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 [1] and Bellman-Ford [2] 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.

Specifically, our contributions are:

- Provide a concrete implementation of Dijkstra's algorithm in Idris.
- Offer a verification program for Dijkstra's algorithm written in Idris. Although there are a few holes in some minor functions in the program, we are confident to provide the complete implementation if granted more time.

The structure of this thesis 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

Software bugs are generally undesirable, especially in safety-critical and mission-critical systems. Back in 1985, errors in programs that controlled the Therac-25 radiation therapy machine were responsible for causing patience death by giving massive overdose of raidations [1]. The Northeast Blackout in 2003 due to race condition in power control systems has affected more than 50 million people in 8 states, causing an estimated loss of over 4 billion dollars [2]. In practice,

[1]Therac-25 Wikipedia page

[2](1) Northeast Blackout 2003 Wikipedia Page (2) The Economic Impacts of the August 2003 Blackout

people usually convince themselves that a program is probably correct through testing, however as Dijkstras emphasized back in 1970s, "Program testing can be used to show the presence of bugs, but never to show their absence!" [3]. Concerning the serious consequences that might be caused by software errors in real life applications, it is importnat to validate the actual behaviors of programs.

As computer programs can be considered as formal mathematical objects whose properties are subject to mathematical proofs, program verification aims to provide proofs of correctness for programs by using formal, mathematical techniques [4]. Common techniques in program verification include using proof systems, for instance the Why3 Platform [5] applies the SMT solver[3], and automatic verification techniques. Applications of program verification include the Compcert C Compiler, which is verified using machine-assisted mathematical proofs, and is considered exempt from miscompilation issues[4].

In this thesis we aim to present verification as a programming issue. We want to show that with certain functional programming languages, we can specify the expected bahaviors in function type signatures, and any incorrect function definitions will fail to type check. This not only indicates that program verification can be achieved at compilation level, but more importantly, presents a technique that enforces programmers to write programs that are correct by construction. We choose Dijkstra's and Bellman-Ford algorithms as our targets as both algorithms, or variants of them, are widely applied in many fields including computer networks and artificial intelligence.

Based on the above motivations, we choose the Idris programming language for implementing our verification program [6]. Compare to other proof management systems, the Idris type checker is based on a smaller code base, which reduces the chance of introducing unexpected bugs into our verification program. Idris is a functional programming language with dependent types, which allows programmers to provide more precise description of function's expected behaviors through its type signature. As we plan to achieve verification with type checking, this feature is essential to our verification process. In addition, the compiler-supported interactive editing feature in Idris allows programmers to inspect functions based on their type and thus to use type as guidance for writing programs, 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` expression, and interactive editing.

[3]information on SMT solver
[4]main page of Compcert C

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 a type 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
```

`CharList` is a type that stands for `List` of `Char`s, and declaring a variable of type `CharList` is the same way as we declare a variable of type `List Char`. The following example declares a variable `lisChar` of type `CharList`. `lisChar` contains the characters for the English word "hello".

```
lisChar : CharList
lisChar = 'h' :: 'e' :: 'l' :: 'l' :: 'o' :: Nil
```

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 named 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 of type `Type` 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
```

An example of calling `reverse` is provided below. The variable `nats` has type `List Nat`. When calling `reverse` on `nats`, the first argument of `reverse` denotes the type of the input list and output list, which is `Nat` in this case, then the output of `(reverse Nat nats)` is also of type `List Nat`, as specified by the type of `reverse_nats`.

```
nats : List Nat
nats = 3 :: 2 :: 1 :: Nil

reverse_nats : List Nat
reverse_nats = reverse Nat nats
```

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.

```
-- 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 need to provide the name and type of the data type starting with the keyword `data`, followed by the id and the type of the data type. 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, written as 'Z', or the
   successor of another natural number 'n', written as 'S n'
data Nat : Type where
  Z : Nat
  S : (n : Nat) -> Nat
```

Idris allows data types to be parameterized. The data type defined below shows that the type constructor `List` takes in a parameter `elem` of type `Type`, which stands for the type of elements in the list, and the type constructed is a list of elements of type `elem`. `List` type has two data constructor, `Nil` and `(::)`. `Nil` builds an empty list of type `List elem`. `(::)` append a new element `x` of type `elem` to the head of an existing list `xs` of type `List elem`, and builds a new list `x :: xs` of the same type as `xs`.

```
-- 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[7]. They allow programmers to specify certain properties of data types explicitly in their type. 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 two `Vect`s of length 3 and 4 are considered as different types, and two `Vect`s of same length but with element type `Nat` and `Char` are considered as different types. Dependent types allow programmers to obtain more confidence in a function's correctness by specifying its expected behaviors in its type. For instance, consider a function `concat` that concatenates two `Vect`, whose type signature is presented below.

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

The output value of `concat` is a vector that concatenates both input vectors, which means its length should be the sum of the length of the two input vectors, i.e., (n+m), hence `resultType` has the type `Vect (n+m)elem`. The dependent type system helps to ensure the function correctness of `concat` through the Idris type checker. By providing a function type for `concat` that specifies the length of the output `Vect`, if the definition of `concat` does not return a vector of length (n+m), `concat` would fail type check. Take the following definition of `concat` as an example.

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

The type of `concat` 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 definition of `concat` eliminates one element from the input vector `x :: xs` during each recursive call, which is not the expected function behavior. Idris gives the following error message when compiling this function definition:

```
Type checking ./Example.idr
Example.idr:6:23-34:
  |
6 | concat (x :: xs) ys = concat xs ys
  |                       ~~~~~~~~~~~~
When checking right hand side of Example.concat with expected type
        Vect (S (plus len m)) Nat

Type mismatch between
        Vect (plus len m) Nat (Type of concat xs ys)
and
        Vect (S (plus len m)) Nat (Expected type)

Specifically:
        Type mismatch between
                plus len m
        and
                S (plus len m)
```

The error message clearly indicates that the expected return type is `Vect (S (plus len m)) Nat (Expected type)`, which is a vector of length `S (plus len m)`, however the type of `concat xs ys` is `Vect (plus len m)Nat`, whose length is one less than the length of the expected type. As the return type of this definition fail to match with the return type specified in the type of `concat`, it fails to be type checked. A correct implementation of `concat` is provided below.

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

-- definition of 'plus' in Idris
total plus : (n, m : Nat) -> Nat
plus Z right        = right
plus (S left) right = S (plus left right)
```

Under the case where the first input argument is `(x :: xs)` (i.e., vector is not empty), the length of the first vector `n` should be the successor of some other natural number `n'`, i.e. `n = S n'`, then `(x :: xs)` has type `Vect (S n') Nat`, and `xs` has type `Vect n' Nat`. The `concat` function is defined by appending the head of the first input argument, `x`, to the result of `concat xs ys`. As the types of `xs, ys` are `Vect n' Nat`, `Vect m Nat`, the type of `concat xs ys` is `Vect (plus n' m)Nat`, hence the vector obtained by appending `x` to `concat xs ys` has type `Vect (S (plus n' m))Nat`. Based on the definition of `plus` in Idris (which is provided above), we see that `S (plus n' m)= plus (S n')m`, which is exactly the expected output type `Vect (plus n m)Nat`, which indicates that the above definition of `concat` type checks.

The `concat` example above illustrates how dependent types help programmers to ensure function correctness with the Idris type checker. In program verification, dependent types can be used to specify intended behaviors of a program, and thus allowing us to verify its correctness.

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 (x :: xs) v2 = x :: concat xs v2
```

Total function are defined for all possible input values and are guaranteed to terminate. Partial functions are not total, and hence might crash for some inputs. To secure the termination of programs, every function definition in Idris is checked for totality after type checking. However, due to the undecidability of the halting problem, the Idris totality checker is conservative, i.e., is never certain on whether a function is total or not. Based on the Idris Tutorial, Idris decides a function `f` is total if all of the following holds [8]:

- Cover all possible inputs
- Be well-founded — i.e. by the time a sequence of (possibly mutually) recursive calls reaches f again, it must be possible to show that one of its arguments has decreased.
- Not use any data types which are not strictly positive
- Not call any non-total functions

Specifically, `f` is considered as total if it is defined for all possible input values, for instance given an input of type `Nat`, `f` must cover the cases where it is either `Z` or the successor of another `Nat` (of the form `S n'`); and must have at least one argument that has a property, for instance its value (the `Nat` data type) or length (the `Vect` data type), that is strictly decreasing during each recursive call; the strictly positive restriction is a technical restriction that does not really concern us here, and lastly, `f` cannot call any non-total functions, otherwise `f` might fail to terminate due to the non-total functions called. To illustrate totality checking in Idris, continue with our `concat` function (the definition of `concat` below is not total):

```
concat : Vect n Nat -> Vect m Nat -> Vect (n+m) Nat
concat (x :: xs) ys = x :: (concat xs ys)
```

We use the `:total` command to check whether the above definition of `concat` is total, and we get the following message:

```
*Example> :total Example.concat
Example.concat is not total as there are missing cases
```

As `concat` is not defined for the case where the first input vector is `Nil`, hence the Idris totality checker marks `concat` as not total. If we check totality for the correct implementation of `concat` provided under the `Dependent Types` section, we see that Idris considers it as total:

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

-- totality checking result for concat
Type checking ./Example.idr
*Example> :total Example.concat
Example.concat is Total
```

`case` expressions

`case` expression can be used to inspect a data value by matching on several cases. The syntax for case expression is as follow:

```
case <test> of
    <case 1>  => <expr>
    <case 2>  => <expr>
    ...
    otherwise => <expr>
```

where `<test>` is the expression being matched on, followed by all cases in the next few lines. Consider the following example that defines a function `findNat` with `case` expressions. `findNat` checks whether a given number `n` is an element of the input vector of `Nat`s.

```
findNat : Nat -> Vect m Nat -> Bool
findNat _ Nil = False
findNat n (x :: xs) = case (n == x) of
                        True => True
                        False => findNat n xs
```

The base case is when input vector is `Nil`, which indicates that `n` is not an element in the vector. Otherwise we check whether the head of the input vector `(x ::xs)` is equal to `n` with `(n == x)`. Using `case` expression, we can match on the value of `(n == x)`, that if `(n == x)` is True, then `n` is an element of the input vector, `findNat` returns `True`; otherwise we recur on the remaining of the vector `xs` to keep searching.

The `with` Rule

In a dependently typed language, matching on the resulting value of an intermediate computation can affect what we know about other values. In program implementation and theorem proving, it is a common technique to match on intermediate value in order to obtain more information. Idris provides the `with` rule for this purpose. Consider the following example `checkEvenPrf`:

```
checkEven : Nat -> Bool
checkEven Z = True
checkEven (S n) = case (checkEven n) of
                       True => False
                       False => True


checkEvenPrf : (n : Nat) ->
               (checkEven n = True) ->
               checkEven (S n) = False
checkEvenPrf n prf = ?check
```

The checkEven function checks whether a given Nat is even or not. It returns True if the input Nat is an even number, and returns False otherwise. The checkEvenPrf function is a proof that if a natural number is even, then its successor must not be even. The type of checkEvenPrf describes the premise and conclusion of this proof: given a natural number n, if the result of calling checkEven on n is true (as specified by checkEven n = True), then the successor of n must not be even, and the result of calling checkEven on (S n) must be False, which is specified by the output type checkEven (S n)= False.

Idris allows holes in a proof which stands for incomplete parts of a program, for instance ?check in the example above is a hole. Idris allows programmers to inspect the type of holes and write functions incrementally. Inspecting the type of check we get the following:

```
*Example> :t check
  n : Nat
  prf : checkEven n = True
--------------------------------------
check : (case (checkEven n) of
           True => False
           False => True) = False
Holes: Example.check
```

The types of arguments of checkEven is presented above the dash line in the terminal output, and the expected return type, which is the type of the check hole, is presented below the dash line. The information provided by the terminal output shows that the type of check relies on the value of (checkEven n), which indicates that matching on the value of (checkEven n) with with rule might provide more insights in writing this proof, as presented below.

```
checkEvenPrf : (n : Nat) ->
               (checkEven n = True) ->
               checkEven (S n) = False
checkEvenPrf n prf with (checkEven n) proof nIsEven
  | True = ?checkT
  | False = ?checkF
```

In the checkEvenPrf definition above we use the with rule to match on the value of checkEven n, which can be either True or False (as checkEven has return type Bool). By postfix the with clause with proof nIsEven, a proof named nIsEven generated by the pattern match will be in scope. By inspecting the type of checkT under the cases where (checkEven n) is matched as True, we get the following information.

```
*Example> :t checkT
```

```
  n : Nat
  prf : True = True
  nIsEven : True = checkEven n
--------------------------------------
checkT : False = False
Holes: Example.checkF, Example.checkT
```

Notice that nIsEven is a proof of True = checkEven n generated by the pattern match directly. As the with rule matches the value of (checkEven n) to True, and based on the definition of checkEven, Idris is able to deduce that the value of checkEven (S n) should be False, and hence the expected type of checkT is False = False as presented above. When (checkEven n) is matched to False, the type of checkF is as follows:

```
*Example> :t checkF
  n : Nat
  prf : False = True
  nIsEven : False = checkEven n
--------------------------------------
checkF : True = False
Holes: Example.checkF, Example.checkT
```

As the second argument of checkEvenPrf indicates that the value of (checkEven n) should be True, Idris is able to deduce that under this case the type of prf should be (False = True), which is an absurdity, indicating that the value of (checkEven n) cannot be False. Hence we call the built-in function absurd on prf to mark that the case where (checkEven n) is matched to False is impossible. Refl is the data constructor for the equality data type (=). sym and trueNotFalse are built-in functions in Idris that helps with constructing proof with impossible cases in Idris. The complete checkEvenPrf proof is presented below.

```
checkEvenPrf : (n : Nat) ->
               (checkEven n = True) ->
               checkEven (S n) = False
checkEvenPrf n prf with (checkEven n) proof nIsEven
  | True = Refl
  | False = absurd $ trueNotFalse (sym prf)
```

On the other hand, Idris also restricts programmers from proving something that is not true. Consider the following proof checkEven_wrong.

```
  predN : Nat -> Nat
  predN Z = Z
  predN (S n) = n


  checkEven_wrong : (n : Nat) ->
                    (checkEven n = True) ->
                    checkEven (predN n) = False
  checkEven_wrong Z prf = ?caseZ
  checkEven_wrong (S pn) prf with (checkEven pn) proof pnIsEven
    | True = absurd $ trueNotFalse (sym prf)
    | False = Refl
```

The predN function calculates the predecessor of a natural number (of type Nat). The predecessor of zero Z is Z itself, and the predecessor of (S n) is n. Given the definition of predN, the function `checkEven_wrong` attempts to prove that for a natural number `n`, if `(checkEven n)` is `True`, as specified by `(checkEven n = True)`, then the predecessor of `n` must not be even, as specified by the output type `checkEven (predN n)= False`. Similar to the `checkEvenPrf` function, the implementation of `checkEven_wrong` under the case where input value `n` is (S pn) (the second case) is straightforward, however as we inspect the hole `?caseZ` in the first case where `n` is `Z`, we notice that it is impossible to complete this proof:

```
*Example> :t caseZ
  prf : True = True
--------------------------------------
caseZ : True = False
```

As the type of `caseZ` is `True = False`, which is an absurdity, and there is no information available (above the dash line is what we know for approaching the proof) for us to reach this absurdity, there is no way for us to complete this hole, that the implementation for `checkEven_wrong` can never be completed, which indicates that Idris restricts programmers from writing proofs that are not true.

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[1], and it is widely applied in many real-life applications, for instance Internet routing protocols such as the Open Shortest Path First protocol, and a variant of Dijkstra's algorithm is formulated as an instance of the best-first search algorithm in artificial intelligence.

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[9], and was published by Richard Bellman and Lester Ford, Jr in 1958 and 1956 respectively[2]. 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. As all edge weight are non-negative, and we assume a connected input graph, all edge weight should be non-negative and not equal infinity, whereas Dijkstra's algorithm initialize the distance value of all nodes in the graph (except the source node) as infinity. Based on this consideration we defined a `Distance` data type in addition to the user-defined edge weight type. `Distance` is parameterized over the user-defined weight type and can have value of either infinity, or the 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` [10].)

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

$$vv_0v_1....v_{n-1}w$$

where each adjacent nodes v_{i-1}, v_i has an edge from v_{i-1} to v_i in G. 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 $w : path(v, w) = vv_0v_1....v_{n-1}w$, the prefix of this $v-w$ path is defined as a subsequence of $path(v, w)$ that starts with v and ends with some node $w' \in path(v, w)$ (w' is a vertex in the sequence $path(v, w)$).

Definition 4.3. Length of Path
The length of a path $p = vv_0v_1....v_{n-1}w$ is the sum of the weights of all edges in p. We write:

$$length(p) = \sum weight(v_{i-1}, v_i), \forall v_{i-1}, v_i \in p \text{ where } (v_{i-1}, v_i) \in G.$$

Definition 4.4. Shortest Path
Denote $\Delta(s, v)$ as a 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
$$\forall p' \in path(s, v), length(\Delta(s, v)) = \delta(v) \leq length(p')$$

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, $unexplroed$ contains all node in g, and the distance value from s to every node $v \in g$ is ∞ except for s itself, whose distance value to s is 0, as shown below:

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

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

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:

$$\texttt{choose } u_k \in unexplored_k \texttt{ and } \forall u' \in unexplored_k, dist_k[u_k] \leq dist_k[u']$$

$unexplored_{k+1} = unexplored_k - \{u_k\}$
`for(`$\forall v \in g$`) {`

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

`}`

This implementation of Dijkstra's algorithm can be viewed as generating a matrix, where the i^{th} column in the matrix stores the value of $unexplored_i$ and $dist_i$. After calculating a matrix with n columns, the $(n+1)^{th}$ column can be calculated based on the value of $unexplored_n$ and $dist_n$ stored in the last column, i.e., the n^{th} column in the matrix. This reprensentation provides a clear recursive structure for the implementation of Dijkstra's algorithm, and the correctness of the program can be verified by proving that certain properties, for instance distance value of explored nodes stored in each column is the minimum distance value, hold for every column generated.

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:

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

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:

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

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. For any node $v \in g$, for each $u_i \in explored_{n+1}$, $n \geq 1, 1 \leq i \leq n$, $dist_{n+1}[v] \leq dist_i[u_i] + weight(u_i, v)$.

Proof. We will prove Lemma 4.4 by inducting on the number n.
Let P(n) be: for any node $v \in g$, for each $u_i \in explored_{n+1}$, $n \geq 1, 1 \leq i \leq n$, $dist_{n+1}[v] \leq dist_i[u_i] + weight(u_i, v)$.
Base Case: We shall show P(1) holds.
Based on the algorithm, $dist_1[s] = 0$, and for all node $v \in g$ other than s, $dist_1[v] = \infty$, and $explored_2$ only contains s. For node s, $dist_2[s] = 0 \leq dist_1[s] + weight(s, s) = 0$. For all node $v \in g$ other than s, we have:

$$\begin{aligned} dist_2[v] &= min(dist_1[v], dist_1[s] + weight(s, v)) \\ &\leq dist_1[s] + weight(s, v) \end{aligned}$$

Since s is the only node in $explored_2$, then the above equation directly shows that P(1) holds.

Induction Hypothesis: Suppose P(k) holds for $k > 1$. That is, for any node $v \in g$, for each $u_i \in explored_{k+1}$, $k > 1, 1 \leq i \leq k$, $dist_{k+1}[v] \leq dist_i[u_i] + weight(u_i, v)$.

Inductive Step: we shall show P(k+1) holds. That is, for $k + 1 > 1$, forall nodes $v \in g$, for each $u_i \in explored_{k+2}$, $k > 1, 1 \leq i \leq k+1$, $dist_{k+2}[v] \leq dist_i[u_i] + weight(u_i, v)$.
Suppose u_{k+1} is the node being explored during the $(k+1)^{th}$ iteration, then $explored_{k+2} = explored_{k+1} \cup \{u_{k+1}\}$. Forall node $v \in g$, we have:

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

Hence we have:

$$\begin{aligned} dist_{k+2}[v] &\leq dist_{k+1}[v]([E4.4.1]) \\ dist_{k+2} &\leq dist_{k+1}[u_{k+1}] + weight(u_{k+1}, v)([E4.4.2]) \end{aligned}$$

The induction hypothesis implies that $dist_{k+1}[v] \leq dist_i[u_i] + weight(u_i, v), \forall u_i \in explored_{k+1}$. Combining with [E4.4.1], we have:

$$dist_{k+2}[v] \leq dist_i[u_i] + weight(u_i, v), \forall u_i \in explored_{k+1}[E4.4.3]$$

Since $explored_{k+2} = explored_{k+1} \cup \{u_{k+1}\}$, then equation [E4.4.2] and equation [E4.4.3] implies that $dist_{k+2}[v] \leq dist_i[u_i] + weight(u_i, v), \forall u_i \in explored_{k+1} \cup \{u_{k+1}\} = explored_{k+2}$. P(k+1) holds. By the principle of prove by induction, P(n) holds. `Lemma 4.4` proved. □

Lemma 4.5. Assume g is a connected graph. Forall node $v \in explored_{n+1}$:

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

Proof. We will prove `Lemma 4.5` by inducting on the number of iterations.

Let P(n) be: For a connected graph g, for $n \geq 1$, forall node $w \in explored_{n+1}$: (L1) $dist_{n+1}[w] < \infty$; (L2) $dist_{n+1}[w] \leq \delta(w')$, $\forall w' \in unexplored_{n+1}$; (L3) $dist_{n+1}[w] = \delta(w)$.

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 $dist_2[s] = 0 < \infty$, then (L1) holds for P(1). 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 $dist_2[s] \leq \delta(v')$, $\forall v' \in unexplored_2$. Thus (L2) and (L3) holds for P(1). Hence P(1) holds.

Induction Hypothesis : Suppose P(i) is true for all $1 \leq i \leq k$. That is, forall $1 < i \leq k$, forall node $w \in explored_{i+1}$: (L1) $dist_{i+1}[w] < \infty$; (L2) $dist_{i+1}[w] \leq \delta(w'), \forall w' \in unexplored_{i+1}$; (L3) $dist_{i+1}[w] = \delta(w)$;

Inductive Step : We shall show P(k+1) holds. That is, forall node $w \in explored_{k+2}$, (L1) $dist_{k+2}[w] \neq \infty$; (L2) $dist_{k+2}[w] \leq \delta(w'), \forall w' \in unexplored_{k+2}$; (L3) $dist_{k+2}[w] = \delta(w)$;

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 (L1)(L2) and (L3) holds for all nodes in $explored_{k+1}$ in `Part (a)`, and `Part (b)` proves (L1)(L2)(L3) holds for u_{k+1}, so that the statements holds forall nodes in $explored_{k+2}$.

- Part(a): `WTP: After the` $(k+1)^{th}$ `iteration,` $\forall w \in explored_{k+1}$`, (L1)(L2)(L3) holds.`

  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 induction hypothesis, $dist_{i+1}[q] = \delta(q)$, and $\delta(q) \leq \delta(q'), \forall q' \in unexplored_{i+1}$.

  `Proof of (L3)`: Since for each node $q \in explored_{k+1}$, the induction hypothesis implies that $dist_{k+1}[q] = \delta(q)$, then `Lemma 3.3` imples that $dist_{k+2}[q] = dist_{k+1}[q] = \delta(q)$. (L3) holds for $explored_{k+1}$.

  `Proof of (L2)`: 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 $\delta(q) \leq \delta(q'), \forall q' \in unexplored_{k+1}$. Since proof of (L3) above shows that $dist_{k+2}[q] = \delta(q)$, then $dist_{k+2}[q] \leq \delta(q'), \forall q' \in unexplored_{k+1}$. (L2) holds for $explored_{k+1}$.

  `Proof of (L1)`: Since the induction hypothesis implies that $\forall q \in explored_{k+1}, dist_{k+1}[q] < \infty$, and the proof of (L3) above shows that $dist_{k+2}[q] = dist_{k+1}[q]$, then $dist_{k+2}[q] < \infty$. (L1) holds for $explored_{k+1}$.

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

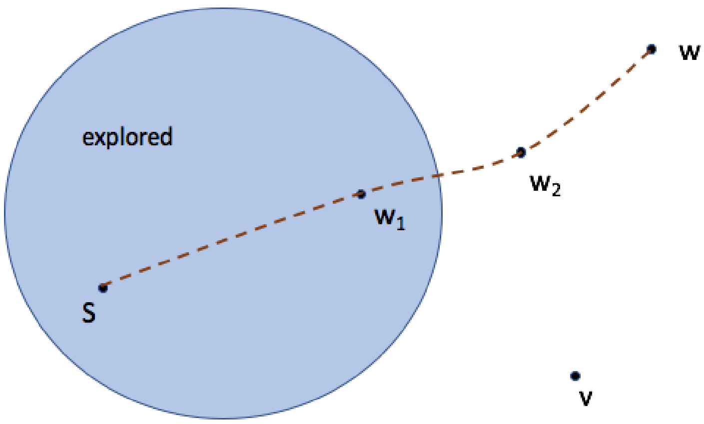
- Part(b): (L1)(L2)(L3) holds for $\{u_{k+1}\}$.
  Specifically, we want to show: (L1) $dist_{k+2}[u_{k+1}] < \infty$; (L2) $dist_{k+2}[u_{k+1}] \leq \delta(v')$, $\forall v' \in unexplored_{k+2}$, and (L2) $dist_{k+2}[u_{k+1}] = \delta(u_{k+1})$.

  1. (L1) $dist_{k+2}[u_{k+1}] \neq \infty$
     Since g is a connected graph, then s must have a path to u_{k+1}. Since u_{k+1} is the node currently being explored, then we know there must exists a $s - u_{k+1}$ path, denote as $p(s, u_{k+1})$, such any node proceeding u_{k+1} in $p(s, u_{k+1})$ are explored before u_{k+1}, i.e., in $explored_{k+1}$.
     Denote the node right before u_{k+1} in $p(s, u_{k+1})$ as u', $u' \in explored_{k+1}$. Suppose

u' is explored during the i^{th} iteration, $i < k+1$. The induction hypothesis implies that $dist_{i+1}[u'] < \infty$. Since $dist_{i+1}[u'] = min(dist_i[u'], dist_i[u'] + weight(u', u')) = min(dist_i[u'], dist_i[u']+0) = dist_i[u']$, then $dist_i[u'] < \infty$. `Lemma 4.4` implies $dist_{k+2}[u_{k+1}] \leq dist_i[u'] + weight(u', u_{k+1}]$, then it follows that $dist_{k+1}[u_{k+1}] < \infty$. (L1) holds for u_{k+1}.

2. (L2) $dist_{k+2}[u_{k+1}] \leq \delta(v'), \forall v' \in unexplored_{k+2}$

   We will prove (L2) by contradiction. Suppose there exists $w \in unexplored_{k+2}$, such that $dist_{k+2}[u_{k+1}] > \delta(w)$([E4.5.1]).
   Consider the shortest path $\Delta(s, w)$ from s to w, $\delta(w) = length(\Delta(s, w))$. Since $w \notin explored_{k+2}$, 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 $w_1 \in explored_{k+2}$. The image 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:

$$\begin{aligned}
\Delta(s, w_2) &= p(s, w_2) = p(s, w_1) + (w_1, w_2) \\
\delta(w_2) &= length(\Delta(s, w_2)) \\
&= length(p(s, w_2)) \\
&= length(p(s, w_1)) + weight(w_1, w_2) \\
&= \delta(w_1) + weight(w_1, w_2)([E4.5.2])
\end{aligned}$$

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 non-negative, then:

$$\delta(w_2) = \delta(w_1) + weight(w_1, w_2) \leq \delta(w) \text{ ([E4.5.3])}$$

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: $w_1 = u_{k+1}$
Since $\delta(w_2) = \delta(w_1) + weight(w_1, w_2) \leq \delta(w)$ and all edge weights are non-negative, then $\delta(w_1) \leq \delta(w)$. When $w_1 = u_{k+1}$, we have $\delta(u_{k+1}) \leq \delta(w)$. Since $dist_{k+2}[u_{k+1}] > \delta(w)$ and $\delta(u_{k+1}) \leq \delta(w)$, we have $\delta(u_{k+1}) < dist_{k+2}[u_{k+1}]$.
Suppose the node right before u_{k+1} in $\Delta(s, u_{k+1})$ is w_3. We know $length(\Delta(s, u_{k+1})) = length(p(s, w_3)) + weight(w_3, u_{k+1}))$, where $p(s, w_3)$ is the prefix of $\Delta(s, u_{k+1})$. Based on `Lemma 3.1`, we know $length(p(s, w_3)) = \delta(w_3)$. Hence:

$$\begin{aligned}\delta(u_{k+1}) &= length(p(s, w_3)) + weight(w_3, u_{k+1})) \\ &= \delta(w_3) + weight(w_3, u_{k+1}) \\ &< dist_{k+2}[u_{k+1}]\end{aligned}$$

i.e.

$$dist_{k+2}[u_{k+1}] > \delta(w_3) + weight(w_3, u_{k+1})([E4.5.6])$$

Based on the construction, w_2 is the first node along $\Delta(s, w)$, w_1 is right before w_2 in the path, w_3 is right before $w_1 = u_{k+1}$ in the path, then $w_3 \in explored_{k+2}$. Assume w_3 is explored during the j^{th} iteration. Then based on `Lemma 4.4`, we have:

$$dist_{k+2}[u_{k+1}] \leq dist_j[w_3] + weight(w_3, u_{k+1})([E4.5.7])$$

The induction hypothesis implies $dist_{j+1}[w_3] = \delta(w_3)$. For $dist_{j+1}[w_3]$ we have:

$$\begin{aligned}dist_{j+1}[w_3] &= min(dist_j[w_3], dist_j[w_3] + weight(w_3, w_3)) \\ &= min(dist_j[w_3], dist_j[w_3] + 0) \\ &= dist_j[w_3]\end{aligned}$$

Hence $dist_j[w_3] = \delta(w_3)$, combine with [E4.5.7], we have:

$$dist_{k+2}[u_{k+1}] \leq \delta(w_3) + weight(w_3, u_{k+1})([E4.5.8])$$

The equation [E4.5.8] contradicts with equation[E4.5.6]. Hence by the principle of prove by contradiction, (L2) holds when $w_1 = u_{k+1}$.

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:

$$\begin{aligned} dist_{i+1}[w_1] &= min(dist_i[w_1], dist_i[w_1] + weight(w_1, w_1)) \\ &= min(dist_i[w_1], dist_i[w_1] + 0) \\ &= min(dist_i[w_1], dist_i[w_1]) \\ &= dist_i[w_1] \end{aligned}$$

Since the induction 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 [E4.5.2] 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:

$$dist_{k+1}[w_2] = dist_{i+1} = \delta(w_2)([E4.5.4])$$

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 $dist_{k+1}[w_2] \geq dist_{k+1}[u_{k+1}]$. For $dist_{k+2}[u_{k+1}]$ we have:

$$\begin{aligned} dist_{k+2}[u_{k+1}] &= min(dist_{k+1}[u_{k+1}], dist_{k+1}[u_{k+1}] + weight(u_{k+1}, u_{k+1})) \\ &= min(dist_{k+1}[u_{k+1}], dist_{k+1}[u_{k+1}] + 0) \\ &= dist_{k+1}[u_{k+1}] \end{aligned}$$

Hence $dist_{k+1}[w_2] \geq dist_{k+2}[u_{k+1}]$. Combine with [E4.5.4], [E4.5.3] we have:

$$\begin{aligned} &dist_{k+1}[w_2] \geq dist_{k+2}[u_{k+1}](\\ &dist_{k+1}[w_2] = dist_{i+1} = \delta(w_2)(from[E4.5.4]) \\ &\qquad \delta(w) \geq \delta(w_2) = \delta(w_1) + weight(w_1, w_2)(from[E4.5.3]) \end{aligned}$$

Hence $\delta(w) \geq dist_{k+2}[u_{k+1}]$, which contradicts with [E4.5.1]. Hence by the principle of prove by contradiction, when $w_1 \neq u_{k+1}$, $dist_{k+2}[u_{k+1}] \leq \delta(w), \forall w \in unexplored_{k+2}$. (L2) holds for u_{k+1}.

3. (L3) $dist_{k+2}[u_{k+1}] = \delta(u_{k+1})$

   We will prove this by contradiction.
   Since (L1) proves $dist_{k+2}[u_{k+1}] \neq \infty$, then `Lemma 3.2` implies that $dist_{k+2}[u_{k+1}]$ is the length of some $s - u_{k+1}$ path, denote as p. Suppose there is a $s - u_{k+1}$ path p' that's shorter than p, i.e, $dist_{k+2}[u_{k+1}] > length(p')$([E4.5.9]). Suppose the node right before u_{k+1} in p' is v'. Then we know:

   $$\begin{aligned} &length(p') = length(p(s, v')) + weight(v', u_{k+1}) \\ &length(p') < dist_{k+2}[u_{k+1}] \end{aligned}$$

   , where $p(s, v')$ is the prefix of p' from s to v'. Hence:

   $$dist_{k+2}[u_{k+1}] > length(p(s, v')) + weight(v', u_{k+1})$$

   Based on the definition of shortest path, $length(p(s, v')) \geq \delta(v')$, then we have:

   $$dist_{k+2}[u_{k+1}] > \delta(v') + weight(v', u_{k+1})([E4.5.10])$$

   There are two cases to consider: (1) $v' \in explored_{k+2}$; (2) $v' \notin explored_{k+2}$

   Case(1): $v' \in explored_{k+2}$
   Suppose v' is explored during the i^{th} iteration. Then `Lemma 4.4` implies:

   $$dist_{k+2}[u_{k+1}] \leq dist_i[v'] + weight(v', u_{k+1})([E4.5.11])$$

   The induction hypothesis implies $dist_{i+1}[v'] = \delta(v')$, and for $dist_{i+1}[v']$ we have:

   $$\begin{aligned} dist_{i+1}[v'] &= min(dist_i[v'], dist_i[v'] + weight(v', v')) \\ &= min(dist_i[v'], dist_i[v'] + 0) \\ &= dist_i[v'] \end{aligned}$$

   Hence $dist_i[v'] = \delta(v')$. Combining [E4.5.11], we have:

   $$dist_{k+2}[u_{k+1}] \leq \delta(v') + weight(v', u_{k+1})([E4.5.12])$$

   Hence equation [E4.5.12] contradicts with equaltion [E4.5.10]. By the principle of prove by contradiction, (L3) holds when $v' \in explored_{k+2}$.

   Case(2): $v' \notin explored_{k+2}$
   Since $length(p') = length(p(s, v')) + weight(v', u_{k+1})$, $p(s, v)$ is the prefix of p' from s to v', then based on the definition of shortest path, $length(p(s, v')) \leq \delta(v')$, and thus $\delta(v') + weight(v', u_{k+1}) \leq length(p(s, v')) + weight(v', u_{k+1}) = length(p')$. Since all edge weights are non-negative, then $\delta(v') \leq length(p')$.
   Since $v' \notin explored_{k+2}$, i.e., $v' \in unexplored_{k+2}$, based on proof of (L2), $dist_{k+2}[u_{k+1}] \leq \delta(v')$. Since $dist_{k+2}[u_{k+1}] \leq \delta(v')$ and $\delta(v') \leq length(p')$, then $dist_{k+2}[u_{k+1}] \leq length(p')$, which contradicts with our assumption ([E4.5.9]). Hence by the princi-

ple of prove by contradiction, (L3) holds when $v' \notin explored_{k+2}$.
Since we have proved (L3) for both cases, then (L3) holds for P(K+1).

Since we have proved (L1)(L2)(L3) 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 4.5` 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 4.5` to the last iteration, denote as m^{th} iteration, of the algorithm, we obtained that for all nodes n in the explored list, $dist_{m+1}[n]$ 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 of `weight` that user needs 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 user-defined edge weight type and returns a type, and the data constructor `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.

As we assume the input graph is a connected graph, the value of edge weight between two adjacent nodes are considered as not infinity, whereas Dijkstra's algorithm initializes the distance value from source node to all other nodes in the graph as infinity. Consider that the value of edge weight is not infinity, and Dijkstra's algorithm requires a representation of infinity value, we define a `Distance` type to represent the distance value between two nodes. `Distance` is parameterized over the user-defined `weight` type, and the value of `Distance weight` can be either infinity or sum of `weight`s. The definition of `Distance` data type is provided below.

```
data Distance : Type -> Type where
  DInf : Distance weight
  DVal : (val : weight) -> Distance weight
```

The data constructor `DInf` builds a value of `Distance weight` that represents infinity distance, and `DVal` carries a value `val` of type `weight`, which can be the sum of one or more `weight`s.

We also defined data structures to represent graph and its main components, such as `Node`, `nodeset` and `Path`. Definition of each data types are specified below.

The `Node` type represents 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 corresponding `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 specific 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 `nodeset`. 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 definition 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 unexplored 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. Mange and 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 [11]. Although the concrete implementation of this work is unavailable, the report demonstrates the verification process. Function behaviors are specified with preconditions, postconditions, and invariants, and Jahob allows programmers to provide these specifications in high-order logic(HOL), which reduces the problem of program verification to the validity of HOL formulas.

Klasen et. al. verifies Dijkstra's algorithm with the KeY system [12], an interactive theorem prover for Java. Concrete implementations of Dijkstra's algorithm with different variants are provided, and all of them are written in Java. Similarly to the work by Mange and Kuhn, the verification process in the work by Klasen involves describing the behavior of each function with preconditions, postconditions and modifies clause. Loop invariants are specified to support the verification. A function is then verified as correct by the KeY system, with respect to its behavior specifications, if the postconditions specified hold after execution. A similar implementation is provided by 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 [13][5].

The only work found concerning verification of Bellman-Ford algorithm is by Filliâtre and Takei, who verified the Bellman-Ford algorithm with Why3, and the concrete implementation of the verification program is provided [14] [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 is presented explicitly. This reduces the chance of introducing errors into our verification program due to bugs in the proof management systems, and additionally, provides an example of how program verification can be achieved with a general-purpose programming language, and that the implementation is highly similar to that of any other programs.

8 Conclusion

References

[1] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1(1):269–271, 12 1959.

[2] Richard E. Bellman. On a routing problem. *Quarterly of Applied Mathematics*, 16(1):87–90, 1958.

[3] Edsger W. Dijkstra. *Structured Programming*, chapter 1, pages 1–82. Academic Press Ltd. London, UK, UK, 1972. Section 3 ("On The Reliability of Mechanisms"), corollary at the end.

[4] Robert S. Boyer and J Strother Moore. Program verification. *Journal of Automated Reasoning*, 1:17–23, 1985.

[5] Jean-Christophe Filliâtre et al. François Bobot. *The Why3 platform*. Toccata, Inria Saclay-Île-de-France / LRI Univ Paris-Sud 11 / CNRS, 1 edition.

[6] Christoph Herrmann Edwin Brady and Kevin Hammond. Lightweight Invariants with Full Dependent Types. In *Draft Proceedings of Trends in Functional Programming 2008*, 2008.

[7] Ana Bove and Peter Dybjer. *Dependent Types at Work*, volume 5520, pages 57–99. Springer, 2008.

[8] The Idris Tutorial. `http://docs.idris-lang.org/en/latest/tutorial/index.html#the-idris-tutorial`, 2017.

[9] Alfonso Shimbel. Applications of matrix algebra to communication nets. *BULLETIN OF MATHEMATICAL BIOPHYSICS*, 13:165–18, 1951.

[10] Susanna S. Epp. *Discrete Mathematics with Applications*. Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 4 edition, 2010.

[11] R. Mange and J. Kuhn. Verifying dijkstra algorithm in Jahob. Student Project at Ecole Polytechnique Fédérale de Lausanne, 2007.

[12] Volker Klasen. *Verifying Dijkstra's Algorithm with KeY*. PhD thesis, Universitat Koblenz-Landau, 3 2010.

[13] Jean-Christophe Filliâtre. Dijkstra's Shortest Path Algorithm. Toccata, 2007.

[14] Jean-Christophe Filliâtre Yuto Takei. A proof of Bellman-Ford algorithm. Toccata.

Appendix

Statutory Declaration