VERIFICATION OF DIJKSTRA'S ALGORITHM 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

Foremost, I would like to express my sincere gratitude to my advisor, Professor Richard A. Eisenberg at Bryn Mawr College, who offered invaluable guidance for my research, and has been a constant source of support, encouragement, and inspiration during my college years. I truly enjoyed and learned a lot from every conversation, every meeting we had, and I could not have imagined a better advisor and mentor during my Undergraduate study.

I take this opportunity to thank all of the Department faculty members for their help and support. My sincere thanks also goes to my fellow senior CS majors for providing great suggestions in improving my thesis. I owe a debt of gratitude to my dear friends, especially Rachel Xu, Christa Schmidt, Feichi Hu, Patricia Sanchez, Jiaping Wang, Zhanpeng Wang, and TianMing Xu. I was extremely fortunate to become friends with all of you, and thank you for making my college life amazing and memorable.

Finally, I would like to thank my family. I must express my profound gratitude towards my parents, for their unrequited love and unceasing encouragement. This achievement would not have been possible without them.

Abstract

Program verification is the process of proving program correctness with formal, mathematical methods. In spite of the significance of validating program behaviors, program verification is seldom involved in software development and other real-life applications. This thesis offers a verification of Dijkstra's algorithm [1] implemented with the Idris Programming Language [2], aiming to show how verification can be approached as a programming issue. We first provide a detailed mathematical correctness proof for Dijkstra's algorithm, including lemmas that are generally assumed by most proofs on Dijkstra's. We then offer demonstrations on the construct of our algorithm implementation and verification program. During this verification process, we notice a parallel between our mathematical proofs and the corresponding Idris implementations, which indicates that direct translation of clear, step-by-step mathematical proofs can be a feasible approach in program verification. We are also aware of certain incomplete proofs in our program, and recognize a few downsides of our design and potential errors in Idris that are partly accountable for this, however we are confident to provide the complete implementation with more time granted.

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 algorithm [1] is one of the most well-known single-source shortest path algorithms, and is implemented in various fields including network protocols and aritificial intelligence.

Given the importance of Dijkstra's in real-life applications, we are interested in verifying the implementation of Dijkstra's algorithm. We first provide concrete implementation for Dijkstra's, and then define functions with precise type signatures which carry specifications that should hold for the correct implementation, 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, which is available online this (`https://github.com/EileenFeng/algorithm_verification`). Although the proof of some lemmas are incomplete, we are confident that we can 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 the Idris functional programming language, follows up by briefl introduction on Dijkstra's algorithm. 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 mathematical proof of Dijkstra's, 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. In 1985, errors in programs that controlled the Therac-25 radiation therapy machine were responsible for causing patient death by giving massive overdose of radiations [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, people

[1] Therac-25 Wikipedia page
[2] (1) Northeast Blackout 2003 Wikipedia Page (2) The Economic Impacts of the August 2003 Blackout

usually convince themselves that a program is probably correct through testing, however as Dijkstra 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 important 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 multiple SMT solvers [6] [7] [8], 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 [9].

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 behaviors 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 algorithm as our target as since it is widely applied in many fields, such as artificial intelligence.

Based on the above motivations, we choose the Idris programming language for implementing our verification program [2]. Compared to other proof management systems, the Idris type checker is based on a smaller code base. Most proofs in our work will be implemented from scratch, 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. Idris is a functional programming language with dependent types, which allows programmers to provide more precise description of functions' 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.

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 as declaring 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
```

Functions

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 constructors, `Nil` and `(::)`. `Nil` builds an empty list of type `List elem`. `(::)` appends 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
```

The `Fin` Type

We introduce the `Fin` data type here as it is used in our Dijkstra's implementation. The definition of `Fin` data type is provided below.

```
data Fin : (n : Nat) -> Type where
    FZ : Fin (S k)
    FS : Fin k -> Fin (S k)
```

Given a natural number `n`, `Fin n` captures a finite set of natural numbers that is greater than or equal to zero, and strictly less than `n`. For instance `Fin 3` is the set of natural numbers from 0 to 2. The type of the data constructors `FZ` and `FS` restricts that the input `Nat n` must be the successor of some other natural number `k` (i.e., `n` is greater than `Z`), hence we cannot construct a value of type `Fin Z`. `FZ` stands for the first element in the finite set, and `FS n` stands for the $(n+1)^{th}$ element in the set. In our Dijkstra's implementation, we use `Fin` type values as the indices for accessing elements from a list, which helps to ensure safe indexing.

Dependent Types

Dependent types are types that depend on elements of other types[10]. 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` is `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 fails 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 `concat` function that concatenates two vectors, to define `concat`, we need to provide definitions on all possible values of `Vect`, which can either be `Nil`, i.e., a vector of length zero, or a non-empty vector of the pattern `(x :: xs)`.

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 [11]:

- 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 value of `(checkEven n)` might effect the type of `check`, 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 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.

4 Overview of Dijkstra's Implementation and Proof of Correctness

This section provides an overview of our Dijkstra's implementation and mathematical proof of correctness. Section 4.2 provides definitions of key concepts used in our work, and Section 4.4.1 presents the lemmas of our proof.

4.1 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 particular 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.

4.2 Definition

Our implementation and correctness proof are based on the following definitions of key concepts used in Dijkstra's algorithm. The following definitions are inspired by Epp [12].

Definition 4.1. Path
A path from node v_0 to v_n is a finite sequence of adjacent vertices of G, which does not contain any repeated edge or vertex. A path from v to w has the form:

$$v_0 v_1 .... v_{n-1} v_n$$

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_0 to v_n as $path(v_0, v_n)$.

Definition 4.2. Prefix of Path
Given a path p from node v_0 to v_n, i.e., $p = v_0 v_1 .... v_{n-1} v_n \in path(v_0, v_n)$, the prefix of this $v_0 - v_n$ path is defined as a subsequence of p that starts with v_0 and ends with a node v_i for some $0 \leq i \leq n$ (v_i is a vertex in the nodes sequence of p).

Definition 4.3. Length of Path
The length of a path $p = v_0 v_1 .... v_{n-1} v_n$ is the sum of the weights of all edges in p. We write:

$$length(p) = \sum weight(v_{i-1}, v_i), \forall 0 \leq i \leq n, v_{i-1}, v_i \in p \text{ where } (v_{i-1}, v_i) \in G.$$

Definition 4.4. Shortest Path
Denote $\Delta(s, v)$ as an arbitrary choice of a shortest path from s to v, and denote $\delta(v)$ as the length of $\Delta(s, v)$. $\Delta(s, v)$ must fulfills:

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

4.3 Pseudocode

We denote (u, v) as an edge from node u to v, $weight(u, v)$ as the weight of edge (u, v). If (u, v) is not an edge in the graph, then define $weight(u, v) = \infty$. 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 set of unexplored nodes, and $dist$ as the set of distance values[3] 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 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, let u_i be the node being explored at the i^{th} iteration, and let $dist_i$, $unexplored_i$ be 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` $u_k \in unexplored_k$ `s.t.` $\forall u' \in unexplored_k, dist_k[u_k] \leq dist_k[u']$
$unexplored_{k+1} = unexplored_k - \{u_k\}$
`for`$(\forall v \in g)$ $\{$

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

$\}$

[3]For convenience purpose, in this thesis we denote the 'distance value' for a node 'n' in a graph 'g' as the distance from the source node to n in 'g'

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 representation 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.

4.4 Proof of Correctness

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

4.4.1 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:

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

Denote 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 a 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. Forall node $v \in g$, $n \geq 0$, 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, $n \geq 0$, 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(0) 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(0) holds.

Inductive Hypothesis : Suppose $\forall i, 1 \leq i \leq k$, P(i) holds. That is, for all nodes $v \in g$, if $dist_{i+1}[v] \neq \infty$, then $dist_{i+1}[v]$ 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:

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

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 $dist_{k+1}[u_{k+1}] = dist_{k+2}[v] - weight(u_{k+1},v) \neq \infty$. 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 $dist_{i+1}[v] = \delta(v)$, then $\forall j > i$, $dist_{j+1}[v] = dist_{i+1}[v] = \delta(v)$.

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:

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

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, for $i > 0$, if $dist_{i+1}[v] = \delta(v)$, then for the $(i+k)^{th}$ iteration, $k \geq 1$, $dist_{i+k+1}[v] = dist_{i+1}[v] = \delta(v)$.

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 $n \geq 1$, for all u_i explored during the i^{th} iteration for some $1 \leq i \leq n$ ($u_i \in explored_{n+1}$), $dist_{n+1}[v] \leq dist_i[u_i] + weight(u_i, v)$.

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 4.3 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 $dist_{i+1}[u'] = min(dist_i[u'], dist_i[u'] + weight(u', u')) = min(dist_i[u'], dist_i[u'] + \infty) = dist_i[u']$, 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 $w_1 \in explored_{k+2}$. Figure 1 below illustrates this construction.

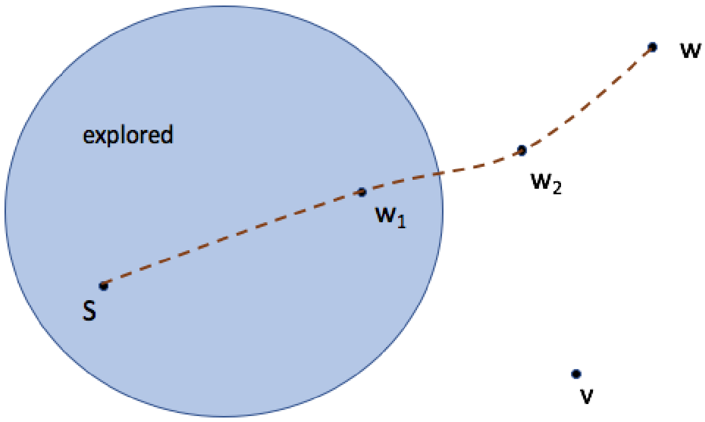

Figure 1: Lemma 4.5 Proof Construction

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 definitions in Section 4.2, $p(s, w_1)$ 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 4.1, $p(s, w_1)$ 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 4.1 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:

$\delta(w_2) = \delta(w_1) + weight(w_1, w_2) \leq \delta(w)$ ([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 4.1, 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:

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

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 (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:

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

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 4.2, $dist_i[w_2]$ 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 4.3, 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:

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

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

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

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 4.2 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:

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

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 (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. Thus Lemma 4.5 holds. □

4.4.2 Proof of Termination

Proof. As the algorithm goes through each node in the graph exactly once when exploring one particular node, and as the size of the unexplored list decreases by one during each iterations, the algorithm is guaranteed to terminate. □

4.4.3 Proof 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

5.1 Data Structures

5.1.1 The `WeightOps` data type

Our implementation of Dijkstra's algorithm allows user-defined edge weight type, with a `WeightOps` data type specifying the operations and properties of the edge weight type that user needs to provide. `WeightOps` is similar to a Java interface for the user-defined weight type, except that it also includes properties of the weight type besides the operators.

Below presents part of the definition of `WeightOps`.

```
using (weight : type)
  record WeightOps weight where
    constructor MKWeight
    -- zero value of weight
    zero : weight
    -- greater than or equal to
    gtew : weight -> weight -> Bool
    -- equality
    eq : weight -> weight -> Bool
    -- addition
    add : weight -> weight -> weight
    ...
    triangle_ineq : (a : weight) ->
                    (b : weight) ->
                    gtew (add a b) a = True
    ...
    addComm : (a : weight) ->
              (b : weight) ->
              add a b = add b a
```

`WeightOps` is defined as a record data type, which allows programmers to collect several values (referred as record's fields) together. `WeightOps` is parameterized over the user-defined edge weight type `weight`. The `MKWeight` constructor takes in all the fields and build a `WeightOps weight` type. The field name can be used to access the field value. For instance given a value `ops` of `WeightOps weight` type, `add ops` will gives the addition operator for the `weight` data type.

The `zero` field stands for the zero value for the `weight` type, and `gtew`, `eq`, `add` are basic operators of `weight`: `gtew` is the greater than or equal comparator, `eq` is the operator for checking equality, and add is the operator for calculating addition. The triangle_ineq field in WeightOps ensures that the value of weight data type can only be non-negative. Given any two values a, b of type weight, triangle_ineq specifies that the sum of a, b is greater than or equal to either of them, which guarantees that both a, b have non-negative values. The remaining fields in WeightOps are required for Dijkstra's implementation and verification, for instance the addComm property (which states commutativity for the add operator) is later applied in one of the helper functions in proving Lemma 4.1 in Section 5.3.1.1.

To provide a concrete example of constructing a WeightOps data value, we present the definition of the natOps variable below.

```
-- a 'WeightOps' for 'Nat'
natOps : WeightOps Nat
natOps = MKWeight Z gte (==) plus
                   ...
                   nat_tri ... plusCommutative
```

We have eliminated a few fields in the definition of natOps as they do not concern us here. The type of natOps indicates that this record collects operators and properties for the Nat data type. The first argument passed into MKWeight for constructing natOps is Z, which is the zero value for Nat and corresponds to the zero field in the definition of WeightOps data type. The next few arguments of MKWeight provides the greater than or equal, equal, and plus operators for Nat. The nat_tri function states triangle inequality for Nat, which ensures that the there is no negative values of the Nat data type, and plusCommutative is a built-in function in Idris that states commutativity for the plus operator of Nat. We can also define WeightOps for other weight types, for instance the Double type, or even Char type with user-defined operators for comparison, add, and checking equality etc., and the process of constructing WeightOps data values over other weight types is similar to the definition of natOps variable provided above.

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. Based on this consideration, 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 is either infinity, or sum of weights. 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 is the sum of one or more weights. Arithmetic operators for the Distance weight type is defined based on operators of weight.

5.1.2 Data Types for Node, nodeset, and Graph

The design of our data structures for a graph and its components are inspired by the adjacency list representation of a graph. We define the size of a graph as the number of the nodes in the graph. A graph of size gsize is defined to contain a Vect of gsize nodesets, where nodeset stands for the adjacent list for each node in the graph, and that each node in the graph carries the index for accessing its list of neighboring nodes from the graph. In other words, if we enumerate the set of nodes in a graph by natural numbers starting from 0, then the `Vect` of `nodesets` is ordered in a sense that, the first element in this `Vect` is the `nodeset` for the node numbered 0, and the second element is the `nodeset` for the node numbered as 1 ... etc. Figure 2 illustrates this construction.

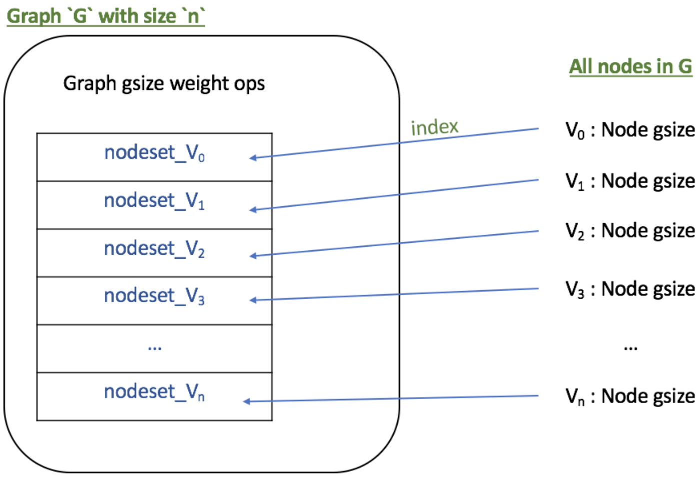

Figure 2: Graph Data Types Illustration

The definition of `Node`, `nodeset` and `Graph` data structure are presented below.

```
-- definition of Node
data Node : (gsize : Nat) -> Type where
  MKNode : (nodeVal : Fin gsize) -> Node gsize

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

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

A `nodeset` is a `List` of pairs of type `(Node gsize, weight)`, where the first element of the pair is the neighboring node, and the second element is the edge weight. For instance, if the `nodeset` of a node `v` in graph `g` contains a pair `(w, edge_w)`, this means node `v` has an edge to `w` in `g`, with edge weight `edge_w`. As the edge weight type is user defined, the `Graph` data type is parameterized over the edge weight type `weight`, and index by the size of the graph `gsize`(which means there are `gsize` nodes in this graph). Operators and properties of `weight` are carried in the `Graph` data type by the `ops` parameter. `edges` is a `Vect` of nodesets with length `gsize`, as each node in the graph has its corresponding `nodeset`. Since each node carries the index for accessing its `nodeset` in the graph, a `Node` type is indexed by the graph size `gsize`, and defined to carry a value of type `Fin gsize`. Relating to what we discussed above, to enumerate all nodes in the graph, the `Node` numbered as 0 carries the value `FZ`, and the `Node` numbered as 1 carries the value `(FS FZ)` ... etc.

The type `Fin gsize` captures the set of natural numbers within the range from 0(inclusive) to `gsize`(exclusive), with size `gsize`, which means that are only `gsize` possible values of the type `Fin gsize`. Such construction ensures that, for a graph `G` of type '`Graph gsize weight ops`' (graph size is `gsize` and edge weight type is `weight`), the type `Node gsize` indicates that there are indeed `gsize` number of nodes in the graph `G`. More importantly, as our implementation uses the value carried by each `Node` to index its `nodeset` in the graph, as long as we restrict the type of each node in `G` as `Node gsize`, then the value carried by each node would have the type `Fin gsize`, which is guaranteed to be a bounded index for the vector `edges` in `G`.

Based on the above construction, a node `m` is considered as adjacent to a node `n` in a graph `g` if `m` is in the `nodeset` of `n`. The definition of adjacent nodes is provided below.

```
adj : (g : Graph gsize weight ops) ->
      (n, m : Node gsize) ->
      Type
adj g n m = (inNodeset m (getNeighbors g n) = True)
```

The `getNeighbors` function takes in a graph `g` and a node `n`, and gets the `nodeset` of `n` in `g`, and the `inNodeset` function takes in a node and a `nodeset`, and returns true if the input node is in the input nodeset, returns false otherwise. `adj` is defined as a predicate on the input graph `g` and two nodes `m, n`. The type '`adj g n m`' states that the node `m` is adjacent to node `n` in `g` as `m` is an element of the `nodeset` of `n` in `g`.

5.1.3 `Path` and shortest `Path`

A path in a graph is defined as a sequence of non-repeating nodes, where each two adjacent nodes have an edge in this graph. A path can contain only one node, as specified by the `Unit` data constructor below, or multiple nodes, as the `Cons` data constructor allows a new path to be constructed from an existing path. Specifically, given a path from node `s` to `v`, if `n` is an adjacent to `v` (`adj g v n` specifies 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
```

To implement a shortest path in a graph, recall in section 4.2, we define the length of a path as the sum of the weights of all edges in the path, and define a shortest path as follows:

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 fulfill:

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

The above definition specifies that given a shortest path $\Delta(s, v)$, the length of $\Delta(s, v)$ is smaller than or equal to the length of any other $s - to - v$ path in the graph. We then provide the following implementation of shortest path based on the above definition.

```
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
```

The statement stated by the return type of `shortestPath` is highly similar to our mathematical definition of shortest path above. Specifically, given a graph `g`, and a path `sp` from node `s` to `v` in `g`, the definition of `shortestPath` specifies that, given any path `lp` from `s` to `v` in `g`, the length of `lp` must be greater than or equal to the length of `sp`. `dgte` is the greater than or equal to operator for `Distance` data type.

5.1.4 The `Column` data type

As we mentioned back in section 4.3, our implementation viewed Dijkstra's algorithm as generating a matrix, where each column in the matrix represents one state of the algorithm, we define a `Column` data type for this purpose. The definition of `Column` type is provided below.

```
    data Column : (len : Nat) -> (g : Graph gsize weight ops) -> (src
 : 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
```

The `Column` data type takes in the input graph `g`, the source node `src`, the number of unexplored nodes `len`, which is also the length of `unexp`, the vector of unexplored nodes. `dist` is a `Vect` of distance values from source for all nodes in the graph, with length `gsize` as there are `gsize` number of nodes in the graph. As discussed in Section 5.2.1, the construction of `dist` ensures that the elements are ordered the same way as `edges` in the `Graph` structure above, i.e., the first element in `dist` is the distance of the `Node` that carries `FZ`, and the second element in `dist` is the distance of the `Node` that carries '`FS FZ`' ... etc., hence we can again use the value carried by each `Node` to index its corresponding distance value stored in the `Column` with a `nodeDistN` function, which is mentioned in the implementation of Lemma 4.2 in Section `lemma2V`. The `Column` type is indexed by the number of unexplored nodes. As Dijkstra's requires a source node for running the algorithm, the type `Column` is also dependent on the input graph as well as the source node `src`.

Such definition of `Column` data type provides enough information for us to calculate a new column in the matrix, as the `unexp` and `dist` vectors provides enough information for calculating the current unexplored node with minimum distance value and generating the new distance vector with updated distance values for all nodes in the graph. Our implementation of Dijkstra's algorithm has a recursive structure, and generates a new `Column` with one less unexplored node during each recursive call. With an input graph of size `gsize`, the first column should have length `gsize` as all nodes are unexplored, and the last column generated contains an empty vector for unexplored nodes, and a vector of the minimum value from source to all nodes in the graph.

5.2 Implementation of Dijkstra's Algorithm

Our implementation of Dijkstra's algorithm can be viewed as generating a matrix, where one column of the matrix represents one state during the execution of the algorithm. Each column stores the original input graph, source node, a vector of currently unexplored nodes, and a vector of current distance values for all nodes in the graph, and a new column is calculated based on the value stored in the last column generated. The `Column` data type in the data structure section (section 5.1.4) is defined for this column representation. However, since the calculation of a new column does not requires all previous columns calculated, and in order to simplify the data structures, the implementation does not maintain the whole matrix, rather, only one variable of the `Column` data type is maintained to store the last column calculated.

Even though the whole matrix is not presented, we can still visualize this implementation as generating a matrix representation of Dijkstra's algorithm, where the columns shows how distance value of all nodes in the graph is gradually updated, hence in the following sections we still refers to this matrix representation of Dijkstra's algorithm, based on the above clarification that the actual matrix is not presented in the implementation. Eliminating the matrix structure not only reduces some redundancy in our implementation, but also allows us to verify Dijkstra's algorithm by proving properties over each `Column` calculated, which provides a clear structure for our verification program.

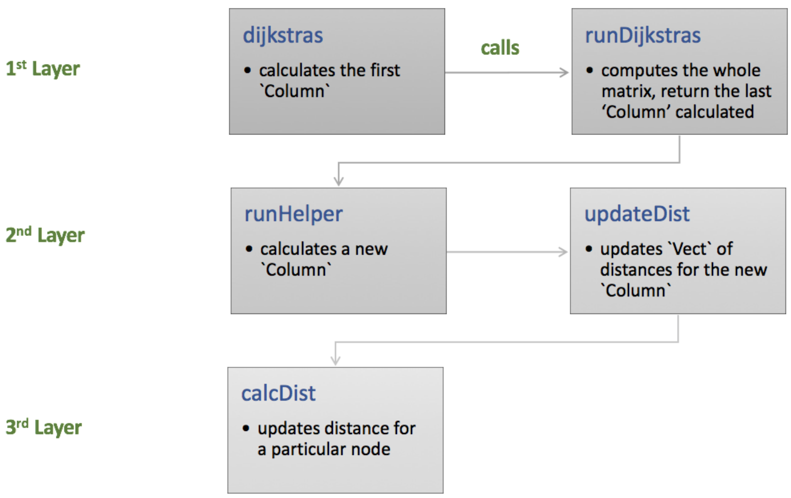

Figure 3: Structure of Dijkstra's Implementation

As illustrated by Figure 3 above, the implementation can be divided into three layers, where each layer breaks down the calculation to deal with a smaller structure. Specifically, the first layer calculates the whole matrix representation by calling function from the second layer. The second layer is responsible for generating a new `Column` data based on the last `Column` calculated, where the updated distance value for each node in the graph is calculated by the third layer. The remaining of this section provides more details on our three layers of calculation.

5.2.1 `dijkstras` and `runDijkstras`

The first layer involves two functions, `dijkstras` and `runDijkstras`, where `dijkstras` takes in the input graph and the source node, generate the first `Column` of the matrix representation, and calls `runDijkstras` on the first `Column` to calculate the whole matrix and returns the last column generated. The definition of `dijkstras` and `runDijkstras` are provided below.

```
   -- 'runDijkstras' generates the whole matrix and returns the last
'Column' calculated
   runDijkstras : {g : Graph gsize weight ops} ->
                  (cl : Column len g src) ->
                  Column Z g src
   runDijkstras {len = Z} {src} cl = cl
   runDijkstras {len = S l'} cl@(MKColumn g src (S l') _ _ ) =
runDijkstras $ runHelper cl


   -- 'dijkstras' function creates the first 'Column' 'cl' and call '
runDijkstras' on 'cl'
   dijkstras : (gsize : Nat) ->
               (g : Graph gsize weight ops) ->
               (src : Node gsize) ->
               (nadj : ((n : Node gsize) ->
                     inNodeset n (getNeighbors g n) = False)) ->
               (Vect gsize (Distance weight))
   dijkstras gsize g src nadj {weight} {ops} = cdist $ runDijkstras
cl
     where
       nodes : Vect gsize (Node gsize)
       nodes = mkNodes gsize
       dist : Vect gsize (Distance weight)
       dist = mkdists gsize src ops
       cl : Column gsize g src
       cl = (MKColumn g src gsize nodes dist)
```

The `dijkstras` function takes in the size of graph `gsize`, the input graph `g`, the source node `src`, and a function `nadj` stating that any node in the graph cannot be in its own `nodeset`. `nadj` ensures that when constructing a `Path` data with the `Cons` constructor, it is not possible to append a node `n` to itself, as `adj g n n` cannot hold for any node in the input graph(definitions of `adj` and `Path` is illustrated in section 5.1.3). `dijkstras` function constructs the first `Column` `cl` and calls the `runDijkstras` on `cl`. `runDijkstras` traverses all unexplored nodes and returns the last `Column` calculated, which should contain an empty vector of unexplored nodes. The `dijkstras` function then returns the vector of distance values in the `Column` returned by `runDijkstras`, which contains the minimum distance values for all nodes in the graph.

The `mkNodes` function in the `where` clause takes in a `Nat` and generates a vector `nodes` with type '`Vect gsize (Node gsize)`', where the '`Fin gsize`' value carried by each node in `nodes` is increasing in order. Specifically, suppose `gsize` is not zero, i.e., `gsize = S n`, then the first node in nodes FZ with type 'Fin gsize', the second node carries 'FS FZ', ..., and the last element in nodes carries a value of type Fin gsize that captures the natural number n, which the largest Nat value that falls into the range of Z to n. Since we enumerate all nodes in the graph by natural numbers starting from 0 (as mentioned in section 5.1.2), the vector generated by mkNodes contains all nodes in the input graph. Since initially all nodes are unexplored, when constructing the first Column cl, the dijkstras function calls the mkNodes function to generate the initial vector of unexplored nodes. The mkdists function generates the initial vector of distance values for all nodes in the graph(distance value for all nodes are infinity except for the source node, which is 0), in the same ordering as the vector generated by mkNodes. In the definition of dijkstras, both mkNodes and mkdists return a vector of length gsize (named as dists and nodes correspondingly), which ensures that the i^{th} element dists is the initial distance value for the i^{th} node in nodes. Later paragraphs shows how this constructions gives a clear recursive structure for the implementation of Dijkstra's algorithm.

The runDijkstras algorithm takes in a parameter cl of type 'Column len g src', traverses all unexplored nodes in cl(if there is any), and returns a value of type 'Column Z g src', which is a Column with an empty list of unexplored nodes. runDijkstras is defined recursively: if the input value cl contains an empty vector of unexplored nodes, then simply returns cl, otherwise we extracts the unexplored nodes in cl with minimum distance value, calculate a new Column with updated vectors of unexplored nodes and distance values, and recurs on the new Column calculated. The calculation of new Column is completed with the runHelper function, which is elaborated in the following.

5.2.2 runHelper and updateDist

The second layer includes two functions, runHelper and updateDist, which calculate a new Column value based on the last column generated. runHelper takes in a column with non-empty unexplored list type and returns the new Column calculated with one less unexplored nodes, and the updateDist function calculates the updated distance vector for the new Column. The implementation of runHelper and updateDist are provided below.

```
    -- 'updateDist' updates the 'Vect' of distance values based on '
min_node'
    updateDist :   (g : Graph gsize weight ops) ->
                   (min_node : Node gsize) ->
                   (min_dist : Distance weight) ->
                   (nodes : Vect m (Node gsize)) ->
                   (dist : Vect m (Distance weight)) ->
                   Vect m (Distance weight)
    updateDist g min_node min_dist Nil Nil = Nil
    updateDist g min_node min_dist (x :: xs) (d :: ds)
      = (calcDist g min_node x min_dist d) :: (updateDist g min_node
min_dist xs ds)

    {- 'runHelper' get the current min node with 'getMin'
      and calls 'updateDist' to generate the updated 'Vect' of
distance values 'newds' -}
    runHelper : {g : Graph gsize weight ops} ->
                (cl : Column (S len) g src) ->
                Column len g src
```

```
    runHelper cl@(MKColumn g src (S len) unexp dist) {gsize} {weight}
  {ops}
       = MKColumn g src len
              (deleteMinNode min_node unexp (minCElem cl)) newds
       where
         min_node : Node gsize
         min_node = getMin cl
         ...
         newds : Vect gsize (Distance weight)
         newds = updateDist g min_node min_dist (mkNodes gsize) dist
```

The input value `cl` of `runHelper` has type `Column (S len) g src`, which is a `Column` with non-empty vector of unexplored nodes, as specified by `S len` in the type. `runHelper` extracts the unexplored node with minimum distance value from the unexplored vector in `cl`, and calculates a new column with the updated unexplored and distance vectors. The currently unexplored node with minimum distance value is named as `min_node` and calculated by calling `getMin cl` under the `where` clause. The return value of `runHelper` has type `Column len g src`, indicating the unexplored vector in the new `Column` has one less element than that `cl`. The `deleteMinNode` is responsible for calculating the updated vector of unexplored nodes based on that of `cl`, but with `min_node` removed. `deleteMinNode` requires a proof that the targeting node to be removed is in the input vector, as specified by `(minCElem cl)`.

The updated vector of distance values for the new `Column`(named as `newds` in definition of `runHelper`) is calculated by `updateDist`, which takes in a graph `g`, the minimum node `min_node` and its distance value `min_dist`, and vectors of nodes and distances, both have the same length. Notice that in `runHelper`, `updateDist` is called on the vector generated by `mkNodes gsize` (which contains all nodes in the graph) rather than the vector of unexplored nodes, and the initial input distance vector of `updateDist` is calculated by `mkdists` and passed down from the `dijkstras` function. The definition of `mkNodes` and `mkdists` mentioned previously indicates that the first element of `dists` is the distance value corresponds to the first element of `nodes`, which allows `updateDist` to call `calcDist` on the heads of `nodes` and `dists` (`x` and `d` respectively), and recur on the corresponding elements in both `Vect`s to update the distance value for every node in the graph, provided that the elements orderings in `nodes` and `dists` remain the same during each recursive step. Since `updateDist` never changes the ordering of the input nodes vector, and during each recursive step, the new distance value for the current head node `x` calculated is append to the result of calling `updateDist` on the remaining nodes `xs` and their distance values `ds`, the element ordering of the distance vector is also unchanged. This definition of `updateDist` again provides a clear recursive structure in implementing certain lemma proofs in our verification program, which is expanded in more details in Section 5.3.1.3.

5.2.3 `calcDist`

The third layer contains the function `calcDist`, which is called by `updateDist` to calculate the updated distance value for one specific node in the graph. Below presents the implementation of `calcDist`.

```
    -- 'min' calculates the minmum between two distance values
    min : (ops : WeightOps weight) ->
          (m : Distance weight) ->
```

```
      (n : Distance weight) ->
      Distance weight
min ops m n = case (dgte ops m n) of
                True  => n
                False => m

{-
  'calcDist' compares 'cur_dist' with the distance of the
    path from source to 'cur' passing 'min_node', and returns
    the smaller one
-}
calcDist : (g : Graph gsize weight ops) ->
           (min_node : Node gsize) ->
           (cur : Node gsize) ->
           (min_dist : Distance weight) ->
           (cur_dist : Distance weight) ->
           Distance weight
calcDist g min_node cur min_dist cur_dist
  = min ops cur_dist (dplus ops (edgeW g min_node cur) min_dist)
```

Given the input graph g, the current node being explored(named as min_node), the distance value of min_node(named as min_dist), a node cur and its distance value cur_dist, calcDist compares the distance value from source node to cur through min_node in g with cur_dist and returns the smaller value. The distance value of cur passing min_node is calculated by adding min_dist with the weight of edge from min_node to cur. If there is no edge between min_node and cur in g, the edge weight will be infinity, which is already greater than or equal to the original distance of cur, then cur_dist will be returned by calcDist in this case. Otherwise, calcDist returns the minimum value between cur_dist, and the sum of min_dist and weight of edge from min_node to cur, which is calculated by calling the the min function.

5.3 Verification of Dijkstra's Algorithm

Our verification of Dijkstra's algorithm is based on and has a similar structure as the implementation in section 5.2. Instead of proving Dijkstra's correctness based on the dijkstras and runDijkstras functions directly, we approach the verification by proving that certain properties maintain for each new Column value generated by every call to the runHelper function. Specifically, since the Column structure carries information on the unexplored nodes and distance values calculated for all nodes in the graph, we can re-state Lemma 4.2 to Lemma 4.5 in the mathematical proof of Dijkstra's correctness(in Section 4.4) as properties on Column, and prove that these properties are preserved after calling runHelper. As runHelper is called by the runDijkstras function, the implementation of our verification follows the same structure by defining a function that recursively applies the above proof of properties preservation, and shows that same properties also hold for the last Column value calculated, i.e., the value returned by the runDijkstras function, which verifies the correctness of Dijkstra's algorithm.

In the following sections, we first provide proofs of lemmas that state the properties preservation of each new Column generated by runHelper, and then present the functions that directly verify the correctness of Dijkstra's algorithm. As the implementation of all proofs are highly complicated and involves significantly amount of details, the following only elaborates on the implementation of two lemma proofs for the purpose of presenting some techniques on how proofs are approached in our verification, and discuss on the types of other lemmas instead. As this thesis aims to verify Dijkstra's algorithm with the Idris type checker, the types of proofs should provide sufficient information on our verification program.

5.3.1 Lemmas

The mathematical proof of Dijkstra's algorithm in section 4.4 includes five lemmas, however in implementing our verification program, we found it easier to approach by merging Lemma 4.4 into Lemma 4.5 as one of its statements. Lemma 4.1 to 4.5 are defined in order, meaning that the proof of Lemma 4.2 is built on Lemma 4.1, and proof of Lemma 4.3 is built on Lemma 4.1 and Lemma 4.2 etc. The implementation of Lemma 4.5 (function `l5_spath`) is directly applied in verifying Dijkstra's correctness, and implementations of Lemma 4.1 to Lemma 4.4 are helper proofs for proving Lemma 4.5.

The following first presents the types for all lemmas of our verification program, and then elaborate on the implementation of one of the lemma proofs, in order to provide more insights into how we approach proofs generally. We choose to expand on the proofs of Lemma 4.1 (which corresponds to function `l1_prefixSP`) and Lemma 4.3 (which corresponds to the `l3_preserveDelta` function), as compare to other lemma proofs, proof of Lemma 4.3 involves less details but presents enough information on our techniques in implementing proofs.

5.3.1.1 Lemma 1 - `l1_prefixSP`

Lemma 4.1 states that the prefix of a shortest path is also a shortest path. In section 4.2, we provide the following definition for the prefix of a path.

Definition 5.1. 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 $path(v, w)$).

Specifically, a prefix of a path is a subsequence of this path, and has the same start node (i.e., the first node in a path) as the path. Based on the above mathematical definition of path prefix, and our `Path` data type defined in section 5.1.3, we first define a `append` function that concatenates two paths by appending one path to the beginning of the other path, and then implement the prefix of a path based on the `append` function. The following presents the implementation of `append` and `pathPrefix`.

```
append : (p1 : Path s v g) ->
         (p2 : Path v w g) ->
         Path s w g

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 type of `append` function specifies that, given a path `p1` from node `s` to `v` in `g`, and a path `p2` from node `v` to `w` in `g`, the result of appending `p1` to the head of `p2` is a path from node `s` to `w` in `g`. Notice that the ending node `v` in `p1` is exactly the starting node of `p2`, and the resulting path of appending `p1` to `p2` (i.e., return value of `append p1 p2`) starts from the same node as `p1`, and ends at the same node as `p2`. Then according to our definition of prefix of a path above, the first input path `p1` is actually a prefix of the return value of `append p1 p2`.

The `pathPrefix` function is a predicate stating that the first input path `pprefix` is a prefix of the second input path `p`. `(v ** P)` is the syntax for dependent pairs, which states that the second element `P` in the pair is dependent on the value of the first element `v`. Dependent pairs are used to represent existential quantification in Idris. For instance, the dependent pair `(n : Nat ** Vect n Nat)` states the existence of a natural number `n`, such that `n` is the length of the `Vect` included as the second element of the pair. In the definition of `pathPrefix`, as `pprefix` is the prefix of `p`, then there only exists one path (with type `Path w v g`) such that the result of appending `pprefix` to this path is `p`. This is specified by the dependent pair `(ppost : Path w v g ** append pprefix ppost = p)` in our definition, which quantified a specific path `ppost` with type `Path w v g` such that the result of `append pprefix ppost` is `p`, and hence the path `pprefix` is a prefix of `p`.

Given the definition of `pathPrefix` above and definition of shortest path in section 5.1.3, the implementation of Lemma 4.1 is provided below.

```
shorter_trans : {g : Graph gsize weight ops} ->
                (p1 : Path s w g) ->
                (p2 : Path s w g) ->
                (p3 : Path w v g) ->
                (p : dgte ops (length p1) (length p2) = False) ->
                dgte ops (length $ append p1 p3)
                         (length $ append p2 p3) = False


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)
l1_prefixSP spath (post ** appendRefl) lp_pre {ops} {sp_pre}
  with (dgte ops (length lp_pre) (length sp_pre)) proof lpsp
    | True = Refl
    | False = absurd $ contradict (spath (append lp_pre post))
                                  (rewrite (sym appendRefl) in
                                    (shorter_trans
                                      lp_pre sp_pre post (sym lpsp)))
```

The type of the `l1_prefixSP` states that, given an input graph `g`, nodes `s, v, w`, a path `sp` from `s` to `v` in `g` (as specifies by the type `Path s v g`), the prefix of `sp` from `s` to `w` (named as `sp_pre`), if `sp` is a shortest path from `s` to `v`, as specifies by `shortestPath g sp`, then the prefix `sp_pre` of `sp` is also a shortest path from `s` to `w` in `g`.

The definition of `shortestPath` allows us to bring into scope a variable `lp_pre` with type `Path s w g` that quantifies over any path from `s` to `w` in `g`. We approach the proof of `l1_prefixSP` by matching on the value of (dgte ops (length lp_pre) (length sp_pre)), which compares the length of lp_pre against the length of the prefix sp_pre of sp.

When (dgte ops (length lp_pre) (length sp_pre)) is matched to True, this indicates that the length of any path from s to w is longer than or equal to the length of sp_pre, then sp_pre is a shortest path from s to w based on the definition of shortest path.

When (dgte ops (length lp_pre) (length sp_pre)) is matched to False, this indicates that the length of lp_pre is smaller than the length of sp_pre, i.e., length(lp_pre) < length(sp_pre). Since sp_pre, lp_pre are both paths from s in w in g, and appending sp_pre to ppost gives us the path sp from s to v (sp_pre is the prefix of sp), then we can construct another path p' : Path s v g from s to v by appending lp_pre to ppost, whose length is smaller than that of sp as we conclude length(lp_pre) < length(sp_pre) before. As indicated by the type of shorter_trans provided above, (shorter_trans lp_pre sp_pre post (sym lpsp)) is a proof that shows, since we know length(lp_pre) < length(sp_pre), then the length of the path obtained by appending lp_pre to ppost (the length p'), is smaller than the length of the path obtained by appending sp_pre to ppost (the length of p). This contradicts with the statement that p is a shortest path from s to v in g(specified by shortestPath sp p). Hence with prove by contradiction we can show that the case when (dgte ops (length lp_pre) (length sp_pre)) is matched to False is impossible, i.e., the length of sp_pre is smaller than or equal to the length of any other s to w path in g, and that sp_pre is a shortest path from s to v in g. Proof of l1_prefixSP is completed.

The structure of the implementation of Lemma 4.2 and Lemma 4.5 is as follows: we first define functions that specifies the Column properties stated by each lemma, and then implement a function that proves the preservation of these properties. This structure provides a more clear and straightforward type signatures for our functions in the verification program by separating the types that specifies Column properties from the types of the proofs.

5.3.1.2 Lemma 2 - l2_existPath

In our mathematical proof of Dijkstra's correctness, Lemma 4.2 states that given an input graph g, for all nodes v in g, if $dist_{n+1}[v] \neq \infty$, then $dist_{n+1}[v]$ is the length of some s-to-v path in g. As mentioned at the beginning of this section, in our verification program, we state Lemma 4.2 as a Column property and prove these properties preserve after calling runHelper. The function neDInfPath provided below specifies the Column property stated by Lemma 4.2.

```
neDInfPath : {g : Graph gsize weight ops} ->
             (cl : Column len g src) ->
             Type
neDInfPath cl {g} {src} {ops} {gsize}
  = (v : Node gsize) ->
    (ne : dgte ops (nodeDistN v cl) DInf = False) ->
    (psv : Path src v g ** dEq ops (nodeDistN v cl) (length psv) =
True)
```

Given cl with type Column len g src, the function neDInfPath specifies that for any node v in the graph, if the distance value of v stored in cl is smaller than infinity, then it is the length of some path from src to v in g. nodeDistN is a function that indexes the distance value for a specific node in a Column, and in the definition of neDInfPath, nodeDistN v cl gets the distance value of v stored in cl, and the dependent pair (psv : Path src v g ** dEq ops (nodeDistN v cl) (length psv) = True) specifies the existence of a path psv from src to v in g, such that the distance value of v stored in cl is the length of psv. We then define the type of the function l2_existPath that states the preservation of the neDInfPath property.

```
l2_existPath : {g : Graph gsize weight ops} ->
               (cl : Column (S len) g src) ->
               (l2_ih : neDInfPath cl) ->
               neDInfPath (runHelper cl)
```

l2_existPath states that given cl with type Column (S len) g src, if neDInfPath holds for cl (specified by l2_ih), then it also holds for the column generated by (runHelper cl). Notice that the input cl of l2_existPath contains a non-empty vector of unexplored nodes, which is restricted by the runHelper function. Similar to the previous proof on l1_prefixSP, we can bring the node v and statement ne : dgte ops (nodeDistN v cl) DInf = False in neDInfPath into scope. The proof of l2_existPath is approached by matching on the distance value of v stored in the Column generated by runHelper cl. If the distance value of v in runHelper cl is the same as that in cl, then the proof is given by l2_ih. Otherwise we check whether v is equal to getMin cl (the unexplored node with minimum distance value chosed by the algorithm for exploring its neighbors, mentioned in Section 5.2.2), and prove both cases by applying l2_ih on (getMin cl). In the case when v is not equal to getMin cl, the proof is still incomplete due to an unclear type error introduced by calling the inNodeset function, as discussed under Section 6.

5.3.1.3 Lemma 3 - l3_preserveDelta

In verifying Dijkstra's correctness, it is important to show that forall nodes v in the input graph, once the distance value calculated for v is equal to the minimum distance value from the source node to v, then the distance value of v does not change through the execution of the algorithm. This is proved by Lemma 4.3 in the mathematical proof of Dijkstra's algorithm, and implemented by the function l3_preserveDelta below (the proof of l3_preserveDelta is provided in later paragraphs).

```
l3_preserveDelta : {g : Graph gsize weight ops} ->
                   (cl : Column (S len) g src) ->
                   (l2_ih : neDInfPath cl) ->
                   (v : Node gsize) ->
                   (psv : Path src v g) ->
                   (psv_sv : shortestPath g psv) ->
                   (eq : dEq ops (nodeDistN v cl)
                           (length psv) = True) ->
                   dEq ops (nodeDistN v (runHelper cl)) (length psv) =
  True
```

l3_preserveDelta states that given a Column named cl, for any node v in graph, given a shortest path named psv from src to v (specified by psv_sv), if the distance value of v stored in cl is equal to the length of psv(stated by the input eq : dEq ops (nodeDistN v cl) (length psv) = True), then the distance value of v stored in runHelper cl is also equal to the length of psv. Since the proof of Lemma 4.3 is based on Lemma 4.2 as we mentioned at the beginning of Section 5.3.1, the proof of property neDInPath on cl is provided by the input l2_ih : neDInfPath cl.

We prove `l3_preserveDelta` by showing that the distance of a node `v` stored in a `Column` is non-increasing (either decrease or remain the same) each time after calling `runHelper`. If the current distance stored in `Column` for `v` is equal to the minimum distance value, than there cannot exist a smaller distance value for `v`, hence the distance stored for `v` remains unchanged each time after calling `runHelper`.

To implement `l3_preserveDelta`, we first need to show that the distance value stored for all node is non-increasing after each call of `runHelper`. The function `runDecre` provided below states this property. We provide a detailed discussion on the implementation of `runDecre` as it presents how we approach the proofs of some key lemmas in our verification. Specifically, we break the statement that we want to prove into smaller ones by destructing the data structures involved in the original statement. In other words, the implementation of functions that involve more complex data types can be built on functions that deal with simpler data types, which are easier to approach. The following explanation on the implementation of `runDecre` illustrates this technique.

```
runDecre : {g : Graph gsize weight ops} ->
           (cl : Column (S len) g src) ->
           (v : Node gsize) ->
           dgte ops (nodeDistN v cl)
                (nodeDistN v (runHelper cl)) = True
runDecre (MKColumn g src (S len) unexp dist) (MKNode nv) {gsize} {ops}
  {weight}
  = distDecre g min_node min_dist (mkNodes gsize) dist
          (finToNat nv) {p=nvLTE nv}
  where
   ...
```

The return type of the `runDecre` function specifies that for all node `v`, the distance value stored for `v` in `cl` is either decreasing, or maintains the same after each call of `runHelper` on `cl`. Since `runDecre` involves the `Column` data type, and the main field in `Column` that concerns us here is the `Vect` of distance values, the implementation of `runDecre` is built on a function `distDecre`, which states the same non-increasing property of distance values calculated but only involves the `Vect` of distance values instead. As the function `runHelper` calls `updateDist` for updating the distance value `Vect` (mentioned in Section 5.2.2), the type of `distDecre` also involves the `updateDist` function. Since we use the `Fin` type value carried by node `v` to index its distance value stored in `cl` (Section 5.1.4), and recursing on a `Fin` type value with base case `FZ` introduces more complications, we define a `finToNat` function that convert a `Fin` into its corresponding `Nat` value. As in the definition of `runDecre`, the `distDecre` function is called on the corresponding `Nat` value of `nv` calculated by the function `finToNat`. The implementation of `distDecre` is presented below.

```
distDecre : (g : Graph gsize weight ops) ->
            (mn : Node gsize) ->
            (min_dist : Distance weight) ->
            (nodes : Vect m (Node gsize)) ->
            (dist : Vect m (Distance weight)) ->
            (nv : Nat) ->
            {auto p : LT nv m} ->
            dgte ops (indexN nv dist)
                 (indexN nv
                  (updateDist g mn min_dist nodes dist)) = True
distDecre g mn _ Nil Nil nv {p} = absurd $ succNotLTEzero p
```

```
distDecre g mn min_dist (n :: ns) (d :: ds) Z
  = calcDistEq g mn n min_dist d
distDecre g mn min_dist (n :: ns) (d :: ds) (S nv) {p=LTESucc p'}
  = distDecre g mn min_dist ns ds nv {p= p'}
```

Inputs of distDecre include the current node with minimum distance, named as mn, and its distance value min_dist; nodes and dist denotes a Vect of nodes in the graph and a Vect of the corresponding distance values for these nodes respectively. As distDecre recurs on both nodes and dist, both Vects have the same length m but the value of m changes during each recursive step.

The nv argument is the index of the node v(from the runDecre function) in nodes and its corresponding distance value in dist, with respect to the current recursive step. Specifically, as we define the distDecre function to recur on nodes, dist, and nv simultaneously, the length of nodes and dist, and the value of nv decreases by one during each recursive step until one of them reaches Z, which is the base case. For instance if the the value of nv is 'S Z' during the current recursion, then in the next recursion the value passed in for nv should be Z, however as distDecre recurs on the tails of nodes and dist, nv denotes the locations of the same node in nodes and its corresponding distance value in dist during every recursive step. Lastly, the implicit parameter p is a proof stating that the current index nv is smaller than the length of Vects m, which ensures safe indexing. The return type of distDecre specifies that the distance of node indexed by nv stored in dist is non-decreasing after calling updateDist on dist.

Similar to the structure of runDecre, the distDecre function is again built on a calcDistEq function, which concerns the distance value for one specific node rather than a Vect of distance values. The definition of calcDistEq is provided below.

```
calcDistEq : (g : Graph gsize weight ops) ->
             (min_node : Node gsize) ->
             (cur : Node gsize) ->
             (min_dist : Distance weight) ->
             (cur_dist : Distance weight) ->
             dgte ops cur_dist
                  (calcDist g min_node cur min_dist cur_dist) = True
calcDistEq g min_node cur min_dist cur_dist {ops}
  with (dgte ops cur_dist
           (dplus ops (edgeW g min_node cur) min_dist)) proof sdist
    | True = sym sdist
    | False = dgteRefl
```

The return type of calcDistEq states that the distance value for the node cur (named as cur_dist) is smaller or equal to that after running calcDist on cur. Based on the implementation of calcDist in Section 5.2.3, the proof of calcDistEq is directly given by matching on the result of checking whether cur_dist is greater than or equal to the sum of min_dist and edge weight between min_node and cur using the with rule. When cur_dist is greater(or equals), then the proof sdist generated by the match states exactly what we want to prove. When cur_dist is smaller than the sum, then the distance value of cur calculated by calcDist should be cur_dist as calcDist chooses the minimum distance value between the two, hence we simply call the dgteRefl function, which states that a distance value is greater than or equal to itself.

Based on the runDecre function defined above, we offer the following implementation for l3_preserveDelta.

```
l3_preserveDelta : {g : Graph gsize weight ops} ->
                   (cl : Column (S len) g src) ->
                   (l2_ih : neDInfPath cl) ->
                   (v : Node gsize) ->
                   (psv : Path src v g) ->
                   (psv_sp : shortestPath g psv) ->
                   (eq : dEq ops (nodeDistN v cl)
                            (length psv) = True) ->
                   dEq ops (nodeDistN v (runHelper cl)) (length psv) =
  True
l3_preserveDelta cl l2_ih v psv psv_sp eq {g} {ops} {src}
  with (l2_existPath cl l2_ih v
        (dgteDInfTrans {ops=ops}
          (nodeDistN v cl)
          (nodeDistN v (runHelper cl))
          (pathlenNotDInf (nodeDistN v cl) psv eq)
          (runDecre cl v)))
    | (lpath ** runclv_lp)
      = dgteEq (dgteEqTrans runclv_lp True (psv_sp lpath))
               (dgteEqTrans (dEqComm eq) True (runDecre cl v))
```

Given that `neDInfPath` holds for `cl` (specified by input argument `l2_ih`), we first apply Lemma 4.2 `l2_existPath` to show that the distance value of `v` after running `runHelper`, i.e., '`nodeDistN v (runHelper cl)`' is the length of some `src-to-v` path. In order to apply `l2_existPath`, we need to show that the value of '`nodeDistN v (runHelper cl)`' is smaller than `DInf` (infinity distance value). `dgteDInfTrans` is a proof stating that, given two distance values `d1, d2`, if `d1` is smaller than `DInf`, and `d2` is smaller than `d1`, then `d2` is also smaller than `DInf`. When applying `dgteDInfTrans` in the implementation of `l3_preserveDelta` above, '`nodeDistN v cl`' corresponds to `d1` and '`nodeDistN v (runHelper cl)`' corresponds to `d2`. The `pathlenNotDInf` function states that the length of any path is smaller than `DInf`. Since we know '`nodeDistN v cl`' is the length of a path `psv` (specified by input argument `eq`), then '`nodeDistN v cl`' is smaller than `DInf`. '`runDecre cl v`' states that the value of '`nodeDistN v (runHelper cl)`' is smaller than or equal to '`nodeDistN v cl`'. By applying `dgteDInfTrans` to all of these information above, we obtained the dependent pair `(lpath ** runclv_lp)`, where `lpath` is a `src-to-v` path, and `runclv_lp` is a proof that the value of '`nodeDistN v (runHelper cl)`' is the length of `lpath`.

The `dgteEq` function states that for two distance values `d1, d2`, if `d1` is greater than or equal to `d2` and `d2` is greater than or equal to `d1`, then `d1` must equal to `d2`. The `dgteEqTrans` function states that, given three distance values `d1, d2, d3`, if `d1` equals to `d2`, and `d2` is smaller(or greater) than `d3`, then `d1` is smaller(or greater) than `d3`. By applying `dgteEqTrans` to `runclv_lp` and '`psv_sp lpath`' (which states that the length of `lpath` is smaller than the length of `psv`), we know that the value of '`nodeDistN v (runHelper cl)`' is greater than or equal to '`length psv`' (mark as [S1]). However applying `dgteEqTrans` to `eq` and 'runDecre cl v' we obtained that the '`nodeDistN v (runHelper cl)`' should be smaller than the length of `psv`(mark as [S2]). Hence by applying `dgteEq` to [S1] and [S2], we know that '`nodeDistN v (runHelper cl)`' is equal to the length of `psv`. The implementation of `l3_preserveDelta` proof is complete.

5.3.1.4 Lemma 5 - `l5_spath`

The structure of the implementation of l5_spath for proving Lemma 4.5 is similar to l2_existPath in Section 5.3.1.2. In our verification program, we first define functions that specify the Column properties stated by Lemma 4.5 (properties are included in Section 4.5), and prove that if all of the properties hold for a Column cl, then the properties preserve after calling runHelper on cl. Similar to the structure of lemmas, the properties are also defined in order, that the preservation proofs of properties defined later depend on properties defined earlier. As the preservation proofs for all properties are quite complicated and involve large amount of details, in the following we only provide the types of the function that implements the proof, and summaries how we approach the proof. Explanation on the definition and preservation proof for each property are provided in the order that they are defined, hence the preservation proof in the later sections are depends on the proofs introduced earlier.

The proof for the base case of statement 2 to statement 4 require extra information to be provided, for instance the for the Vect of distance values in the first Column initialized by the dijkstra function, the only node whose distance value is not DInf is the source node. These information are given by the initialization of Dijkstra's algorithm, however as the current definitions of functions(mkdists and mknodes) that calculate the initial values are difficult to work with, the proof on the base cases are still incomplete. This issue does not largely affect the validity of our implementation of Lemma 4.5 below, as it can be resolved by providing better definitions for mkdists and mknodes.

As mentioned in Section 5.3.1, in the verification program we merge Lemma 4.4 into Lemma 4.5 as one of its statement, then the implementation of Lemma 4.5 involves four statements (i.e. Column properties). To provide a more clear, structured type for the preservation proof for Lemma 4.5, we define the l5_stms function below that collects all statements together with a tuple.

```
l5_stms : {g : Graph gsize weight ops} ->
          (cl : Column len g src) ->
          Type
l5_stms cl = (lessDInf cl,
              distv_min cl,
              unexpDelta cl,
              expDistIsDelta cl)
```

Given a Column cl, the function l5_stms is a predicate stating that collects four function, lessDInf, distv_min, unexpDelta, expDistIsDelta, in the form of a tuple, where each of them state one Column property. The elements in the tuple are listed in the same order as they are defined, hence the preservation proof for expDistIsDelta is built on the proof for the previous three properties, and the proof that shows the preservation of unexpDelta is again built on the preservation proof on the previous two ... etc. The following provides more detail on the definition and the corresponding preservation proof for each function mentioned above.

The function lessDInf is defined for the first property, which states that the distance value calculated for any explored node is less than DInf(infinity distance value).

```
lessDInf : {g : Graph gsize weight ops} ->
           (cl : Column len g src) ->
           Type
lessDInf cl {gsize} {ops}
  = (v : Node gsize) ->
    (expV : explored v cl) ->
```

```
        dgte ops (nodeDistN v cl) DInf = False
```

Given a Column cl corresponds to the graph g, lessDInf states that for any node v in g, if v is explored (stated by expV), then the distance value for v stored in cl(indexed by the nodeDistN function) is smaller than DInf. The l5_stm1 function below provides the preservation proof for the lessDInf property.

```
 {-  the implementation of l5_stm1 is not included here
     and a summary of our proof is included in the curly braces
instead
 -}
 l5_stm1 : {g : Graph gsize weight ops} ->
           (cl : Column (S len) g src) ->
           (l5_ih : l5_stms cl) ->
           lessDInf (runHelper cl)
 l5_stm1 cl (ih1, ih2, ih3, ih4) v expVR {ops} {src}
   with (getMin cl == v) proof min_is_v
     | True = ?l51t_hole
     | False = {- apply l5_ih assumption -}
```

As mentioned at the beginning of this section, the preservation proof for each property assumes that all properties specified by l5_stms hold for the current Column, and shows how each property preserves after calling runHelper. Given an input Column cl, l5_ih states that all properties in l5_stms, including lessDInf, hold for cl, and based on this assumption, the function l5_stm1 states that the lessDInf property preserves after calling runHelper on cl.

Similar to the proof of Lemma 4.2 in Section 5.3.1.2, the definition of lessDInf allows us to bring the node v and expV : explored v (runHelper cl) into scope. Notice that expV states that v is an explored node with respect to runHelper cl rather than cl. The implementation of l5_stm1 is approached by checking whether v equal to 'getMin cl', i.e., the current node being explored (mentioned in Section 5.2.2). If v is not 'getMin cl', then v must also be an explored node in cl, then the proof can be completed by applying the assumption l5_ih. In the case when v is equal to 'getMin cl', the proof requires an assumption that the graph is a connected graph, such that there must have a src-to-v path in the graph. The proof under this case is still incomplete as the assumption of connected graph is not yet passed in to the l5_stm1 as an input parameter(although it is assumed through our verification process), however with more time granted, this proof can be completed by adding a parameter that states the connected graph assumption.

The second statement is specified by the distv_min function presented below.

```
  distv_min : {g : Graph gsize weight ops} ->
              (cl : Column len g src) ->
              Type
  distv_min cl {g} {gsize} {src}
    = (v, w : Node gsize) ->
      (exp_w : explored w cl) ->
      (psw : Path src w g) ->
      (spsw : shortestPath g psw) ->
      dgte ops (dplus ops (edgeW g w v) (nodeDistN w cl))
               (nodeDistN v cl) = True
```

Given a Column cl, distv_min states that for an explored node w (specified by exp_w), for any node v in g, the sum of the distance of w stored in cl and weight(w, v) (notation introduced in Section 4.3) is greater than or equal to the distance value of v stored in cl. We also introduce a shortest `src-to-w` path `psw` (`spsw` states that `psw` is a shortest path) as it is required for the `l5_stm2` function below that implements the preservation proof for `distv_min`.

```
{-  the implementation of l5_stm2 is not included here
    and a summary of our proof is provided in the curly braces
instead
-}
l5_stm2 : {g : Graph gsize weight ops} ->
          (cl : Column (S len) g src) ->
          (l2_ih : neDInfPath cl) ->
          (l5_ih : l5_stms cl) ->
          distv_min (runHelper cl)
l5_stm2 cl l2_ih (ih1, ih2, ih3, ih4) v w expW psw spsw {ops} {g}
  with (getMin cl == w) proof min_is_w
    | True = {- apply 'runDgteMin' and 'minDist_preserve' -}
    | False = {- apply l5_ih -}
```

Given a `Column cl`, the input `l2_ih` provides the assumption that property `neDInfPath` holds for `cl`, which is required by the implementation of our proof. Passing in assumption of properties stated by Lemma 4.2 is valid as in our verification program, the proof of lemmas defined later depend on the previous lemmas defined. Input `l5_ih` provides the assumption that `l5_stms` holds for `cl`. The return type of `l5_stm2` states that the sum of the distance of `w` stored in '`runHelper cl`' and `weight(w, v)` is greater than or equal to the distance value of `v` stored in '`runHelper cl`'.

The implementation of `l5_stm2` is again approached by checking whether the explored node `w` is equal to '`getMin cl`'. When `w` is not equal to '`getMin cl`', the proof can be completed by applying the assumption `l5_ih` and Lemma 4.3 `l3_preserveDelta` (Section 5.3.1.3 (which requires the input `l2_ih`).

When `w` is equal to '`getMin cl`', since our Dijkstra's implementation chooses the minimum value between the sum of '`getMin cl`' and `weight(getMin cl, v)` and the distance value of `v` stored in `cl` (specifically, by calling the `calcDist` function mentioned in Section 5.2.3), we implement a helper proof named `runDgteMin` that states this property of our implementation. The type of `runDgteMin` is provided below.

```
runDgteMin : {g : Graph gsize weight ops} ->
             (cl : Column (S len) g src) ->
             (v : Node gsize) ->
             dgte ops (dplus ops (edgeW g (getMin cl) v)
                         (nodeDistN (getMin cl) cl))
                      (nodeDistN v (runHelper cl)) = True
```

Specifically, `runDgteMin` specifies that for all node `v` in the graph, the sum of `weight(getMin cl, v)` and the distance value of '`getMin cl`' stored in `cl`(specified by '`nodeDistN (getMin cl) cl`'), is larger than or equal to the distance value of `v` stored in `runHelper`. We eliminate the details on the implementation of `runDgteMin` as it is highly similar to that of `runDecre` in Section 5.3.1.3.

Based on `runDgteMin` we know that sum of '`nodeDistN (getMin cl) cl`' and `weight(getMin cl, v)` is larger than or equal to '`nodeDistN v (runHelper cl)`'. Recall what we want to prove is that the property `distv_min` holds for the `Column` calculated by '`runHelper cl`', hence under this case when `w` is equal to '`getMin cl`', what we want to show is that the sum of '`nodeDistN (getMin cl) (runHelper cl)`'(instead of `cl`) and `weight(getMin cl, v)` is larger than or equal to '`nodeDistN v (runHelper cl)`'. This can be resolved with a proof (named `minDist_preserve`) that shows the distance value of 'getMin cl' stored in cl and 'runHelper cl' is the same. The type of minDist_preserve is presented below.

```
minDist_preserve : {g : Graph gsize weight ops} ->
                   (cl : Column (S len) g src) ->
                   dEq ops (nodeDistN (getMin cl) cl)
                      (nodeDistN (getMin cl) (runHelper cl)) = True
```

The function dEq is an operator that checks whether two distance values are equal. The return type of minDist_preserve specifies that the distance value of 'getMin cl' remains unchanged after calling runHelper on cl. Although this proof is not yet complete due to the unclear type error in applying the inNodeset function (explained in Section 6), from a theoretical prospective this is a valid statement, as when updating the distance value for 'getMin cl', since a node cannot be in the nodeset of itself, hence calcDist compares the value of 'nodeDistN (getMin cl) cl' with the sum of weight(getMin cl, getMin cl) and 'nodeDistN (getMin cl) cl', the value of weight(getMin cl, getMin cl) would be DInf, and the updated distance for 'getMin cl' is still 'nodeDistN (getMin cl) cl', which is exactly what stated by the minDist_preserve function above. The proof on l5_stm2 under the case when w is 'getMin cl' can be implemented by combining the functions runDgteMin and minDist_preserve.

The third statement is defined by the unexpDelta function below.

```
unexpDelta : {g : Graph gsize weight ops} ->
             (cl : Column len g src) ->
             Type
unexpDelta cl {g} {gsize} {ops} {src}
  = (v, w : Node gsize) ->
    (expV : explored v cl) ->
    (unexpW : unexplored w cl) ->
    (psw : Path src w g) ->
    (sp : shortestPath g psw) ->
    dgte ops (length psw) (nodeDistN v cl) = True
```

Given a Column cl corresponds to the graph g, for two nodes v, w in the graph g, where v is explored and w is unexplored (stated by expV and unexpW respectively), unexpDelta states that the length of a shortest path from src to w in g(psw denotes the shortest path) is greater than or equal to the distance value of v stored in cl. The function l5_stm3 functions proves the preservation of the unexpDelta property.

```
{-  the implementation of l5_stm3 is not included here
    and a summary of our proof is provided in the  curly braces instead
-}
l5_stm3 : {g : Graph gsize weight ops} ->
          (cl : Column (S len) g src) ->
          (nadj : ((n : Node gsize) ->
                inNodeset n (getNeighbors g n) = False)) ->
          (l2_ih : neDInfPath cl) ->
          (l5_ih : l5_stms cl) ->
          (st1 : lessDInf (runHelper cl)) ->
          (st2 : distv_min (runHelper cl)) ->
          unexpDelta (runHelper cl)
-- case1 : w = src, and w is unexplored, then no nodes are explored
l5_stm3 cl nadj l2_ih l5_ih st1 st2 v w expVR unexpWR
  {src=w} (Unit g w) spsw = ?l53Impossible
-- case2 : shortest src-to-w edge is a path with one edge
```

```
l5_stm3 cl nadj l2_ih l5_ih st1 st2 v w expVR unexpWR {src}
  (Cons (Unit g src) w adj_src_w) spsw = ?l53Base
-- case3 : shortest src-to-w edge is a path with more than one edge
l5_stm3 cl nadj l2_ih (ih1, ih2, ih3, ih4) st1 st2 v w expVR unexpWR
  (Cons (Cons psx u adj_x_u) w adj_uw) spsw {g} {ops}
    -- check if v is equal to (getMin cl)
    with (getMin cl == v) proof min_is_v
      -- case[a]: if true, check if u is explored
      | True with (checkUnexplored u (runHelper cl)) proof u_exp
        -- case[b]
          | Yes unexpU = {- recursively apply l5_stm3 on v and u -}
          -- check if v is equal to u
          | No expU with (v == u) proof v_is_u
            -- case[c]
            | True with (adj_getPrev adj_x_u)
              | x with (checkUnexplored x (runHelper cl)) proof x_exp
                | Yes unexpX
                    = {- recursively apply l5_stm3 to v and x -}
                | No expX with (u == x) proof u_is_x
                  | True = {- apply nadj -}
                  | False with (l2_existPath cl l2_ih v (st1 v expVR))
                    | (lpsv ** rclvEq)
                        = {- apply st2 and l3_preserveDelta -}
            -- case[d]
            | False = {- apply ih2, ih5, minCl, and runDecre -}
      | False = {- apply l5_ih -}
```

Notice that the previous two statements, `st1` and `st2`, are passed in as parameters for the implementation of `l5_stm3`. This is valid as we mentioned at the beginning of this section, the proof of statements defined later is built on the statements defined earlier. The proof for the first two cases are incomplete due to time limit and the issue concerning functions `mkdists` and `mknodes` mentioned above.

In the third case, the shortest path from `src` to `w` contains more than one edges. The proof for this case is highly simiarly to the mathmatical proof of Lemma 4.5. Specifically, the shortest `src-to-w` path (denote as `psw`) is constructed from a `src-to-u` path (denote as `psu`), which is again constructed from a `src-to-x` path named `psx`. In other words, `x`, `u` denotes the two nodes right before `w` in the path `psw`. Notice that `psw` denotes an arbitrary shortest `src-to-w` path, hence `x`, `u` are arbitrary nodes that denote the two nodes right before `w` in any shortest `src-to-w` path rather than one specific shortest path. As the proof for this case is very complicated and involves lots of details, we only provide the skeleton of our proof by listing all the intermediate values that we have matched on in the implementation of this proof. The summary of the proof under each cases is specified in curly braces. We first bring all the premises of `unexpDelta` into scope. Similar to the proof of previous statement, we check whether `v` is equal to '`getMin cl`'. When `v` is not equal to '`getMin cl`', and since `v` is already explored, then `v` must also be an explored node in `cl`, hence we can apply the `l5_ih` assumption to complete the proof.

When `v` is equal to '`getMin cl`'(case [a]), we check whether the node `u` right before `w` is explored or not (`x` is the node right before `u`). When `u` is unexplored(case[b]), then we can recursively apply `l5_stm3` on `v` and `u` to complete the proof. When `u` is explored, we check if `v` is equal to `u`, i.e., checking whether the node right before `w` in the path `psw` is equal to `v`.

If `u` is not equal to `v`(case[d]), then we apply `ih2` and `ih4`(from `l5_ih`) on the node `u` to show that the sum of `weight(u, w)` and the length of `psu` (shortest `src-to-u` path) is greater than or equal to the distance value of `w` in `cl`. Since `v` was the unexplored node with minimum distance value, than the distance value of `v` in `cl` is smaller than the distance value of `w` in `cl`(specified by a `minCl` function implemented), hence the distance of `v` in `cl` is smaller than or equal to the sum of `weight(u, w)` and the length of `psu`. As `u` is the node right before `w` in `psw`, then the distance of `v` in `cl` is smaller than or equal to the the length of `psw`. Combine this with the function `runDecre` that states `nodeDistN v cl` is greater than or equal to `nodeDistN v (runHelper cl)`, then we complete the proof that the length of `psw` is greater than or equal to `nodeDistN v (runHelper cl)`.

If `u` is equal to `v`(case[c]), then we follow the same pattern and check whether the node `x` right before `u` is explored or not. When `x` is unexplored, we recursive apply `l5_stm3` on `x` and `v`. Otherwise we check if `u` is equal to `x`. The case when `u` is equal to `x` is restricted by `nadj` as `x` cannot be adjacent to `u`.

When `u` is not equal to `x`, we apply `st2` and `l3_preserveDelta` (Section 5.3.1.3) to show that the sum of `weight(x, u)` and length of path `psx` is greater than or equal to the distance of `u` in '`runHelper cl`', hence the result of adding `weight(u, w)` to the sum of `weight(x, u)` and length of path `psx`, which is exactly the length `psw`, is greater than or equal to the distance value of `u` stored in '`runHelper cl`'. As under this case `u` is equal to `v`, the proof of `l5_stm3` is complete.

The function `expDistIsDelta` below specifies the last statement in Lemma 4.5.

```
expDistIsDelta : {g : Graph gsize weight ops} ->
                 (cl : Column len g src) ->
                 Type
expDistIsDelta cl {g} {gsize} {ops} {src}
  = (v : Node gsize) ->
    (exp : explored v cl) ->
    (psv : Path src v g) ->
    (sp : shortestPath g psv) ->
    dEq ops (nodeDistN v cl) (length psv) = True
```

For all node `v` that is explored in the `Column cl` (specified by `exp`), `expDistIsDelta` states that the distance value of `v` stored in `cl` is equal to the length of a shortest `src-to-v` path(named as `psv`). In other words, `expDistIsDelta` states that the distance value of an explored node `v` stored in `cl` is indeed the minimum distance value from `src` to `v`. The function `l5_stm4` below implements the preservation proof for `expDistIsDelta`.

```
{-  the implementation of l5_stm4 is not included here
    and a summary of our proof is included in the curly braces instead
-}
l5_stm4 : {g : Graph gsize weight ops} ->
          (cl : Column (S len) g src) ->
          (nadj : ((n : Node gsize) ->
                  inNodeset n (getNeighbors g n) = False)) ->
          (l2_ih : neDInfPath cl) ->
          (l5_ih : l5_stms cl) ->
          (st1 : lessDInf (runHelper cl)) ->
          (st2 : distv_min (runHelper cl)) ->
          (st3 : unexpDelta (runHelper cl)) ->
          expDistIsDelta (runHelper cl)
-- case1 : when v is src
```

```
l5_stm4 cl nadj l2_ih l5_ih st1 st2 st3 v expVR {src=v} (Unit g v) spsv
   = ?l5_unit
-- case1 : when v is not src
l5_stm4 cl nadj l2_ih (ih1, ih2, ih3, ih4) st1 st2 st3 v expVR (Cons psw
    v adj_wv) spsv {g} {ops} {src}
  with (getMin cl == v) proof min_is_v
    -- case[a]
    | True with (l2_existPath cl l2_ih v (st1 v expVR))
      | (lpsv ** rclvEq) with (adj_getPrev adj_wv)
        | w with (checkUnexplored w (runHelper cl)) proof w_exp
          -- case[c]
          | Yes unexpW = {- combine st3 and the rclvEq proof -}
          -- case[d]
          | No expW with (v == w) proof v_is_w
            | True  = {- apply nadj -}
            -- case[e]
            | False = {- apply dgteEq on rclvEq and the proof obtained
   by applying l3_preserveDelta and ih4 to u -}
    -- case[b]
    | False = {- apply ih4 and l3_preserveDelta -}
```

The functions `st1`, `st2`, `st3` specify that the previous three statements hold for '`runHelper cl`' respectively, and are passed in as input parameters as they are required for the implementation of `l5_stm4`.

In the base case when `v` is the source node `src`, the proof is not yet complete, again due to the issue concerning `mkdists` and `mknodes`, and can be resolved by providing a better definitions for both functions. In the case when `v` is not the source node, we check whether `v` is equal to '`getMin cl`'. If `v` is not equal to '`getMin cl`'(case[b]), then `v` must be an explored node in `cl`, hence we can combine the assumption `ih4` and Lemma 4.3 `l3_preserveDelta` to show that '`nodeDist v (runHelper cl)`' is also equal to '`nodeDist v cl`', which is equal to the length of shortest `src-to-v` path.

In case[a] when `v` is equal to '`getMin cl`', we first apply Lemma 4.2 `l2_existPath` and `st1` (which states that the `distv_min` property holds for '`runHelper cl`') to show that the distance of `v` stored in '`runHelper cl`' is the length of some `src-to-v` path named `lpsv` (`rclvEq` specifies this equality). We then bring the node `w` right before `v` in the shortest `src-to-v` path (denote as `psv`, which equals to (`Cons psw v adj_wv`) in the implementation above) into scope, hence the length of `psv` is equal to the sum of `weight(w, v)` and length of `psw`.

We then check whether `w` is explored or not. When `w` is unexplored(case[c]), then based on `st3` we know that the length of `psw` is greater than or equal to the distance of `v` in '`runHelper cl`', hence the length of `psv` must be greater than or equal to the distance of `v` in '`runHelper cl`' (marked [S2]). However the statement of shortest path `spsv` states that the length of `psv` is smaller than or equal to the length of `lpsv`, which is equal to the distance of `v` in '`runHelper cl`'([S4]). By applying the function `dgteEq`(which states that for two distance values d1, d2, if d1 >= d2 and d2 >= d1, then d1 = d2) on [S3] and [S4], we prove that the distance of `v` in '`runHelper cl`' is equal to the length of `psv`.

When `w` is explored (case[d], we check whether `v` is equal to `w`, i.e., check whether the node right before `v` in the shortest path `psv` is `v`. As we restrict a node from being a neighbor of itself, then the case when v is equal to w is marked as absurd by applying the nadj function.

In case[e] v is not w, as w is explored, we apply st2 on node v and w (which specifies that the distv_min property holds for 'runHelper cl') to obtain a proof that the sum of weight(w, v) and distance of w in 'runHelper cl' is greater than or equal to the distance value of v in 'runHelper cl'. Then by applying ih4 and l3_preserveDelta on the node w, we know that the distance of w in 'runHelper cl' is equal to the length of psw. Hence the sum of weight(w, v) and length of psw, which is exactly the length of psv, is greater than or equal to the distance value of v in 'runHelper cl'[S5]. However by the shortest path property spsv, the length of lpsv, which is equal to distance value of v in 'runHelper cl', is larger than or equal to the length of psv [S6]. Hence by applying the function dgteEq on [S5] and [S6] we know that the distance value of v in 'runHelper cl' is equal to the length of psv. Proof of l5_stm4 is complete.

Lastly, we define the following l5_spath function that proves the preservation of l5_stms properties after calling runHelper.

```
l5_spath : {g : Graph gsize weight ops} ->
           (cl : Column (S len) g src) ->
           (nadj : ((n : Node gsize) ->
                  inNodeset n (getNeighbors g n) = False)) ->
           (l2_ih : neDInfPath cl) ->
           (l5_ih : l5_stms cl) ->
           l5_stms (runHelper cl)
l5_spath cl nadj l2_ih l5_ih
  = (l5_stm1 cl l5_ih,
     l5_stm2 cl l2_ih l5_ih,
     l5_stm3 cl nadj l2_ih l5_ih (l5_stm1 cl l5_ih) (l5_stm2 cl l2_ih
   l5_ih),
     l5_stm4 cl nadj l2_ih l5_ih (l5_stm1 cl l5_ih)
                                 (l5_stm2 cl l2_ih l5_ih)
                                 (l5_stm3 cl nadj l2_ih l5_ih
                                    (l5_stm1 cl l5_ih)
                                    (l5_stm2 cl l2_ih l5_ih)))
```

Function l5_spath states that, the properties specified by l5_ih preserve after calling runHelper on cl. Based on the preservation proof of each statement above, the implementation of l5_spath is quite straightforward and is constructed by applying each preservation proof (function l5_stm1 to l5_stm4) to the input assumption l5_ih.

Discussion on Implementation of Lemma 4.5

Although the above explanation only provides a summary or skeleton of our proof on Lemma 4.5, by comparing with the mathematical proof of Lemma 4.5 in Section 4.5, we notice that the general structure and the reasoning process of the mathematical proof are highly similar to the Idris implementations discussed above. During this verification process, our mathematical proof of Dijkstra's correctness serves as the basis for structuring and implementing most of our lemma proofs, including the implementation of l5_spath discussed above. This suggests the significance of providing a structured and detailed mathematical proof for a program, which lists out and proves all implicit assumptions, before start implementing the verification program.

5.3.2 Verification of Correctness

The lemma proofs implementation discussed in the previous section provides us a function `l5_spath` which states that if the `Column` properties specified by the functions `neDInfPath` (in Section 5.3.1.2) and `l5_stms`(in Section 5.3.1.4) hold for a `Column cl`, then the properties perserve after calling `runHelper` on `cl`. Based on `l5_spath` we define the below function `correctness`, which states that if the first `Column` named `cl` generated in the `dijkstras` function in our Dijkstra's implementation (Section 5.2.1), fulfills the properties stated by `neDInfPath` and `l5_spath`, then the properties reserve after callign `runDijkstras` on `cl`. The function `correctness` is defined by recursively applying the lemmas `l2_existPath` and `l5_spath`, as presented below.

```
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)
```

`nadj` is a function that states a node cannot be in the `nodeset` of itself, which is required by the function `l5_spath`. `l2_ih` and `l5_ih` states that the properties specified by function `neDInfPath` and `l5_stms` hold for the input `Column cl`, and the return type states that `l5_stms` also holds for the `Column` generated by running `runDijkstras` on `cl`. `correctness` is defined recursively. When the input `cl` contains no unexplored nodes, then the input argument `l5_ih` directly provides the definition. Otherwise, since `runDijkstras` calls `runHelper` for updating the `Column` value, we recur the `correctness` function on the new `Column` generated by '`runHelper cl`', and updates the corresponding inputs by applying lemmas `l2_existPath` and `l5_spath` on `cl`.

We then defined the `dijkstras_correctness` function presented below, which wraps up all lemmas and helper proof, 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
```

```
dijkstras_correctness (S len) g src v psv spsv nadj {weight} {ops}
  = (l5_sp {cl=runDijkstras col}
           (correctness col nadj lemma2_ih base_stm))
           v
           (allExp g src nodes dist) psv spsv
   ...
```

The type of dijkstras_correctness states that, given a input graph g and the source node src, for any node v in the graph and a shortest path named psv from src to v in the graph g(specified by the input spsv), the distance value from src to v calculated by running dijkstras function (from our implementation in Section implementation) is equal to the length of the shortest path psv, i.e., is the minimum distance from src to v. The function indexN uses the value carried by the node v to index its distance value from a given Vect, which is the Vect of distance values returned by 'dijkstras gsize g src nadj' in the type of dijkstras_correctness provided above.

To implement the proof dijkstras_correctness, we first construct the first Column cl in the matrix representation of Dijkstra's, in the same way the dijkstras function (Section 5.2.1). Then given two proofs lemma2_ih and base_stms, which state that the properties specified by neDInfPath and l5_stms hold for cl, we apply correctness on cl, lemma2_ih, and base_stms to obtain the proof 'l5_stms (runDijkstras cl)', which states that the properties specified by l5_stms hold for the Column calculated by 'runDijkstras cl'. Hence the fourth statement in l5_stms, which is expDistIsDelta(Section 5.3.1.4), also holds for 'runDijkstras cl'. By applying 'expDistIsDelta (runDijkstras cl)' on the input node v and its shortest path psv, we prove that for any node v in the input graph g, the distance value of v stored in the Vect calculated by 'dijkstras gsize g src nadj' is indeed the minimum distance value from src to v, which verifies the correctness of Dijkstra's algorithm.

Unfortunately, the implementation of lemma2_ih and base_stms is still incomplete, since the current definitions of the mkdists and mknodes functions are hard to work with in implementing the proofs. However the incompleteness does not largely affect the validity of our verification program, as this issue can be resolved by providing a better, easy-to-approach definitions for mkdists and mknodes, and we are confident to provide the full implementation if granted more time.

6 Discussion & Future Work

In our verification program, to prove certain properties of a function, we usually structure the proof based on the implementation of this function. For instance, if a the definition of a function includes a case expression that matches on the result of an intermediate computation, than a proof concerned with this function can be approached by matching on the result of the same computation using the with rule. In other words, if a function that involves more complex data types is built on other functions that involve simpler data types, a proof on this function can generally be approached by breaking the proof into smaller ones based on how the complex data types are destruct in the implementation of the function. A detailed illustration of this technique is provided in the proof of Lemma 4.3 back in Section 5.3.1.3.

We also recognize a parallel between the mathematical proofs of our lemmas and the corresponding Idris implementations. Most of our lemma proofs are implemented following the same structure and reasoning as the mathematical proofs. For instance, the reasoning behind the implementation of `l1_prefixSP` in Section 5.3.1.1 is highly similar the mathematical proof of Lemma 4.1 included in Section 4.1. This indicates that, with a dependent typed language that allows precise specification of data type properties and function behaviors, direct translation of mathematical proofs might be a feasible approach in implementing verification programs.

Future Work We originally intended to verify Dijkstra's and Bellman-Ford algorithms, as both have wide real-life applications in various fields. Due to time limit we were only able to provide a verification for Dijkstra's algorithm with a few incomplete proofs. From the theoretical perspective, verifying Dijkstra's algorithm with the Idris Programming Language is feasible. As we have implemented most lemma proofs, this indicates that the current design and structure of our verification program provides a feasible approach, and that Idris is completely capable for verifying programs. However, we do recognize certain downsides in our algorithm implementation (in Section 5.2) as well as potential errors in Idris. One of the major issues concerns with a `inNodeset` function defined to check whether a node `n` is in the `nodeset` of another node `m`. The definition of `inNodeset` simply compares `n` against every node in the `nodeset` of `m`. When we attempt to match on the value of calling `inNodeset` on any two nodes (denote as `n` and `m`) using the `with` rule, an unclear type error occurs, which states that there is a type mismatch between `warg = warg` and `warg = inNodeset m (getNeighbors g n)`. This unclear type error restricts us from obtaining enough information for completing certain lemma proofs, such as the `l2_existPath` function. In spite of potential limitations, we are confident to complete the verification of Dijkstra's if granted more time.

7 Related Work

The increasing importance of Dijkstra's algorithm in many real-world applications has raised an interest on verifying its 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 [13]. 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 [14], 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 [15][5].

8 Conclusion

This thesis offers a verification for Dijkstra's algorithm with the Idris Programming Language [2]. Our contributions include a detailed mathematical proof on Dijkstra's correctness, a concrete algorithm implementation in Idris, and a verification program with a few incomplete proofs. In the verification process we have obtained many valuable experience and observations. Specifically, we observe a similarity between the mathematical proofs of lemmas and their corresponding Idris implementations, which indicates that with a dependently typed language, by providing precise function types, directly translation of mathematical proofs to verification programs can be a feasible approach. We also recognize a few design issues in our algorithm implementations, and some potential errors in Idris that are accountable for the incompleteness of our verification program, however we are confident to provide the complete implementation with more time granted.

References

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

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

[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] Alt-Ergo. `https://alt-ergo.ocamlpro.com/`, 2011.

[7] CVC4 the smt solver. `http://cvc4.cs.stanford.edu/web/`, 2014.

[8] Z3 Theorem Prover. `https://github.com/Z3Prover`, 2015.

[9] COMPCERT. `http://compcert.inria.fr/`, 2008.

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

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

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

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

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

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