Shortest Path Algorithms Verification with Idris

Yazhe Feng

February 17, 2019

1 Introduction

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

Existing resource on verifying programs that implement Dijkstra's and Bellman-Ford are relatively limited. In most cases the correctness of program relies on the theoretical proof of the underlying algorithms, whereas the verification of program itself remains unattended. Consider the increasing significance of software implementing both algorithms in solving real-world issues, it is important to be able to verify the behaviors and ensure the correctness of programs themselves.

Our work focuses on the verification of Dijkstra's and Bellman-Ford algorithms, which involves two parts. First, we will implement both Dijkstra's and Bellman-Ford algorithms from scratch, which requires defining data structures used in both algorithms(such as node, graph, and prioirty queues), programming basic sorting methods, and then implementing both algorithms based on these definitions. This allows more flexibility in the second part of our work, which is verifying our implementations. We aim to present algorithm verification as a programming issue, hence instead of using verification tools and proof management systems such as the Coq Proof Assistance, both parts of the work will be done with the Idris functional programming language, which embraces nice features such as dependent types that are siginificantly helpful in program verification(Section 3 provides more backgrounds on Idris).

Literature Review

Contributions

Specifically, our contributions are:

1. verified Dijkstra's with Idirs

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

2 Motivation

In this work we focuses on verifying simple programs that implement Dijstra's and Bellman-Ford algorithms. Our implementaion starts with defining data structures used in both algorithms(for instance node, edge, and graph), and involves proving properties of data types such as natural numbers and list. We aim to present verification as a programming issue, showing how properties of data types, behaviors of functions, and correctness of programs can be verified not only through theoretical proofs, but also through implementations.

3 Background

3.1 Dijkstra's Algorithm

3.2 Idris Programming Language

4 High-Level Contribution

4.1 Definitions

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

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

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

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

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

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

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

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

4.2 Dijkstra's Algorithms

4.2.1 Pseudocode

Given input graph g and source node s with types:

g : Graph gsize weight
s : Node gsize

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

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

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

The Dijkstra's Algorithm runs as follows:

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

4.2.2 Assumption

1. Weight of edges are positive
2. Distance value can only be zero, infinity, or summation of edge weights
3. All nodes n and edge e are valid: $n, e \in g$

4.2.3 Proof of Correctness

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

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

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

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

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

Let P(n) be: during the n^{th} iteration of the algroithm for $n \geq 1$, forall node $v \in explored$: (1) $\delta(v) \leq \delta(v'), \forall v' \in unexplored$; and (2) $dist_n[v] = \delta(v)$.

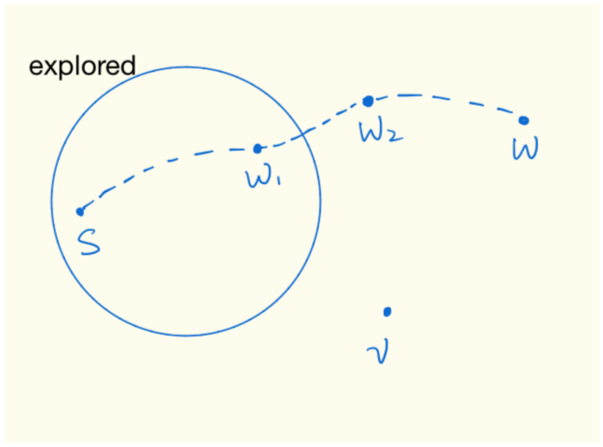
Base Case: We shall show P(1) holds
Based on the algorithm, during the first iteration, the node with minimum distance value is the source node s with $dist_1[s] = 0$. Hence during the first iteration, only s is removed from $unexplored$ and added to $explored$. Since all edge weights are positive, then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

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

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

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

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



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

Then:

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

Since all edge weights are positive, then:

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

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

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

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

(Proof below are not modified yet)

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

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

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

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

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

Proof. **Prove of Correctness**

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

5 Low-Level Contribution

6 Discussion

7 Related Work

8 Conclusion

References

Appendix

Statutory Declaration