_1v_2...v_iqv_j....v_{n-1}e_nw$, then $\Delta(v, w) = p(v, q) + p(q, w)$, and that $length(\Delta(v, w)) = length(p(v, q)) + length(p(q, w))$. Then we have:

$$length(\Delta(v, w)) = length(p(v, q)) + length(p(q, w)) \leq length(p), \forall p \in path(v, w)$$

Since $p(v, q)$ is not the shortest path from v to q by assumption, then based on the definition of shortest path, $length(p(v, q)) < length(\Delta(v, w))$. Hence there exists another $v - w$ path $p'(v, w)$ such that:

$$p'(v, w) \in path(v, w)$$
$$p'(v, w) = \Delta(v, q) + p(q, w)$$

$$\begin{aligned} length(p'(v,w)) &= length(\Delta(v,q)) + length(p(q,w)) \\ &< length(p(v,q)) + length(p(q,w)) \\ \text{i.e. } length(p'(v,w)) &< length(\Delta(v,w)) \end{aligned}$$

Hence we have reached a contradiction. Thus by the principle of prove by contradiction, for any the prefix $p(v,q)$ of $\Delta(v,w)$ is the shortest path from v to q. Lemma 4.1 holds. □

Lemma 4.2. After the n^{th} iteration for $n \geq 1$, forall node $v \in explored_{n+1}$, if $dist_{n+1}[v] \neq \infty$, then $dist_{n+1}[v]$ is the length of some $s - v$ path, i.e, $path(s,v) \neq \emptyset$.

Proof. We will prove `Lemma 4.2` by inducting on the number of iterations.
Let P(n) be: After the n^{th} iteration, $n \geq 1$, for all node $v \in g$, if $dist_{n+1}[v] \neq \infty$, then $dist_{n+1}[v]$ is the length of some $s - v$ path.

Base Case : We shall show P(1) holds.
Based on the algorithm, initially $dist_1[s] = 0$ and for all node $v \in g, v \neq s, dist_1[v] = \infty$, then s is the only node whose distance value is not infinity. Based on the definition of path, the path from the source node s to itself is s, $path(s,s) = \{s\}$. Hence P(1) holds.

Inductive Hypothesis : Suppose $\forall i, 1 \leq i \leq k$, P(i) holds. That is, after the i^{th} iteration, $1 \leq i \leq k$, for all nodes $v \in g$, if $dist_{i+1}[v] \neq \infty$, then $dist_{n+1}[v]$ is the length of some $s - v$ path.

Inductive Step : We shall show P(k+1) holds.
For node u_{k+1} being explored during the $(k+1)^{th}$ iteration, based on the algorithm, $dist_{k+1}[u_{k+1}]$ is calculated as:

$$dist_{k+2}[u_{k+1}] = \begin{cases} min(dist_{k+1}[u_{k+1}], dist_{k+1}[u_{k+1}] + weight(u_{k+1}, u_{k+1})), & (u_{k+1}, u_{k+1}) \in g \\ dist_{k+1}[u_{k+1}] & otherwise \end{cases}$$

Since the distance value from u_{k+1} to itself is 0, then $dist_{k+2}[u_{k+1}] = dist_{k+1}[u_{k+1}]$, and that $dist_{k+2}[u_{k+1}]$ and $dist_{k+1}[u_{k+1}]$ are the length of the same $s - u_{k+1}$ path if there exists one.
If $dist_{k+2}[u_{k+1}] \neq \infty$, then $dist_{k+1}[u_{k+1}] = dist_{k+2}[u_{k+1}] \neq \infty$. Since $k \leq k$ and $dist_{k+1}[u_{k+1}] \neq \infty$, then based on the inductive hypothesis, $dist_{k+1}[u_{k+1}]$ is the length of some $s - u_{k+1}$ path, and hence $dist_{k+2}[u_{k+1}]$ is the length of some $s - u_{k+1}$ path.
Then for all node $v \in g$ other than u_{k+1}, there are two cases: (1) $(u_{k+1}, v) \in g$; (2) u_{k+1} does not have an edge to v. We will prove P(k+1) holds in both cases separately.

Case (1): $(u_{k+1}, v) \in g$
Based on the algorithm, as $(u_{k+1}, v) \in g$, $dist_{k+2}[v] = min(dist_{k+1}[v], dist_{k+1}[u_{k+1}] + weight(u_{k+1}, v))$.

- If $dist_{k+1}[v] < dist_{k+1}[u_{k+1}] + weight(u_{k+1}, v)$, then $dist_{k+2}[v] = dist_{k+1}[v]$. Then if $dist_{k+2}[v] \neq \infty$, we have $dist_{k+1}[v] \neq \infty$, and that $dist_{k+2}[v]$ and $dist_{k+1}[v]$ are the length of the same $s-v$ path if there exists one. Since $dist_{k+1}[v] \neq \infty$, the inductive hypothesis implies that $dist_{k+1}[v]$ is the length of some $s-v$ path, hence $dist_{k+2}[v]$ is the length of some $s-v$ path. P(k+1) holds.

- If $dist_{k+1}[v] \geq dist_{k+1}[u_{k+1}]+weight(u_{k+1},v)$, then $dist_{k+2}[v] = dist_{k+1}[u_{k+1}]+weight(w,v)$. If $dist_{k+2}[v] \neq \infty$, then it follows that $dist_{k+1}[u_{k+1}] = dist_{k+2}[v] - weight(u_{k+1},v) \neq \infty$. Then the inductive hypothesis implies that $dist_{k+1}[u_{k+1}]$ must be the length of some $s-u_{k+1}$ path, denote as $p(s,u_{k+1})$. Since there is an edge $(u_{k+1},v) \in g$, then $dist_{k+2}[v] = dist_{k+1}[u_{k+1}] + weight(u_{k+1},v)$ must be the length of the $s-v$ path through u_{k+1}. P(k+1) holds.

Hence P(k+1) holds under under `Case(1)`.

Case (2): u_{k+1} does not have an edge to v
Under this case, our algorithm indicates that $dist_{k+2}[v] = dist_{k+1}[v]$, and that $dist_{k+1}[v]$ and $dist_{k+2}[v]$ are the length of the same $s-v$ path if there exists one. If $dist_{k+1}[v] = dist_{k+2}[v] \neq \infty$, then based on the inductive hypothesis, $dist_{k+1}[v]$ is the length of some $s-v$ path, and hence $dist_{k+2}[v]$ is the length of some $s-v$ path. P(k+1) holds under `Case(2)`.

We have proved P(k+1) holds for u_{k+1} and both cases for all nodes $v \in g$ other than u_{k+1}. Hence by the principle of prove by induction, P(n) holds. Thus `Lemma 4.2` holds. □

Lemma 4.3. For any node $v \in g$, if after the i^{th} iteration, $dist_{i+1}[v] = \delta(v)$, then for each proceeding j^{th} iteration, $j > i$, $dist_{j+1}[v] = dist_{i+1}[v] = \delta(v)$.

Proof. We will prove `Lemma 4.3` by induction on the number iterations after the i^{th} iteration.
Let P(n) be: For any node $v \in g$, if after the i^{th} iteration, $dist_{i+1}[v] = \delta(v)$, then for the $(i+n)^{th}$ iteration, $n \geq 1$, $dist_{i+n+1}[v] = dist_{i+1}[v] = \delta(v)$

Base Case : We shall show P(1) holds.
During the $(i+1)^{th}$ iteration, suppose u_{i+1} is the node being explored, then $dist_{i+2}[v]$ is calculated as:

$$dist_{i+2}[v] = \begin{Bmatrix} min(dist_{i+1}[v], dist_{i+1}[u_{i+1}] + weight(u_{i+1},v)), & (u_{i+1},v)) \in g \\ dist_{i+1}[v] & otherwise \end{Bmatrix}$$

If $(u_{i+1},v)) \in g$, then if $dist_{i+1}[u_{i+1}]$ is the length of some $s-u_{i+1}$ path, then $(dist_{i+1}[u_{i+1}] + weight(u_{i+1},v))$ is the length of some $s-v$ path. Since $dist_{i+1}[v] = \delta(v)$, then based on the definition of shortest path, $dist_{i+1}[v] \leq dist_{i+1}[u_{i+1}] + weight(u_{i+1},v)$, and hence $dist_{i+2}[v] = dist_{i+1}[v] = \delta(v)$.
If u_{i+1} does not have an edge to v, then $dist_{i+2}[v] = dist_{i+1}[v] = \delta(v)$.
Hence in either cases, $dist_{i+2}[v] = dist_{i+1}[v] = \delta(v)$. P(1) holds.

Inductive Hypothesis : Suppose P(k) holds, that is, if after the i^{th} iteration, $dist_{i+1}[v] = \delta(v)$, then for the $(i+k)^{th}$ iteration, $n \geq 1$, $dist_{i+k+1}[v] = dist_{i+1}[v] = \delta(v)$.

Inductive Step : We shall show P(k+1) holds.
For the node u_{i+k+1} being explored during the $(i+k+1)^{th}$ iteration, there are two cases: (1) $(u_{i+k+1}, v) \in g$; (2) u_{i+k+1} does not have an edge to v. We will show that P(k+1) holds under both cases separately.
Case 1: $(u_{i+k+1}, v) \in g$
If u_{i+k+1} has an edge to v, then based on the algorithm, for $dist_{i+k+2}[v]$, we have:

$$dist_{i+k+2}[v] = min(dist_{i+k+1}[v], dist_{i+k+1}[u_{i+k+1}] + weight(u_{i+k+1}, v))$$

Since based on our inductive hypothesis, $dist_{i+k+1}[v] = dist_{i+1}[v] = \delta(v)$, then if $dist_{i+k+1}[u_{i+k+1}]$ is the length of some $s - u_{i+k+1}$ path, then $(dist_{i+k+1}[u_{i+1}] + weight(u_{i+k+1}, v))$ is the length of some $s - v$ path, and hence $dist_{i+k+1}[v] = \delta(v) \leq (dist_{i+k+1}[u_{i+1}] + weight(u_{i+k+1}, v))$. Then:

$$\begin{aligned} dist_{i+k+2}[v] &= min(dist_{i+k+1}[v], dist_{i+k+1}[u_{i+k+1}] + weight(u_{i+k+1}, v)) \\ &= dist_{i+k+1}[v] \\ &= dist_{i+1}[v] = \delta(v) \end{aligned}$$

P(k+1) holds under `Case 1`.
Case 2: u_{i+k+1} **does not have an edge to** v
Since u_{i+k+1} does not have an edge to v, then $dist_{i+k+2}[v] = dist_{i+k+1}[v]$. Based on the inductive hypothesis, $dist_{i+k+1}[v] = dist_{i+1}[v] = \delta(v)$. then $dist_{i+k+2}[v] = dist_{i+1}[v] = \delta(v)$. P(k+1) holds for `Case (2)`.
Thus P(k+1) holds. By the principle of prove by induction, P(n) holds. `Lemma 4.3` proved.

□

Lemma 4.4. Assume g is a connected graph, that the source node s has a path to every node in g. After the n^{th} iteration of the algorithm for $n \geq 1$, forall node $v \in explored_{n+1}$, we have:

1. $\delta(v) \leq \delta(v')$, $\forall v' \in unexplored_{n+1}$.
2. $dist_{n+1}[v] = \delta(v)$

Proof. We will prove `Lemma 4.4` by inducting on the number of iterations.
Let P(n) be: After the n^{th} iteration of the algroithm for $n \geq 1$, forall node $w \in explored_{n+1}$: (1) $\delta(w) \leq \delta(w')$, $\forall w' \in unexplored_{n+1}$; (2) $dist_{n+1}[w] = \delta(w)$.

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 $unexplored_1$ and added to $explored_2$. Since all edge weights are non-negative, then the shortest distance value from s to s is indeed 0, hence $dist_2[s] = 0 = \delta(s)$ and $\delta(s) \leq \delta(v')$, $\forall v' \in unexplored_2$.
P(1) holds.

Inductive Hypothesis : Suppose P(i) is true for all $1 \leq i \leq k$. That is, after the i^{th} iteration forall $1 < i \leq k$, forall node $w \in explored_{i+1}$: (1) $\delta(w) \leq \delta(w')$, $\forall w' \in unexplored_{i+1}$; (2) $dist_{i+1}[w] = \delta(w)$;

Inductive Step : We shall show P(k+1) holds. That is, forall node $w \in explored_{k+2}$, (1) $\delta(w) \leq \delta(w')$, $\forall w' \in unexplored_{k+2}$; (2) $dist_{k+2}[w] = \delta(w)$;
Suppose u_{k+1} is the node added into *explored* during the $(k+1)^{th}$ iteration, then $explored_{k+2} = explored_{k+1} \cup \{u_{k+1}\}$. We will show that (1) and (2) holds for all nodes in $explored_{k+1}$ in `Part (a)`, and `Part (b)` proves (1) and (2) holds for u_{k+1}, so that (1) and (2) holds forall nodes in $explored_{k+2}$.

- **Part(a)**: `WTP: After the` $(k+1)^{th}$ `iteration,` $\forall w \in explored_{k+1}$`, (a.1)` $\delta(w) \leq \delta(w')$, $\forall w' \in unexplored_{k+2}$`; (a.2)` $dist_{i+1}[w] = \delta(w)$

  Consider each node $q \in (explored_{k+1} \cap explored_{k+2}) = explored_{k+1}$, q must be explored before the $(k+1)^{th}$ iteration. Suppose q is explored during the i^{th} iteration for some $i < k+1$, then based on our inductive hypothesis, $dist_{i+1}[q] = \delta(q)$, and $\delta(q) \leq \delta(q'), \forall q' \in unexplored_{i+1}$.
  Proof of (a.1): Based on the algorithm, for each iteration, the algorithm explores exactly one node and never revisits any explored nodes. For each node $q \in explored_{k+1}$ mentioned above, since q is explored before the $(k+1)^{th}$ iteration, then $unexplored_{k+1} \subseteq unexplored_{i+1}$. Since $\delta(q) \leq \delta(q'), \forall q' \in unexplored_{i+1}$, and $unexplored_{i+1}$ includes all node in $unexplored_{k+1}$, then $\delta(q) \leq \delta(q'), \forall q' \in unexplored_{k+1}$. (1) holds for $explored_{k+1}$.
  Proof of (a.2): For all proceeding j^{th} iterations, $j > i$, suppose node q'' is the node being explored for the j^{th} iteration, then the value of $dist_{j+1}[q]$ is calculated as:

$$dist_{j+1}[q] = \begin{cases} min(dist_j[q], dist_j[q''] + weight(q'', q)), & (q'', q) \in g \\ dist_j[q] & otherwise \end{cases}$$

  Since $dist_{i+1}[q] = \delta(q) \leq length(p)$ for all path p from s to q, then for each proceeding j^{th} iteration after the i^{th} iteration, there does not exists such q'' such that $dist_j[q''] + weight(q'', q) < \delta(q) = dist_{i+1}[q]$. Hence $dist_{j+1}[q] = \delta(q) = dist_{i+1}[q], \forall j > i$. Since $k+1 > i$, then for all $q \in S$, $dist_{k+1} = \delta(q) = dist_{i+1}[q]$. (2) holds for $explored_{k+1}$.

  Hence we have proved that both (1) and (2) holds for all nodes in $explored_{k+1}$.

- **Part(b)**: After the $(k+1)^{th}$ iteration, (1) and (2) holds for u_{k+1}.
  We want to show: **(b.1)** $\delta(u_{k+1}) \leq \delta(v')$, $\forall v' \in unexplored_{k+2}$; and **(b.2)** $dist_{k+1}[u_{k+1}] = \delta(u_{k+1})$.
  Proof of (b.1): $\delta(u_{k+1}) \leq \delta(v')$, $\forall v' \in unexplored_{k+2}$
  We will prove (b.1) by contradiction. Suppose there exists $w \in unexplored_{k+2}$, such that $\delta(u_{k+1}) > \delta(w)$.
  The assumption states that source s has a path to every node in g, then $dist_{k+1}[u_{k+1}] \neq \infty$. Thus `Lemma 3.2` implies that $dist_{k+1}[u_{k+1}]$ is the length of some $s - v$ path. Based on the definition of shortest path, $\delta(u_{k+1}) \leq dist_{k+1}[u_{k+1}]$. Since $\delta(u_{k+1}) > \delta(w)$ and $\delta(u_{k+1}) \leq dist_{k+1}[u_{k+1}]$, then we have $\delta(w) < dist_{k+1}[u_{k+1}]$([NE1]).
  Consider the shortest path $\Delta(s, w)$ from s to w, $\delta(w) = length(\Delta(s, w))$. Since $w \notin$

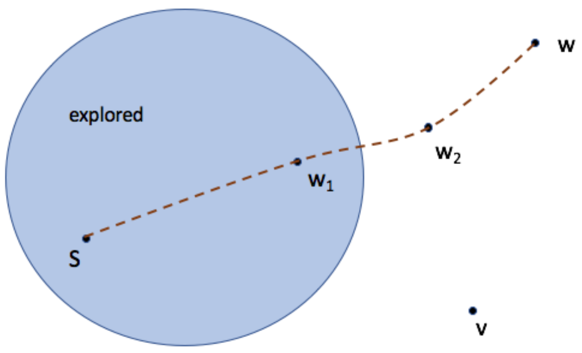

Figure 1: construction of shortest path from `s` to `w`

$explored_{k+2}$, then there must exists some node in $\Delta(s,w)$ that are not in $explored_{k+2}$. Suppose the first node along $\Delta(s,w)$ that is not in the $explored_{k+2}$ list is w_2, and the node right before w_2 in the s to w_2 subpath is w_1, thus $w_1 \in explored_{k+2}$. Fig.1 below illustrates this construction:

Denote the subpath from s to w_1 in $\Delta(s,w)$ as $p(s,w_1)$, subpath from s to w_2 in $\Delta(s,w)$ as $p(s,w_2)$, and subpath w_2 to w as $p(w_2,w)$. Based on `Definition 2.2 Prefix of Path`, $p(s,w_1)$ is a prefix of $\Delta(s,w)$. Since $p(s,w_1)$ is the prefix of the shortest $s-w$ path, then based on `Lemma 3.1`, $p(s,w_1)$ is the shortest path from s to w_1, $\Delta(s,w_1) = p(s,w_1)$, $length(p(s,w_1)) = \delta(w_1)$.
Similarly, since $p(s,w_2) = p(s,w_1) + (w_1,w_2)$, then $p(s,w_2)$ is a prefix of $\Delta(s,w)$, and hence `Lemma 3.1` implies that $p(s,w_2)$ is the shortest path from s to w_2. Then we have:

$$\begin{aligned}\Delta(s,w_2) &= p(s,w_2) = p(s,w_1) + (w_1,w_2)\\ \delta(w_2) &= length(\Delta(s,w_2))\\ &= length(p(s,w_2))\\ &= length(p(s,w_1)) + weight(w_1,w_2)\\ &= \delta(w_1) + weight(w_1,w_2) \text{ ([E1])}\end{aligned}$$

For $\Delta(s,w)$ we have:

$$\begin{aligned}\delta(w) &= length(p_w)\\ &= length(p(s,w_1)) + weight(w_1,w_2) + length(p(w_2,w))\\ &= \delta(w_1) + weight(w_1,w_2) + length(p(w_2,w))\end{aligned}$$

Since all edge weights are positive, then:

$$\delta(w_2) = \delta(w_1) + weight(w_1,w_2) \leq \delta(w) \text{ ([E2])}$$

Since $w_1 \in explored_{k+2}$, there are two cases to consider: $w_1 = u_{k+1}$ and $w_1 \neq u_{k+1}$. We will prove P(k+1) under both cases below.

`Case 1:` $w_1 = u_{k+1}$

When $w_1 = u_{k+1}$, then substitude w_1 by u_{k+1} in [E2], we have:

$$\delta(w_1) + weight(w_1, w_2) = \delta(u_{k+1}) + weight(u_{k+1}, w_2) \leq delta(w)$$
i.e. $\delta(u_{k+1}) \leq \delta(w)$

which contradicts with our assumption that $\delta(u_{k+1}) > \delta(w)$. Hence by the principle of prove by contradiction, $\delta(u_{k+1}) < \delta(w)$. (1) holds for P(k+1).

`Case 2`: $w_1 \neq u_{k+1}$
Since $w_1 \in explored_{k+2}$ and $w_1 \neq u_{k+1}$, w_1 is explored before the $(k+1)^{th}$ iteration. i.e., $w_1 \in explored_{k+1}$. Suppose w_1 is being explored during the i^{th} iteration, $i < k+1$, then based on the algorithm, the value of $dist_{i+1}[w_1]$ is calculated as:

$$dist_{i+1}[w_1] = \begin{Bmatrix} min(dist_i[w_1], dist_i[w_1] + weight(w_1, w_1)), & (w_1, w_1) \in g \\ dist_i[w_1] & otherwise \end{Bmatrix}$$

Thus $dist_{i+1}[w_1] = dist_i[w_1]$. Since the inductive hypothesis implies that $dist_{i+1}[w_1] = \delta(w_1)$, then $dist_i[w_1] = \delta(w_1)$.
Since w_1 has an edge to w_2, then $dist_{i+1}[w_2]$ must have been updated according as follows:

$$\begin{aligned} dist_{i+1}[w_2] &= min(dist_i[w_2], dist_i[w_1] + weight(w_1 + w_2)) \\ &= min(dist_i[w_2], \delta(w_1) + weight(w_1 + w_2)) \end{aligned}$$

Based on [E1] we know that $\delta(w_2) = \delta(w_1) + weight(w_1 + w_2)$, then $dist_{i+1}[w_2] = min(dist_i[w_2], \delta(w_2))$. If $dist_i[w_2] = \infty$, then $dist_{i+1}[w_2] = min(dist_i[w_2], \delta(w_2)) = \delta(w_2)$. If $dist_i[w_2] \neq \infty$, then based on `Lemma 3.2`, $dist_i[w_2]$ is the length of some $s - w_2$ path. Since $\delta(w_2) \leq length(p), \forall p \in path(s, w_2)$, then $dist_{i+1}[w_2] = min(dist_i[w_2], \delta(w_2)) = \delta(w_2)$. Hence in either cases, we conclude that $dist_{i+1}[w_2] = \delta(w_2)$.
Since $dist_{i+1}[w_2] = \delta(w_2)$ and $i < k+1$, then based on `Lemma 3.3`, we have $dist_{k+1}[w_2] = dist_{i+1} = \delta(w_2)$. Based on [E2], $\delta(w_2) < \delta(w)$, then $dist_{k+1}[w_2] < \delta(w)$ [NE2]. Combining with [NE1], we have:

$$\delta(w) < dist_{k+1}[u_{k+1}] \text{ (from [NE1])}$$
$$dist_{k+1}[w_2] < \delta(w)$$

Hence $dist_{k+1}[w_2] < dist_{k+1}[u_{k+1}]$ [NE2].
Based on our assumption, at the beginning of the $(k+1)^{th}$ generation, $u_{k+1}, w_2 \notin explored_{k+1}$ and u_{k+1} is selected by the algorithm, then we must have $dist_{k+1}[w_2] \geq dist_{k+1}[u_{k+1}]$, which contradicts with [NE2]. Hence by the principle of prove by contradiction, there does not exsist $w \in unexplored_{k+2}$, such that $\delta(u_{k+1}) > \delta(w)$, i.e. $\delta(u_{k+1}) \leq \delta(w), \forall w \in unexplored_{k+2}$. Hence (b.1) holds.

Proof of (b.2): After the $(k+1)^{th}$ iteration, $dist_{k+1}[u_{k+1}] = \delta(u_{k+1})$
We will prove this by contradiction.

Suppose $dist_{k+1}[u_{k+1}]$ is the length of some path p from s to u_{k+1}. Assume the shortest path from s to u_{k+1} is some path different from p, i.e. $\Delta(s, u_{k+1}) \neq p$, $\delta(u_{k+1}) \leq dist_{k+1}[u_{k+1}]$([NE3]). Suppose v' is the node just before u_{k+1} in $\Delta(s, u_{k+1})$.

$$\delta(u_{k+1}) = \delta(v') + weight(v', u_{k+1}) < dist_{k+1}[u_{k+1}]$$

Since all edge weights are non-negative, then: $\delta(v') \leq \delta(u_{k+1})$

Based on (a.1) and (b.1), after the $(k+1)^{th}$ iteration, for all nodes $q \in unexplored_{k+2}$, $\delta(q) \geq \delta(u_{k+1})$, and $\delta(v') \leq \delta(u_{k+1})$, then v' cannot be in $unexplored_{k+2}$. Since $unexplored_{k+1} = unexplored_{k+2} \cup u_{k+1}$, then $v' \notin unexplored_{k+1}$. Hence at the beginning of the $(k+1)^{th}$ iteration, v' is already explored. Since v' is explored before the $(k+1)^{th}$ iteration and v' has an edge to u_{k+1}, then the algorithm must have considered $(\delta(v') + weight(v', u_{k+1}))$ against $dist_{k+1}[u_{k+1}]$ and chose $min((\delta(v') + weight(v', u_{k+1})), dist_{k+1}[u_{k+1}])$, which is $dist_{k+1}[u_{k+1}]$. Thus $dist_{k+1}[u_{k+1}] \leq (\delta(v')+weight(v', u_{k+1}))$, i.e. $dist_{k+1}[u_{k+1}] \leq \delta(u_{k+1})$, which contradicts with our assumption [NE3]. Hence by the principle of prove by contradiction, $dist_{k+1}[u_{k+1}] = \delta(u_{k+1})$. (b.2) holds.

Since we have proved both (1) and (2) forall nodes in $explored_{k+1}$ after the $(k+1)^{th}$ iteration, P(k+1) holds. Then by the principle of prove by induction, `Lemma 4.4` holds. □

4.1.4.2 Proof of Termination

Proof. 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. □

4.1.4.3 Prove of Correctness

Proof. By applying `Lemma 3.4` 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 Concrete Implementation of Dijkstra's Verification

Our verification program consists three parts: data structures, implementation of Dijkstra's algorithm, and verification of the implementation. We implemented Dijkstra's algorithm with a matrix representation, where each column of the matrix represents one iteration of the algorithm and carries the source node, current list of unexplored nodes, and distance values of all nodes calculated by the algorithm. New column is then calculated based on the existing columns, and the last column calculated is the output, which contains the minimum distance values from source to all nodes in the graph. Implementation details are provided in the following sections.

5.1 Data Structures

Key structures of our implementation include `WeightOps`, `Distance`, `Graph`, and `Column`. Our implementation allows edge weight type to be user defined, with `WeightOps` specifying all the properties that users need to provide. Below presents the definition of `WeightOps`.

```
using (weight : type)
  record WeightOps weight where
    constructor MKWeight
    zero : weight
    gtew : weight -> weight -> Bool
    eq : weight -> weight -> Bool
    add : weight -> weight -> weight
    eqRefl : {w : weight} -> eq w w = True
    eqComm : {w1, w2 : weight} ->
           eq w1 w2 = True ->
           eq w2 w1 = True
    gteRefl : {a : weight} -> (gtew a a = True)
    gteReverse : {a, b : weight} ->
          (p : gtew a b = False) ->
          gtew b a = True
    gteComm : {a, b, c : weight} ->
              (p1 : gtew a b = True) ->
              (p2 : gtew b c = True) ->
              gtew a c = True
    gteBothPlus : {a, b : weight} ->
                  (c : weight) ->
                  (p1 : gtew a b = False) ->
                  gtew (add a c) (add b c) = False
    triangle_ineq : (a : weight) ->
            (b : weight) -> gtew (add a b) a = True
    gtewPlusFalse : (a, b : weight) -> gtew a (add b a) = False
    gtewEqTrans : {w1, w2, w3 : weight} ->
            (eq w1 w2 = True) ->
            (b : Bool) ->
            (gtew w2 w3 = b) ->
            gtew w1 w3 = b
    addComm : (a : weight) ->
          (b : weight) ->
          add a b = add b a
```

The type constructor of `WeightOps` takes in the userdefined edge weight type and returns a type, and the data contructor `MKWeight`, which takes in all the properties specified by the projection functions, builds the `WeightOps weight` type. As Dijkstra's algorithm requires non-negative edge weights, the user-defined edge weight type is required to fulfill triangle inequality, as specified by the `triangle_ineq` function.

Our implementation defined a few key structures to represent graph and its main components, which includes `Node, nodeset, Graph` and `Path`. Definition of each data types are specified below.

The `Node` type reprensents a node in the graph. As presented in the definition of `Node` below, the type constructor of `Node` takes in a `Nat` that specifies the size of the input graph (i.e., the number of nodes in the graph), and the data constructor `MKNode` takes in a `Fin n` type, which carries a natural number that is strictly smaller than `n`, and builds a node of type `Node n`. Such construction ensures that the natural number value carried by each node is strictly smaller than the size of the graph. As the value carried by each `Node` type is used to index distance value in the graph, this ensures that each indexing is in-bound. Below presents the definition of Node type. Any well-typed Node n is a valid node in a graph of size n, and any valid node in the graph must have a correcsponding Node n value.

```
  data Node : Nat -> Type where
    MKNode : Fin n -> Node n
```

We define a nodeset type to carry the set of pairs of adjacent node and corresponding edge weight for a specifi node. As the number of neighboring nodes is undecidable for each node in the input graph, nodeset is defined as a List rather than a Vect.
Graph is defined based on Node and ndoeset. The type of Graph carries a Nat that specifies the size of the graph, the user defined edge weight weight, and WeightOps weight that carries properties of the edge weight type. Data constructor MKGraph takes in the graph size, denotes as gsize, the type of edge weight weight, WeightOps weight, and a vector of gsize number of nodesets, one for each node in the graph. As the definiton of Node type ensures that a node is valid if and only if it has a corresponding value of type Node gsize, it is not necessary for the Graph data type to carry a list of all nodes in the graph. Below are the definition of nodeset and Graph types.

```
nodeset : (gsize : Nat) -> (weight : Type) -> Type
    nodeset gsize weight = List (Node gsize, weight)

data Graph : Nat -> (weight : Type) -> (WeightOps weight)-> Type where
  MKGraph : (gsize : Nat) ->
            (weight : Type) ->
            (ops : WeightOps weight) ->
            (edges : Vect gsize (nodeset gsize weight)) ->
            Graph gsize weight ops
```

Path is defined as a sequence of non-repeating nodes, where each two adjacent nodes have an edge in the graph. A path can contain only one node, as specified by the Unit data constructor below. The Cons data constructor allows a new path to be constructed from an existing path, that given a path from node s to v, if n is an adjacent to v (adj g v n denotes that there is an edge from v to n in the graph g), then we can obtain a new path from s to n by appending the node n to the end of the existing s-to-v path.

```
data Path : Node gsize ->
            Node gsize ->
            Graph gsize weight ops -> Type where
  Unit : (g : Graph gsize weight ops) ->
         (n : Node gsize) ->
         Path n n g
  Cons : Path s v g ->
         (n : Node gsize) ->
         (adj : adj g v n) ->
         Path s n g
```

A shortest path from node s to v is then defined as a path whose length is smaller than or equal to any other s-to-v paths in the graph, as presented below.

```
shortestPath : (g : Graph gsize weight ops) ->
               (sp : Path s v g) ->
```

```
                 Type
  shortestPath g sp {ops} {v}
    = (lp : Path s v g) ->
      dgte ops (length lp) (length sp) = True
```

We defined a Column type to represent one column of the matrix generated by the algorithm, which contains the input graph, the source node, the number of current unexplored nodes, a vector of current unexplored nodes, and a vector of distance values from source to all nodes in the graph. The definition of Column type is providede below.

```
  data Column : Nat -> (Graph gsize weight ops) -> (Node gsize) -> Type where
    MKColumn : (g : Graph gsize weight ops) ->
               (src : Node gsize) ->
               (len : Nat) ->
               (unexp : Vect len (Node gsize)) ->
               (dist : Vect gsize (Distance weight)) ->
               Column len g src
```

Such definition of Column data type provides enough information for us to calculate the current unexplored nodes with minimum distance value, and the updated distance values for all nodes for the next column. Given an input graph of size gsize, the first column in the matrix should have length gsize as all nodes are unexplored, and the last column of the matrix should contain an empty vector for unexlored nodes, as well as a vector of the minimum value from source to all nodes in the graph.

(To be continued....)

5.2 Implementation of Dijkstra's Algorithm

5.3 Lemmas

We present the type signatures of key lemmas of Dijkstra's verification.

The first lemma specifies that the prefix of shortest path is also a shortest path. We first provide the definition of prefix of path below.

```
pathPrefix : (pprefix : Path s w g) ->
             (p : Path s v g) ->
             Type
pathPrefix pprefix p {w} {v} {g}
  = (ppost : Path w v g ** append pprefix ppost = p)
```

The function pathPrefix specifies that, given a path p in g from node s to v, a path pprefix of type Path s w g is a prefix of p if there exists a path ppost of type Path w v g, which is a path from node w to v in g, such that the path obtained by appending pprefix to ppost is equal to p.

The type of the first lemma l1_prefixSP is defined as follows. Given an input graph g, nodes s, v, w, a path sp from s to v, path sp_pre from s to w, if sp is a shortest s-to-v path in g, (specified by shortestPath g sp), and that sp_pre is a prefix of sp (specified by pathPrefix sp_pre sp), then sp_pre is a shortest s-to-w path in g.

```
l1_prefixSP : {g: Graph gsize weight ops} ->
              {s, v, w : Node gsize} ->
              {sp : Path s v g} ->
              {sp_pre : Path s w g} ->
              (shortestPath g sp) ->
              (pathPrefix sp_pre sp) ->
              (shortestPath g sp_pre)
```

(To be continued....)

5.4 Verification of Correctness

The implementation of lemma proofs in the previous section shows that if certain properties, such as those specified by the function `l5_stms`, holds for the current column `cl`, then they must hold for the new column generated based on `cl`. With the proofs of the lemmas, we are able to define the below recursive function, `correctness`, which specifies that given a column `cl` relating to an input graph `g` and source node `src`, if all properties stated by `neDInfPath` and `l5_stms` hold for `cl` (specified by `l2_ih` and `l5_ih` inputs), then the properties should also hold after calling `runDijkstras` on `cl`. We updates the inputs to the next recursive call by applying lemmas to `l2_ih` and `l5_ih`, which is indeed equivalent to the inductive steps in our theoretical proofs of Dijkstra's algorithm provided back in Section 4.

```
correctness : {g : Graph gsize weight ops} ->
              (cl : Column len g src) ->
              (nadj : ((n : Node gsize) -> inNodeset n (getNeighbors g n) = False)) ->
              (l2_ih : neDInfPath cl) ->
              (l5_ih : l5_stms cl) ->
              l5_stms (runDijkstras cl)
correctness {len = Z} cl nadj l2_ih l5_ih = l5_ih
correctness {len=S n} cl@(MKColumn g src (S n) unexp dist) nadj l2_ih l5_ih
  = correctness (runHelper {len=n} cl) nadj
          (l2_existPath cl l2_ih)
          (l5_spath cl nadj l2_ih l5_ih)
```

We then defined a `dijkstras_correctness` function that wraps up all proofs and verify the minimum distance property for all nodes in the input graph.

```
dijkstras_correctness : (gsize : Nat) ->
                        (g : Graph gsize weight ops) ->
                        (src : Node gsize) ->
                        (v : Node gsize) ->
                        (psv : Path src v g) ->
                        (spsv : shortestPath g psv) ->
                        (nadj : ((n : Node gsize) ->
                            inNodeset n (getNeighbors g n) = False)) ->
                        dEq ops (indexN (finToNat (getVal v))
                                (dijkstras gsize g src nadj)
                                {p=nvLTE {gsize=gsize} (getVal v)})
                            (length psv) = True
```

(To be continued....)

6 Discussion

7 Related Work

The increasing importance of Dijkstra's algorithm in many real-world applications has raised an interest on verifying it's implementation. Robin Mange and Jonathan Kuhn provide a project that verifies a Java implementation of Dijkstra's algorithm with the Jahob verification system in their report on efficient proving of Java programs[3]. Although we failed to obtain the concrete implementation of this work, the report demonstrates the verification process. Function behaviors are specified with preconditions, frame conditions, and postconditions, and Jahob allows programmers to provide these specifications in high-level logic(HOL), which reduces the problem of program verification to the validity of HOL formulas.

Klasen et. al. from the University of Koblenz and Landau verifies Dijkstra's algorithm with the KeY system[1], an interactive theorem prover for Java. Concrete implementations of Dijstra's algorithm with different variants are provided, and all of them are written in Java. Simiarly to the work by Mange and Kuhn, the verification process in the work by Klasen involves describing the behavior of each function with pre- and postconditions and modifies clause. Loop invariants are specified to support the verification. A function is then examine as correct by the KeY systemm, with respect to its behavior specifications, if the postconditions specified hold after execution. A similar implementation is provided by Jean-Christophe Filliâtre, a senior researcher from the National Center for Scientific Research(CNRS), which verifies Dijkstra's implementation with Why3, a deductive program verification platform that relies on external theorem provers[4][5]. All works presented above are largely dependent on theorem proving systems, however our work relies on a significantly smaller trusted code base. Most proofs in our work will be implemented from scratches, and considerable amount of details on verification will be presented explicitly.

In spite of the popularity of Bellman-Ford algorithm in network applications, no resources are found on verifying implementations of Bellman-Ford algorithm.

8 Conclusion

References

[1] V. Klasen, "Verifying dijkstra's algorithm with key," in Diploma Thesis, Universitat Koblenz-Landau, 2010.
[2] Bove, Ana, and Peter Dybjer. “Dependent Types at Work.” Language Engineering and Rigorous Software Development Lecture Notes in Computer Science, 2009, pp. 57–99., `doi:10.1007/978-3-642-03153-3_2`.
[3] R. Mange and J. Kuhn, "Verifying dijkstra algorithm in Jahob," 2007, student project, EPFL.
[4] Filliâtre, Jean-Christophe. “Toccata.” Dijkstra's Shortest Path Algorithm, `toccata.lri.fr/gallery/dijkstra.en.html`.
[5] “Why3.” Why3, May 2015, `why3.lri.fr/#provers`.
[6] E. W. Dijkstra. A note on two problems in connexion with graphs. Numerische Mathematik, 1:269–271, 1959.
[7] Richard E. Bellman. On a routing problem. Quarterly of Applied Mathematics, 16:87–90, 1958.
[8] A. Shimbel. Structure in communication nets. Polytechnic Press of the Poly- technic Institute of Brooklyn, page 199–203, 1955.

Appendix

Statutory Declaration