Verified Dijkstra's Algorithm

Qingyi He
qingyi.he@epfl.ch

Yicong Luo
yicong.luo@epfl.ch

January 9, 2023

1 Introduction

Graph algorithms are an important class of algorithms in computer science, with numerous applications in fields such as network analysis, machine learning, and bioinformatics. Despite their widespread use, the verification of graph algorithms has received relatively less attention in the literature compared to other commonly used data structures and algorithms. In this project, we aim to implement Dijkstra's algorithm for shortest path with a verified graph data structure. To explore this subject, we firstly refered to John Launchbury's paper [2] and the open-source functional graph libraries. Then, we constructed our graph with stainless and verified Dijkstra's Algorithm with stainless in this project.

In the chapters that follow, we'll start by introducing Dijkstra's shortest path algorithm with a piece of pseudo code, and then proceed to discuss how Dijkstra's algorithm and the graph data structure it uses are implemented purely functionally. After that, we'll go over the various approaches for verifying the correctness and reliability of graph algorithms with stainless. The final section will summarize the project and discuss some possible future works.

2 Preliminaries

Graph Graph is a structure amounting to a set of objects in which some pairs of the objects are in some sense "related". The objects correspond to mathematical abstractions called vertices and each of the related pairs of vertices is called an edge[3].

Dijkstra's Algorithm Dijkstra's Algorithm[1] is an algorithm for finding the shortest paths between nodes in a graph, which doesn't contain any negative-valued edges. This algorithm has many different versions. Dijkstra's original algorithm found the shortest path between two given nodes, but in this project, we focus on a more common variant, which finds shortest paths from a fixed a single source node to all other nodes in the graph, producing a shortest-path tree.

Generally, Dijkstra's algorithm has a flavour similar to the Breadth-first search (BFS), which stores all the possible nodes that could be chosen in the next iteration. But Dijkstra's algorithm always chooses the "best candidate" that is the closest to the source node and perform the update with it.

Here is the pseudo code for Dijkstra's algorithm, the `dist` is an array that records the distance from `source` to current node :

```
def Dijkstra(Graph, source) =
  for vertex v in Graph do
    if v == source then
      dist[v] <- INFINITY
    else
      dist[v] <- 0
    add v to Q

  while Q is not empty do
    u <- vertex in Q with min dist[u]
    remove u from Q
    for neighbor v of u in Q do
      alt <- dist[u] + Graph.Edges(u, v)
      if alt < dist[v] then
        dist[v] <- alt

  return dist
```

3 Implementation

In this section, we will describe how to implement Dijkstra's algorithm and the related data structures with stainless, and how to verify the properties it requires. To build a verified Dijkstra's algorithm in stainless, the first thing we need to do is to translate this algorithm into a functional way. Then, a verified graph in a purely functional way is required.

3.1 Functional Graph

We represent graphs using associate lists of each vertex and its edges to its nearby edges with its distance (or weight). This way fits best in the functional programming, and make it easy to access each vertex's neighbors.

```
case class Graph(graph:
    List[(Int, List[(Int, Distance)])])
```

We defined the "valid graph", to avoid stainless generating invalid graphs (e.g. vertexes that have a non-zero distance to themselves, or edges connecting two non-existing vertexes).

```
def validGraph(
  graph: List[(Int, List[(Int, Distance)])]
): Boolean =
  noDuplicates(graph) &&
  graph.forall(e => noDuplicates(e._2)) &&
  graph.forall(e =>
    e._2.forall((i, _) => graph.get(i) != None())) &&
  graph.forall(n =>
    n._2.forall(z => 0.toDist <= z._2)) &&
  graph.forall { case (n, a) =>
    a.get(n) match {
    case None()  => true
    case Some(d) => d == 0.toDist
    }
  }
```

3.2 The Functional Dijkstra's Algorithm

The (functional) Dijkstra's algorithm resembles the original version, which we first generate the list of distances `Q`, then get into the main loop.

```
// type Node = (Int, Distance)
def dijkstra(source: Int): List[Node] = {
    require(graph.get(source) != None())
    val Q = prepare(source)
    iterate(Nil[Node](), Q)
}
```

3.2.1 Representing distance

The distance used in Dijkstra's algorithm, a distance is either an infinite or a non-negative number. However, the such data structure is not built-in in Scala, so we need to define the following.

```
sealed abstract class Distance
case object Inf extends Distance
case class Real(i: BigInt) extends Distance {
    require(i >= 0)
}
```

We also need to define addition and comparsion bewtween Distance, which is just integer adddition and comparsion with infinity.

3.2.2 A verified `getMin`

One important step in the Dijkstra's algorithm is extracting the node with minimal distance to the source, so we need to build a pure function that return the node that is closest to the source and the rest of the nodes.

```
def getMin(l: List[Node]): (Node, List[Node]) = {
  require(l != Nil())
  l match
    case Cons(h, t) => getMinAux(h, t, Nil[Node]())
} ensuring (res =>
  res._2.size == l.size - 1 &&
    res._2.content ++ Set(res._1) == l.content &&
    res._2.map(_._1).content ++ Set(res._1._1) ==
      l.map(_._1).content &&
    res._2.forall(n => res._1._2 <= n._2)
)
```

Here we would call a helper function `getMinAux` which carries the min value, the traversed list, and the rest to get the result. Ensuring that the remaining list is 1 shorter, the set of the content of the remaining list with min node equals the original content of the original list. Finally, the most important property is that all the nodes in the remaining list should have a longer distance than we min one we get.

In this way, we ensure that the `getMin` function is correctly implemented.

3.2.3 Preprocessing

The first stage of the algorithm is to generate a list of vertexes with their initial distance to the source node, which every vertex should have a distance of infinity and the source itself has a distance 0.

```
def prepare(start: Int): List[Node] = {
    require(graph.get(start) != None())
    prepareAux(graph, start)
} ensuring (res => prepareProp(res, graph, start))
```

This is accomplished in the `prepareAux` function, which is a simple mapping of the vertexes in the graph to the corresponding distance (either infinity or 0).

```
def prepareProp(
    res: List[Node],
    graph: List[(Int, List[Node])],
    start: Int
): Boolean =
  res.size == graph.size &&
    res.map(_._1).content ==
      graph.map(_._1).content &&
    res.map(_._1) == graph.map(_._1) &&
    (res.get(start) match {
      case Some(d) => d == Real(0)
      case None()  => res.forall(_._2 == Inf)
    })
```

3.2.4 Main loop

The main loop also resembles the original version, where we extract the vertex `h` with minimal distance to the source then update all the distances of the vertexes left using `h`.

```
def iterate(
  seen: List[Node],
  future: List[Node]
): List[Node] = {
  /* decreases and requires */
  future match
    case Nil() => seen
    case fu @ Cons(_, _) =>
      val (h, t) = getMin(fu)
      iterate(h :: seen, iterOnce(h, t))
} /* ensurings */
```

The updates are accomplished by the `iterOnce` function. It ensures that the nodes in the updated list will have a shorter distance to the source and all of their distances are larger than `cur`, as otherwise `cur` will no longer be the vertex that is the shortest in the list. This helps us verify that the updates of the distances are implemented correctly.

```
def iterOnce(cur: Node, rest: List[Node]): List[Node] = {
  decreases(rest.size)
  require(rest.forall(n => cur._2 <= n._2))
  rest match {
      case Nil()      => Nil()
      case Cons(h, t) =>
        Cons(updateDist(cur, h), iterOnce(cur, t))
  }
} ensuring (res =>
  res.forall(n => cur._2 <= n._2) &&
  res.map(_._1).content ==
    rest.map(_._1).content &&
  res.size == rest.size &&
  res.zip(rest)
    .forall((y, x) => y._1 == x._1 && y._2 <= x._2)
)
```

This is just a simple `map` over the list `rest`, applying function `updateDist`, which is simply defined as the following.

```
def updateDist(cur: Node, tar: Node): Node = {
  require(cur._2 <= tar._2)
  val nd = cur._2 + distance(cur._1, tar._1)
  (tar._1, if nd <= tar._2 then nd else tar._2)
} ensuring (res => res._2 <= tar._2 && cur._2 <= res._2)
```

The `ensuring` part verified that the node returned will always have as a shorter or equal distances to the source.

4 Possible Future Work

However, we are not able to make it past one of the most important properties of the shortest path algorithm, which is the "triangle inequality". That is, if we choose two random vertexes in the resulting list (of distances), we should have the distance of vertex `m` to the source should be equal to or smaller than the distance of sum of first going to vertex `n` then go to vertex `m`, otherwise the distance of `m` should be smaller (which shows a faulty implementation).

```
def itInv(seen: List[Node]): Boolean =
  seen.forall { case (m, d0) =>
    seen.forall { case (n, d) =>
      n == m ||
      (graph.get(n).flatMap(_.get(m)) match
        case None      => true
        case Some(d1) => d0 <= d1 + d
      )
    }
  }
```

Such invariant should hold for `seen` or `res` in the main loop `iterate`. However, we cannot just *require* or *ensure* such property in stainless, since it will timeout as expected.

To help stainless verify such properties, we still need other lemmas. For example, whenever we add a vertex to the `seen` nodes, we know that the newly added vertex should always have a longer distance to the source vertex, otherwise, we haven't extracted the vertex with min distance last time. But to verify this, we need to we would need extra constraints in our `getMin` function, as it is deeply tied to the min node, we have extracted.

Unfortunately, even though we have attempted, we did not get it past stainless.

5 Conclusion

In this project, we verified Dijkstra's algorithm, a well-known graph search algorithm, with stainless. We began by providing an overview of the algorithm and its key features, and then described the verification process in detail, including the methods and tools we used.

We were able to successfully verify several important properties of Dijkstra's algorithm, such as the correctness of each step, and termination. However, we were unable to prove the triangle inequality, which states that the length of any path between two nodes in a graph is no longer than the sum of the lengths of the two segments obtained by dividing the path at any intermediate node. This is a common challenge in the verification of graph algorithms, as it requires a complex combination of mathematical and logical reasoning.

Overall, our project explored the potential and limitations of stainless as a tool for verifying graph algorithms. We hope that our work will serve as a valuable reference and inspiration for future exploration of this topic in more depth.

6 Appendix

Link to the repository: https://github.com/Meowcolm024/fwlab

The `proj` folder contains documents such as the slides of the presentation, this report, and the abstract and review of the reference paper.

The `projcode` folder contains the code of this project.

References

[1] Andrew Goldberg and Tomasz Radzik. A heuristic improvement of the bellman-ford algorithm. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1993.

[2] John Launchbury. Graph algorithms with a functional flavour. In Johan Jeuring and Erik Meijer, editors, *Advanced Functional Programming*, pages 308–331, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

[3] Wikipedia contributors. Graph (discrete mathematics) — Wikipedia, the free encyclopedia, 2022. [Online; accessed 20-December-2022].