# Formalization of the Bellman-Ford Algorithm for Airspace Applications

Esther Conrad Aaron Dutle
NASA Langley Research Center
{esther.d.conrad, aaron.m.dutle}@nasa.gov

May 28, 2024

**Abstract**

This paper describes the formal verification of one of the most well-known algorithms for finding the shortest path between all vertices in a directed graph, namely the Bellman-Ford algorithm. This formal verification, performed in the Prototype Verification System (PVS), is motivated by two applications in the aerospace domain which use the algorithm for path planning. The first is a pre-flight calculation that uses an adapted version of Bellman-Ford to find a route intended to maximize GNSS availability throughout the flight. The second is a more traditional application intended to find the shortest path between an autonomous aircraft's current position and a goal waypoint, while avoiding regions of space specified by geofences. A novel aspect of this formal verification effort is the inclusion of two distinct models of computation for the algorithm, one being a traditional serial computation, and the other being an explicitly parallel computation. The ability to use parallel computation in the Bellman-Ford algorithm is in fact why it was chosen over other traditionally more performant algorithms, especially for the GNSS application, where the size of the graph makes a purely serial computation infeasible.

## 1 Introduction

From the very inception of graph theory as a field of mathematical study—the Königsberg bridge problem [11]—graphs and networks have been used

as capable models to study routing and path planning. In the aviation domain, these tools can be used to enhance the efficiency, safety, and reliability of flight operations. Representing the airspace as a network of interconnected nodes and edges, these mathematical structures enable precise and dynamic route planning. Connections between vertices can represent distance, fuel consumption, navigational risk, or any number of values of interest. A graph-based approach allows for the application of sophisticated planning algorithms which compute flight paths that optimize these values.

Due to intended use of these algorithms in (potentially autonomous) aircraft, the implementation of such algorithms must be assured to be correct to a high level. Years of research in fundamental graph theory, along with proof and reproof of the basic algorithms involved, make it highly unlikely that there are mistakes in the basic algorithms themselves. On the other hand, any particular implementation or augmentation of an algorithm may contain an error that makes it incorrect. Formal methods, which mechanically apply rigorous mathematical techniques to a digital design (in this case, an algorithm), allow for an additional level of assurance that such an algorithm performs as intended.

This work focuses on the specification and verification of the Bellman-Ford algorithm [5, 13], one of the most well-known shortest path algorithms, in the Prototype Verification System (PVS)[19], a formal specification language with a tightly integrated interactive theorem prover. The main motivation is the implementation and use of this algorithm in several path-planning applications in the airspace domain, particularly one application that allows planning of routes that optimize for navigation capability, and one application that finds a shortest path avoiding geofences. One unique aspect of the formalization is that it verifies the algorithm in both a more traditional serial model of computation, as well as a parallel version of the algorithm.

The rest of the paper is organized as follows. In Section 2, the two main motivating applications are introduced, and the particular application of the Bellman-Ford algorithm in each is explained. In Section 3, a short introduction to formal methods, and in particular PVS, is given. Section 4 details the main results, which include the specification and proof of the general Bellman-Ford algorithm in both serial and parallel format, and the adapted version of the algorithm for navigational optimization. Section 5 discusses related work, and Section 6 concludes the paper.

# 2 Motivating Applications

The main contribution of this work is a formalization of the Bellman-Ford algorithm for finding a shortest path in a directed graph, with a focus on allowing for serial or parallel computation. While this formalization itself is interesting, the work is inspired by two real-world applications of the algorithm at NASA.

## 2.1 Path Planning for GNSS Availability

Global navigation satellite systems (GNSS)—such as the United States' Global Positioning System (GPS), the European Union's Galileo, and many others—are used every day by automobiles, aircraft, and spacecraft to give precise location to their users. The triangulation algorithms used to determine this position rely on receiving at least 4 satellite signals, with increasing accuracy given more signals and/or signals coming from diverse directions. As anyone who has driven through a city with tall buildings can attest, these systems can become unavailable or unreliable at times when there is no clear view to the sky.

For the driver of an automobile, this can be frustrating and confusing. For envisioned future airspace operations, e.g., autonomous medical delivery or air taxi, precise positioning and navigation is safety critical. Incorrect position information could possibly cause in-air collisions with other aircraft or obstacles, endangering humans on-board or on the ground.

To mitigate this risk, NASA has developed a method of path planning for an aircraft which attempts to maximize GNSS availability in urban areas based on the forecast positions of the constellation of satellites [15]. A description of the entire system is beyond the scope of this paper, but the parts of relevance to the Bellman-Ford application are as follows.

**Airspace discretization**: As a first step, the airspace of interest is discretized into a grid of cubes called voxels. These cubes serve as nodes in the graph representation of the airspace. It should be noted that the size of voxels used can greatly affect the accuracy of the model, but too much granularity can make the graph under consideration computationally intractable.

**Connectivity between cubes:** Adjacent voxels within the airspace grid are connected to form a graph. These edges allow traversal between neighboring regions in the graph, so that paths in the graph correspond to

feasible paths in the airspace.

**Risk-based edge weights:** To assign weights to the edges between voxels, a metric based on the perceived *risk* associated with traversing that airspace segment is used. This risk factor is essentially inversely proportional to the number of visible satellites at the location[1]. High satellite visibility indicates lower risk for an operation, resulting in lower edge weights, while low satellite visibility signifies high risk.

With this graph in hand, any path-planning, or particularly *shortest* path algorithm could be used to find a route. But as noted above, for an airspace of any interest, and voxels of any reasonable size, the size of the corresponding graph is easily in the millions. To accommodate this, an algorithm that is amenable to massively parallel processing is needed. Hence the algorithm that was implemented is a modified version of the Bellman-Ford algorithm, because it allows the parallelization as described in Section 4.2.

The graph described above gives a way to find a GNSS-aware route through an airspace at a given moment, but due to the constant motion of satellites, the graph is only valid for a short period of time. Unfortunately, even short flights often last longer than a snapshot accurately models the environment. To address this, a time-series of these risk functions is maintained, and a greedy-style iteration is done on this sequence.

More specifically, given an initial voxel in this graph, a risk function, and a distance parameter (an *epoch*, as a stand-in for time assuming constant speed flight), a risk-minimizing path is calculated from the initial voxel to all other voxels within the epoch distance. At this point, the risk function is updated, and the endpoints of the path segments are used as the initial voxels for the next epoch and risk function. These segments are then concatenated into a final route from the initial voxel to all other voxels.

It is noted that the overall algorithm is not known to be guaranteed to find the shortest path, though it has been experimentally validated on simulated airspaces, and performs well. In addition to the formalization of the basic Bellman-Ford algorithm described in this paper, a formalization of this particular time-dependent adaptation of Bellman-Ford is described in Section 4.3.

[1]The actual *risk* function is not strictly inverse proportion of the number of satellites, but it does decrease as the number of satellites increases.

## 2.2 Path Planning in ICAROUS

The second application of the Bellman-Ford algorithm is for in-flight replanning for an autonomous flight system. NASA's Independent Configurable Architecture for Reliable Operations of Unmanned Systems (ICAROUS, [7, 4]) is a modular system intended to allow for integration of many different capabilities in an autonomous aircraft.

The most basic capability that ICAROUS provides is to maintain and update a list of waypoints for the aircraft to follow, which are then sent to the on-board autopilot. The additional capabilities of ICAROUS are intended to add functionality or enhance safety of the operations that are conducted. Two of the basic additional capabilities that ICAROUS provides are *detect and avoid*, which provides detection, alerting, and maneuver guidance for avoiding other aircraft that are in the operational area, and *geofencing* which provides similar detection and alerting for avoiding (staying in or out) of geofences which are defined as two-dimensional polygons with upper and lower altitude bounds.

The integration of these capabilities is where in-flight path planning comes into play. In a nominal operation, a first set of waypoints that describes an autonomous flight is programmed into the system, which is usually prechecked to verify that geofences and known traffic are avoided. Due to any number of factors—avoiding an unanticipated aircraft, winds altering flight capabilities, a new mission waypoint determined—the autonomous aircraft may have to maneuver in a way that makes it deviate, sometimes significantly, from the original flight plan.

When such a situation arises, having the aircraft fly directly to the next waypoint (whether from the original plan or a newly determined waypoint) is not always possible, particularly if the direct line to the next waypoint crosses through geofences, as in Figure 1. A path planning algorithm based on rapidly exploring random trees [17] was originally used by ICAROUS to create a path between the current state and the next waypoint, but this implementation was determined unstable for general use in many of the simulations and flights that were being conducted, as it was originally created with a particular operational paradigm in mind, leading to some undesirable behavior.

To address this, a basic—but predictable—algorithm was implemented. Essentially, if the direct path to the desired waypoint crosses through a geofenced area, the aircraft would fly directly toward the geofence, and then

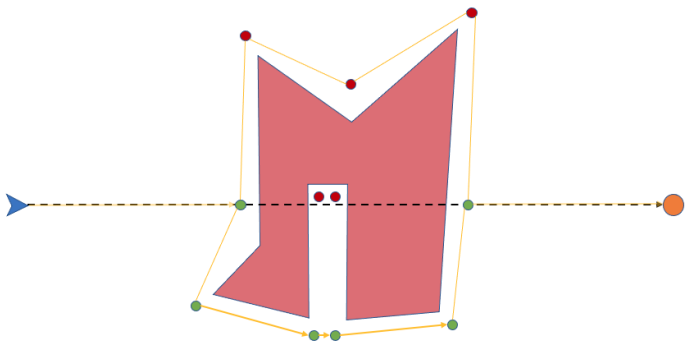

Figure 1: The dotted line indicates the direct path from the current location to the desired waypoint (in orange). Because this intersects the shaded polygonal geofence, the original algorithm flies up to the geofence, and travels along the edge until encountering the original flight path.

travel along the border of the geofence until reaching the place where the original path exits the geofenced area, and then continue along the path, see Figure 1.

While this is easy to implement, and is very reliable, it's far from an optimal solution. To address this shortfall, a graph is built using the—suitably buffered—corners of the offending geofence as well as the beginning and ending waypoints as vertices. For the edges of the graph, if the segment connecting two vertices crosses a geofence, which can be tested using the geofencing application in ICAROUS, the corresponding edge is omitted. Otherwise, a directed edge in each direction is labelled using the physical distance between the corresponding vertices. Then *any* directed path that connects the current waypoint (source) to the goal waypoint (sink) is a path that reaches the desired waypoint without violating the geofence. In particular, a shortest path algorithm such as Bellman-Ford can be used to calculate an optimal one, see Figure 2.

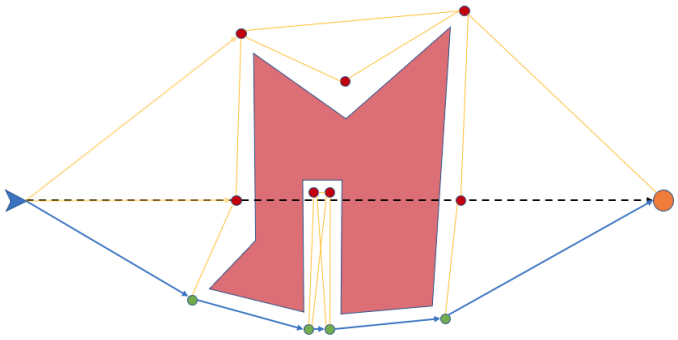

Figure 2: Creating a graph from the polygon vertices and the original and desired waypoints allows for a shortest path algorithm to be employed, finding a much quicker route.

# 3 Formal Verification and Computer-Assisted Theorem Proving

Due to the need for high assurance of correctness for these path planning algorithms, formal verification is employed in their development. Formal verification is a systematic approach that uses mathematically rigorous techniques to ensure that a system adheres to desired properties, behavior, and correctness. While testing is a common way to ensure desired behavior of a system, testing can often be incomplete, and miss cases that are uncommon or unexpected. For systems that are safety-critical, such as those affecting the flight of aircraft over people, or with passengers onboard, ensuring correctness and safety properties is crucial. Formal verification provides methods that can provide assurance on the operation of an algorithm or system on every possible input, where testing cannot.

There are many diverse techniques that can be used for formal verification. Some examples are model checking, which exhaustively checks an entire input space in a systematic and optimized way; static analysis, which examines source or object code to find coding errors; and runtime verification, which alerts at runtime when an expected property has been violated. The tool used in the verification work discussed in this paper belongs to the class of formal methods called computer-assisted theorem provers (or interactive

theorem provers). This class of tools use specialized computer programs and automated tools to assist in writing and proving mathematical theorems. The theorem prover typically provides the framework to express mathematical and logical statements and check the validity of these proofs using an embedded logic framework. It functions as an addition to the human component of proving, checking that the statements are well-formed, and the proofs conform to the deduction rules of the system.

## 3.1 Prototype Verification System

The Prototype Verification System (PVS) is one such interactive theorem prover, used for the formal specification and verification of concurrent and distributed systems as well as software and hardware designs, created and maintained by SRI International [19]. PVS provides an environment for developing formal specifications, constructing mathematical proofs, and verifying system properties. The PVS specification language is a strongly typed functional language based on a classical (i.e., including the law of excluded middle) higher order logic. The system comes with a reasonable collection of predefined theories in what is referred to as the *prelude*, and specifications can import any number of other developments that are available. One of the largest collections of PVS formalizations is NASAlib[2], which is maintained by the NASA Langley Formal Methods Group. In a specification, a user can specify algorithms, data structures, and the majority of other standard programming constructs. Unique to theorem-provers though, is the ability to specify a *Lemma* or *Theorem.* These (interchangeable) keywords in the syntax allow the user to write a statement about the objects previously defined in the specification, and through the interactive console, to prove that the statement is true.

To prove a statement in PVS, a user interacts by entering commands into the theorem-proving console. The console presents a statement to be proven as a list of statements called a sequent. A sequent consists of a list of statements called the *antecedent*, the *turnstile*, and a list of statements called the *consequent*. Such a statement is meant to be interpreted as the conjunction of the antecedent implying the disjunction of the consequent. Each command that a user enters alters the sequent in a way that the result (or conjunction of results in cases where the command generates multiple

[2]https://github.com/nasa/pvslib

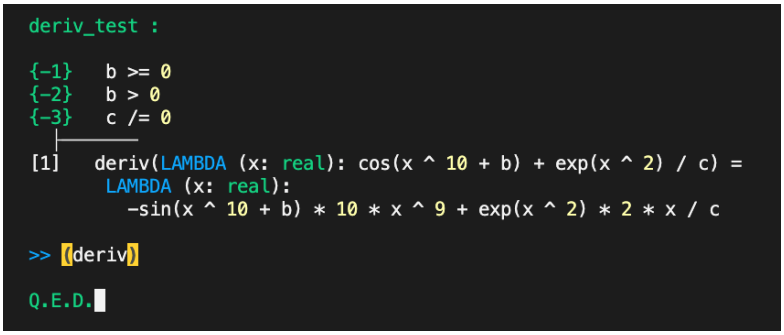


Figure 3: An example of the PVS theorem prover. The (*deriv*) command proves this sequent true in one step.

branches) implies the original statement. This manipulation of the sequent continues until a command is entered that would make TRUE appear in the consequent, FALSE appear in the antecedent, or the same formula appear in both the antecedent and the consequent. Once this is the case for all branches that are created, the formula is considered proven. An example of the PVS prover console and sequent is shown in Figure 3.

# 4 Formalization of Bellman-Ford in PVS

The following sections detail the formalization of the Bellman-Ford algorithm in PVS. First, the basic data structures for graphs and the definitions that enable the specification of the algorithm are described. This includes the defining properties that are true throughout the execution of the algorithm, and that at the end of the algorithm's execution, imply the fundamental shortest path property. Capturing the fundamental aspects of the algorithm in properties of the data allows for diverse implementations to be proven correct in a modular way. Next, the implementation of the algorithm is described, including the serial and parallel computational models. The implementations are each verified to adhere to the defining properties of a Bellman-Ford implementation, and hence to be correct. Finally, the formalization of the adapted version of the Bellman-Ford algorithm used for computing GNSS-aware routes is presented.

## 4.1 Definitions and Structures

The formalization of the Bellman-Ford algorithm requires the implementation of several mathematical structures and definitions in PVS. First, some basic definitions. An *edge-weighted directed graph* $D = (V, A)$ is an ordered pair of sets where $V$ is a set of *vertices*, and $A \subset V \times V \times W$ is a set of *weighted arcs*. The weight set $W$ depends on the context, but is generally some structure with at least an addition operation. If $a = (u, v, w) \in A$, $u$ is called an in-neighbor of $v$ and $v$ is an out-neighbor of $u$. The arc $(u, v, w)$ may be denoted as $u \to v$, and $w$ is called the weight of $u \to v$, sometimes denoted as $u \xrightarrow{w} v$. The set of out-neighbors of the vertex $u$ is the *out-neighborhood* of $u$, denoted $N^+(u)$. The set of in-neighbors of the vertex $u$ is the *in-neighborhood* of $u$, denoted $N^-(u)$. In PVS, the underlying vertex set for these weighted digraphs is chosen to be the natural numbers less than $N$. The values for edge weights, and for use in the Bellman-Ford computation, are drawn from the set extended non-negative reals, denoted $\mathbb{R}^* := \mathbb{R}_{\geq 0} \cup \{\infty\}$

**Definition 4.1** (Weighted DiGraph Representation)**.** Given a vertex set $V = \{0, 1, \ldots, N-1\}$ a function

$$D : V \times V \to \mathbb{R}^*$$

is a *weighted digraph representation.* The value of $D(u)(v)$ is the weight of the edge $u \to v$, where infinite weight represents the non-existence of an edge.

Because this representation returns a value for all pairs, it is convenient to have a definition of what it means to be an actual neighbor, i.e., that the edge $(u, v, w)$ is an arc in the digraph.

$$Neighbor(D, u, v) = \begin{cases} false & D(u)(v) = \infty \\ true & \text{otherwise} \end{cases}$$

A *walk* in $D$ is a sequence of vertices $P = u_0, u_1, u_2, \ldots u_n$, such that for each $i \in \{0, \ldots, n-1\}$, $(u_i, u_{i+1}) \in A$. A *path* is a walk with no repeated vertices. The *weight* of $P$ is $w_0 + w_1 + \ldots w_{n-1}$, where for each $i \in 0, \ldots n-1$, $w_i$ is the weight of $u_i \to u_{i+1}$. An *st-walk* is a walk beginning at $s$, and ending at $t$. Each of these notions has a corresponding specification in PVS.

Next the data structure for holding the information used in the Bellman-Ford algorithm is defined. Recall that the algorithm holds, for each vertex, a record of the weight of the shortest path from the source vertex discovered so far, and the previous vertex on that path. This information is held as a record type in PVS, and a function that returns this record for each vertex.

**Definition 4.2** (Bellman-Ford Data)**.** *BF_Datum* is the set $\mathbb{R}^* \times V \cup \{-1\}$, and an one item of data is denoted $Datum = (w, v) \in BF\_Datum$. The projection function $w$ returns the first value (the *best weight*), and $pv$ returns the second value (the *previous vertex*). *Bellman-Ford Data* is a function

$$Data : V \to BF\_Datum.$$

Note that the previous vertex can be either a vertex or a non-vertex default value of $-1$ if no previous vertex has been assigned. It is also required that a weight is finite if and only if the previous vertex is not the default value. Also, while Bellman-Ford data is defined as a function, it may be referred to as a *set of Bellman-Ford data*.

To capture some of the properties of the data being held, a predicate called *valid* is defined.

**Definition 4.3** (Valid Data)**.** A set of $Data$ on $D$ with source vertex $s$ is said to be *valid*, denoted $valid?(s, D)(Data)$ when $Data(s) = (0, s)$, and for any other vertex $v$, if a $v$ has finite data $(w, u)$ assigned, then the edge $u \to v$ has to exist in $D$, vertex $u$ must have a previous vertex assigned, and there must exist a walk from $s$ to $v$ of weight $w$ with $u$ as the penultimate vertex.

Note that forcing $u$ to have a previous vertex enforces significant structure, as this requirement will recurse through the data. Also note that the predicate enforces the *existence* of a walk of a particular weight, but *not* that this walk is found by following the trail of previous vertices. In general, this stronger statement is not always true, as an update to some vertex along the trail of previous vertices may occur and find a shorter initial segment.

A second predicate on the data is also defined, to capture a measure of optimality or progress.

**Definition 4.4** (Step Valid Data)**.** A set of $Data$ on $D$ with source vertex $s$ is said to be *step valid* for $m \in \mathbb{N}$, and denoted $step\_valid?(s, D)(Data, m)$ when, for each vertex $v$, any walk of length at most $m$ from $s$ to $v$ has weight greater than or equal to the weight held for $v$ in the data.

The Bellman-Ford algorithm will be shown to increase the value of $m$ in this function at each step, eventually reaching $N$, the number of vertices.

If a set of data has both $valid?(s, D)(Data)$ and $step\_valid?(s, D)(Data, N)$ true, then for a vertex $v$ with $Data(v) = (w, u)$, there *is* a walk of weight $w$ using vertex $u$, and $w$ is the smallest weight of *any* walk of length up to $N$. These two properties (plus an assumption that there are no weight-zero cycles) are enough to prove that the data holds the shortest path from $s$ to any vertex $v$ by reading the previous vertices.

The design is intentional, as it allows for diverse implementations of the algorithm to be developed, provided the implementation can meet the validity properties. This is illustrated in the next section, where one serial and one parallel implementation of Bellman-Ford are presented.

## 4.2 Bellman-Ford in PVS

The data structures described it the previous section hold the information needed for the Bellman-Ford algorithm and the properties it should maintain, but do not describe the actual calculation.

The main ingredient in this calculation is a so-called *edge relaxation*. An edge relaxation considers an edge $(u, v, w)$, and determines whether to replace the data being held for vertex $v$. Essentially, if the weight of the currently held path to vertex $u$ plus the weight $w$ is less than the weight currently held for $v$, then this lower weight, and vertex $u$, replace the data for $v$.

The implementation of edge relaxation in PVS is as below, called *edge_update*.

**Definition 4.5** (Edge Update)**.** For $D$ a digraph representation, $Data$ a set of Bellman-Ford data, $Datum = (bw, z)$ and vertices $u$ and $v$, let $nw$ denote the (possibly infinite) value $nw = D(u)(v) + w(Data(u))$. Then

$$update\_edge(D, Data, (bw, z), v, u) := \begin{cases} (nw, u), & nw < bw \\ (bw, z) & \text{otherwise} \end{cases}$$

It is intended that $Data$ holds the current Bellman-Ford data, and $Datum$ holds the current best data for $v$. Note that if $u \to v$ is not an edge, or a path to $u$ isn't known, the current data is kept because the left-hand side of the inequality check is infinite.

Most standard implementations of the Bellman-Ford algorithm proceed by performing a relaxation step on *all* edges in the graph, and then repeating

this $N$ (the number of vertices in the graph) times. Note though, that if the edge relaxation step is done in parallel, there are obvious data-race conditions. Two edge updates for a single vertex using the same original data could begin processing, and both determine that an update should occur. Without a secondary check as to which is better, the relaxation step could set data for this vertex that does not obey the *step_valid*? invariant.

To avoid this, an intermediate step is introduced, where the set of all in-neighbors of a vertex are processed *serially*. This is called a *vertex update*, and is defined as a recursive function in PVS.

**Definition 4.6** (Recursive Vertex Update)**.** For $D$ a digraph representation, $Data$ a set of Bellman-Ford data, $Datum$ a single item of Bellman-Ford Data, and vertices $u$ and $v$, let $UpDatum$ denote the item of Bellman-Ford date $UpDatum = update\_edge(D, Data, Datum, v, u)$. Then define the recursive function

$$update\_vertex\_rec(D, Data, Datum, v, u) := \begin{cases} UpDatum, & u = 0 \\ update\_vertex\_rec(D, Data, UpDatum, v, u-1) & \text{otherwise} \end{cases}$$

In this function, $v$ is a fixed vertex, and the neighbors of $v$ with indices $u$ and below are processed one at a time. The value $Datum$ is intended to hold the current best new data discovered for vertex $v$ in this round.

To process all the neighbors of $v$, this recursive function is called starting with the original data held for $v$ as the current best data, and with the vertex with highest possible index.

**Definition 4.7** (Vertex Update)**.** For $D$ a digraph representation, $Data$ a set of Bellman-Ford data, and $v$ a vertex, define

$$update\_vertex(D, Data, v) := update\_vertex\_rec(D, Data, Data(v), v, N-1)$$

A vertex update is essentially a grouping and ordering of the edge relaxations that might change the data for a particular vertex. Since this function is serial, it is not subject to the data-race conditions mentioned above. Additionally, because this function changes data in only *one* location, the set of all vertex updates can be performed either in serial or in parallel, without risking data collisions.

The serial version of this algorithm is first defined recursively, and then wrapped to execute for all vertices. The function *update_datum* updates the data held in its first parameter only at the location specified by the last parameter. Of note is that the value *NewData* updates the data held for vertex $v$ before performing an update on the next vertex.

**Definition 4.8** (Vertex Round Serial)**.** For $D$ a digraph representation, $Data$ a set of Bellman-Ford data, and $v$ a vertex, let $NewData$ be the set of Bellman-Ford data

$$NewData = update\_datum(Data, update\_vertex(D, Data, v), v),$$

and define

$$vertex\_round\_serial\_rec(D, Data, v) := \begin{cases} NewData, & v = 0 \\ vertex\_round\_serial\_rec(D, NewData, v-1) & \text{otherwise} \end{cases}$$

and

$$vertex\_round\_serial(D)(Data) := vertex\_round\_serial\_rec(D, Data, N-1).$$

For the parallel version defined below, the data used for computing updates is held static throughout the round.

**Definition 4.9** (Vertex Round Parallel)**.** For $D$ a digraph representation, $Data$ and $Data1$ sets of Bellman-Ford data, and $v$ a vertex, let $NewData$ be the set of Bellman-Ford data

$$NewData = update\_datum(Data1, update\_vertex(D, Data, v), v),$$

and define

$$vertex\_round\_parallel\_rec(D, Data, v) := \begin{cases} NewData, & v = 0 \\ vertex\_round\_parallel\_rec(D, Data, NewData, v-1) & \text{otherwise} \end{cases}$$

and

$$vertex\_round\_parallel(D)(Data) := vertex\_round\_parallel\_rec(D, Data, Data, N-1).$$

The parameter *Data1* holds the newly computed data while the parameter *Data* stays constant, holding the data from the previous round.

The remaining structure of the algorithm is the same for both the serial and parallel version—essentially to repeat the round function $N$ times. In order to ensure the modularity of the embedding, and to allow for alternative future implementations, a generic function that updates the Bellman-Ford data is used, called a *Round Function*. This is intended to be used as one round of edge updates, serial or parallel, of the algorithm.

**Definition 4.10** (Round Function)**.** A function that maps one set of Bellman-Ford data to another set fo Bellman-Ford data is called a *Round Function*.

To capture the fundamental properties that a proper implementation of the Bellman-Ford algorithm must satisfy, a predicate is defined that guarantees the *valid*? property is maintained, and the *step_valid*? property progresses.

**Definition 4.11** (Round Validity)**.** For a digraph representation $D$, a vertex $s$ and Bellman-Ford data *Data*, a round function $Round\_fun$ is said to be *Round Valid*, denoted $Round\_fun\_valid?(D, s, Round\_fun)$, when for any natural number $n$,

$valid?(s, D)(Data)$ and $step\_valid?(s, D)(Data, m)$ implies
$valid?(s, D)(Round\_fun(Data))$ and $step\_valid?(s, D)(Round\_fun(Data), m+1)$).

The majority of the time and work done in this formalization was in the proof that each of the implementations described above, *vertex_round_serial* and *vertex_round_parallel*, indeed satisfy *Round_fun_valid*, captured in the pair of lemmas below.

**Lemma 4.12** (Round Valid Serial)**.** *For any digraph representation $D$ and vertex $s$,*

$Round\_fun\_valid?(D, s, vertex\_round\_serial(D))$ *holds.*

**Lemma 4.13** (Round Valid Parallel)**.** *For any digraph representation $D$ and vertex $s$,*

$Round\_fun\_valid?(D, s, vertex\_round\_parallel(D))$ *holds.*

*Proof.* The proofs follow the standard proof of the Bellman-Ford algorithm. It is first shown that any edge update preserves validity. Then assuming step validity for step $m$, any path of length $m + 1$ can be shown to have length greater than or equal to the weight of the best update performed. □

To perform successive rounds of the algorithm, a recursive function that receives an arbitrary round function is defined. This function repeats the execution of the round function a given number of times, with a shortcut exit in the case that the data computed doesn't change from the last round.

**Definition 4.14** (Bellman-Ford Recursion)**.** For a digraph representation $D$, a vertex $s$, Bellman-Ford data $Data$, and a round function $Round\_fun$, let $NewData = Round\_fun(Data)$, and define

$$Bellman\_Ford\_wrap\_rec(D, s, Data, Round\_fun, v)$$
$$:= \begin{cases} NewData, & v = 0 \text{ or } NewData = Data \\ Bellman\_Ford\_wrap\_rec(D, s, NewData, Round\_fun, v-1) \end{cases}$$

The final algorithm is then defined, again for an unspecified round function. Starting with a set of data, $Init(s)$, where the source $s$ has its data set with weight 0 and itself as previous vertex, and all others are set with infinite weight and the default vertex value, the round function is executed (up to) $N$ times.

**Definition 4.15** (Bellman-Ford Wrap)**.** For a digraph representation $D$, a vertex $s$, and a round function $Round\_fun$, define

$$Bellman\_Ford\_wrap(Round\_fun)(D, s)$$
$$:= Bellman\_Ford\_wrap\_rec(D, s, Init(s), Round\_fun, N-1)$$

Next, it is verified that any round function that preserves validity and advances the step validity ensures that the final data has the desired final validity properties.

**Lemma 4.16** (Final Validity)**.** *For a digraph representation $D$, a vertex $s$, and a round function $Round\_fun$, let*

$$FinalData = Bellman\_Ford\_wrap(Round\_fun)(D, s).$$

*Then*

$$Round\_fun\_valid?(D, s, Round\_fun) \text{ implies}$$
$$(valid?(s, D)(FinalData) \text{ and } step\_valid?(s, D)(FinalData, N))$$

## 4.3 Adapted Bellman-Ford in PVS

The Bellman-Ford adaptation that is described in 2.1 is also formalized in PVS. Recall that even though Bellman-Ford has been proved and re-proved to be correct countless times, ensuring validity of the algorithm itself, this does not always guarantee an implementation, or in this case an adaptation of Bellman-Ford, is done properly. In this algorithm, a graph is adapted to represent an airspace, and the edge weights model GNSS availability. Since GNSS availability can change with time, as anyone who may have been caught in a temporary dead zone can attest to, the algorithm is adapted to find paths under these circumstances.

**The graph as an airspace.** To represent the airspace as a graph, the airspace is discretized into voxels. Each voxel center is a triple $(i, j, k)$. Each voxel will have at most 26 neighbors[3]. In the formalization, the airspace is considered as a pre-determined $N \times N \times N$, 3D space. To determine when voxels are neighbors, the difference between the coordinates of the voxels is used. In essence, two voxels are neighbors if each coordinate differs by at most 1.

**Edge weights**: In this application, the goal is to ensure that the GNSS is available at each voxel that we cross in the airspace. For generality, the risk function $E_R$ is formalized as an arbitrary function of voxels and time. We also want to take distance into consideration when flying. Though GNSS availability is important, in an urban environment where the priority is to get to a certain place within a certain time, we consider the edge weights as a function of both $E_R$ and distance. Note that this means that edge weights are a function of time and voxels. This means that as a path is computed, the weights on an edge will change.

In PVS, $E_R$ is specified as a generic type, producing a weight from a voxel and time input.

In the formalization, the risk is an arbitrary function of time and voxels with values from weights in 0 to infinity (inclusive). If a voxel has a value of infinity, it is determined to be unusable at the time.

**Data held at each vertex:** Recall that in Bellman-Ford, each vertex $v$ holds two pieces of data: the best weight found from $s$ to the vertex $v$, and a previous vertex which serves as a guide to find the best path. In the adapted version, each voxel $v$ holds *three* pieces of data: $AR(v)$, $AD(v)$, and $PV(v)$.

[3]An easy visualization is to consider a voxel as the center of a Rubik's cube, and the possible neighbors are the cubes that aren't the center.

$AR(v)$ serves as the *accumulated risk* recorded up to that point, here both the function $E_R$ and the distance is considered. The data in $AD(v)$ serves as the *accumulated distance* up to that point, which considers only the distance travelled. As in the standard version, $PV(v)$ holds the previous voxel in the path discovered so far.

**Edge and Vertex Update:** In the GNSS application, we consider distance as well as GNSS availability to calculate risk in the edges. For a chosen voxel $i$ and a neighbor $n$, the accumulated distance as well as the accumulated risk are used for determining whether to update . Hence the check to replace a piece of data in the adapted version is slightly more complex. If $AR(i) > AR(n) + E_R(i) * distance(i, n)$, or if $AR(i) = AR(n) + E_R(i) * distance(i, n)$ and $AD(i) > AD(n) + distance(i, n)$, then the new data is set as $PV(i) = n, AR(i) = AR(n) + E_R(i) * distance(i, n)$, and $AD(i) = AD(n) + distance(i, n)$. This can be interpreted as using the risk accumulated so far (which includes a measure of distance) as the primary determination of a good path, but if two paths have equal risk, the shorter by distance is preferred.

Another change from the generic algorithm which is implemented in this adaptation is the neighbor selection function. Since each voxel has at most 26 neighbors, when embedding into PVS, a function called *pick_neighbor* is defined, which systematically iterates through these neighbors, taking into consideration the borders of the airspace.

The main new feature of the adaptation is the consideration of time in $E_R$. The adapted version keeps a distance parameter (as a proxy for time) and will perform vertex round steps until the accumulated distance at a vertex reaches the chosen value. Throughout this *epoch* the risk function is assumed to be constant. Once each vertex has stabilized, the risk function $E_R$ is updated, the distance parameter extended for the next epoch, and the algorithm updates proceed.

# 5 Related Work

Without doubt, the most well-known formal verification effort related to graph theory (arguably the most well-known in mathematics) is Gonthier's formal proof in the Coq theorem prover of the four-color theorem [14]. This theorem, stating that every loopless planar graph has chromatic number at

most 4, was first proven by Appel and Haken in 1976 [2, 3] using a combination of standard mathematical proof to reduce to a large number of cases, and custom computation to check each of the cases. This caused some consternation among the mathematical community, because the code itself was not checked by referees, and even if it were, the extraordinary number of cases made it nearly impossible for a reviewer to attest that the entire proof was correct. This sort of situation is exactly where formal verification shines, because a formal proof manages the book-keeping to ensure that even an exceedingly large number of cases cover all possibilities, and the computations themselves can be guaranteed to have been performed correctly as well. Gonthier's proof is an enormous effort and testament to the ability of formal methods in mathematics. The one effort that has (arguably) topped it is Thomas Hales' *Flyspeck* project, which is a formal verification of the Kepler conjecture [16].

There has also been much work done in the theorem-proving community that is more directly related to the work discussed in this paper. Nearly every theorem prover has at least one library of formalizations concerning Graph theory [9, 8, 18, 6]. The level of formalization and number and difficulty of theorems proven varies throughout. The collection of work most closely related to that discussed here is summarized in [1], where the authors discuss a long-term collection of work formalizing graph algorithms and turning these formalizations into executable code. The work in the present paper distinguishes itself by formalizing multiple versions of a single algorithm, the Bellman-Ford shortest path algorithm, under different models of computation (serial and parallel).

There has been related work incorporating graph-theoretic principals into designing and planning of airspace concepts. For example, [10] uses a graph-based approach to both analyze and plan airport location networks in a way that allows for efficient re-routing in the case of a disruption or closure of an airport. The work in [12] examines three different problems—airspace sectorization, airspace sector combination, and precision arrival scheduling—from the lens of graph theory. Path-planning algorithms, including [15] which is one of the primary drivers of the work done in this paper, are widely used in airspace applications, and are inherently graph-based. To the authors' knowledge, though, there is little work that explicitly addresses *formal verification* of graph-theoretic concepts for use in the airspace domain.

# 6 Conclusion

Integrating fundamental graph theory algorithms like Bellman-Ford into formal methods plays a role in enhancing the efficiency and reliability of safety critical systems that heavily rely on these mathematical concepts. Formal methods provides a rigorous framework for verifying and validating the correctness of algorithms, ensuring they meet stringent safety and performance standards. While these mathematical concepts are well-studied, it is in the implementation and adaptation where we must ensure that reliability is upheld.

This work presented was inspired by two real-world applications of the Bellman-Ford algorithm used in aviation at NASA. These applications are thoroughly described in Section 2. Both a parallel version and a serial version of the Bellman-Ford algorithm are embedded and verified formally in the verification tool PVS. The usefulness of this verification shines in the modularity of the embedding. So long as the function meets the necessary properties, the exact implementation of the graph structure and the edge relaxation can be done in a way that fits the needs of the application.

Future work in this topic might consider other graph theory concepts that are or can be applied to safety-critical systems and designs. Having a formal embedding of these well-known concepts and algorithms, which can be directly implemented in applications that require rigorous safety environments, can push the boundaries of autonomous systems and safety mechanics.

# References


[1] M. Abdulaziz, K. Mehlhorn, and T. Nipkow, *Trustworthy graph algorithms*, in International Symposium on Mathematical Foundations of Computer Science, 2019.

[2] K. Appel and W. Haken, *Every planar map is four colorable. i. discharging*, vol. 21, 1977, pp. 429–490.

[3] K. Appel, W. Haken, and J. Koch, *Every planar map is four colorable. ii. reducibility*, vol. 21, 1977, p. 491–567.

[4] S. Balachandran, C. Muñoz, M. Consiglio, M. Feliú, and A. Patel, *Independent configurable architecture for reliable operation*

*of unmanned systems with distributed on-board services*, in Proceedings of the 37th Digital Avionics Systems Conference (DASC 2018), London, England, UK, September 2018.

[5] R. Bellman, *On a routing problem*, Quarterly of Applied Mathematics, 16 (1958), pp. 87–90.

[6] R. W. Butler and J. A. Sjogren, *A pvs graph theory library*, tech. rep., 1998.

[7] M. Consiglio, C. Muñoz, G. Hagen, A. Narkawicz, and S. Balachandran, *ICAROUS: Integrated Configurable Algorithms for Reliable Operations of Unmanned Systems*, in Proceedings of the 35th Digital Avionics Systems Conference (DASC 2016), Sacramento, California, US, September 2016.

[8] Y. Dillies and B. Mehta, *Formalising Szemerédi's Regularity Lemma in Lean*, in 13th International Conference on Interactive Theorem Proving (ITP 2022), J. Andronick and L. de Moura, eds., vol. 237 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2022, Schloss Dagstuhl – Leibniz-Zentrum für Informatik, pp. 9:1–9:19.

[9] C. Doczkal and D. Pous, *Graph theory in coq: Minors, treewidth, and isomorphisms*, Journal of Automated Reasoning, 64 (2020), pp. 795–825.

[10] S. Dunn and S. M. Wilkinson, *Increasing the resilience of air traffic networks using a network graph theory approach*, Transportation Research Part E: Logistics and Transportation Review, 90 (2016), pp. 39–50. Risk Management of Logistics Systems.

[11] L. Euler, *Solutio problematis ad geometriam situs pertinentis*, Comment. Acad. Sci. U. Petrop, 8 (1736), p. 128–140.

[12] A. H. Farrahi, A. T. Goldberg, L. Bagasol, and J. Jung, *Applying Graph Theory to Problems in Air Traffic Management*.

[13] L. R. Ford, *Network Flow Theory.*, RAND Corporation, Santa Monica, CA, 1956.

[14] G. Gonthier, *Formal proof—the four-color theorem*, vol. 55, 2008, p. 1382–1393.

[15] J. Gutierrez, N. A. Neogi, D. Kaeli, and E. T. Dill, *A High-Performance Computing GNSS-aware Path Planning Algorithm for Safe Urban Flight Operations*.

[16] T. C. Hales, M. Adams, G. Bauer, D. T. Dang, J. Harrison, T. L. Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. T. Nguyen, T. Q. Nguyen, T. Nipkow, S. Obua, J. Pleso, J. Rute, A. Solovyev, A. H. T. Ta, T. N. Tran, D. T. Trieu, J. Urban, K. K. Vu, and R. Zumkeller, *A formal proof of the kepler conjecture*, arXiv, 1501.02155 (2015).

[17] S. M. LaValle, *Rapidly-exploring random trees : a new tool for path planning*, The annual research report, (1998).

[18] L. Noschinski, *A graph library for isabelle*, Mathematics in Computer Science, 9 (2015), pp. 23–39.

[19] S. Owre, J. M. Rushby, and N. Shankar, *Pvs: A prototype verification system*, in International Conference on Automated Deduction, Springer, 1992, pp. 748–752.