Why3 and Proving A* Automatically

A Case Study of Why3 as a Tool for Automated Software Verification

Kajetan Neumann[1]

Supervisor(s): Benedikt Ahrens[1]**, Kobe Wullaert**[1]

[1]EEMCS, Delft University of Technology, The Netherlands

A Thesis Submitted to EEMCS Faculty Delft University of Technology,
In Partial Fulfilment of the Requirements
For the Bachelor of Computer Science and Engineering

22nd June 2025

Acknowledgements

Special thanks to Dr. Benedikt Ahrens and Kobe Wullaert for their feedback during the project. We also wish to thank Jeroen Koelewijn, Mohammed Balfakeih, Dinu Blanovschi, and Tejas Kochar, for their help and input. Finally, many thanks to Jean-Christophe Filliâtre for their proof of Dijkstra's algorithm and for assistance with questions regarding Why3.

Keywords: Why3, WhyML, automated formal verification, software verification, A-Star algorithm

Name of the student: Kajetan Neumann
Final project course: CSE3000 Research Project
Thesis committee: Benedikt Ahrens, Kobe Wullaert, Maliheh Izadi

Contents

Abstract

Formal verification of software can provide a more rigorous guarantee of correctness compared to conventional software testing methods. However, doing this by hand requires substantial effort and is often impractical. To combat this, various verification tools have been developed in recent decades to at least partially automate this process. In this paper we explore Why3, a tool for deductive program verification, by implementing and verifying the A* algorithm. We find that Why3's expressive language allows for easy implementation and verification of A*. However, we also find that it has a significant learning curve and requires some knowledge on formal verification to use. In spite of this, we find it is a useful tool for automated verification.

1 Introduction

As Edsger W. Dijkstra said, "Program testing can be used to show the presence of bugs, but never to show their absence!" [1, pg.7]. For this reason, formal software verification is often the preferred approach for ensuring correctness of software in fields like security, aerospace, and low-level software engineering. *Formal verification*, is the process of mathematically checking that a program's behaviour adheres to some predefined properties, via a formal model of the program [2]. This process can be partially automated using *automated theorem provers* (ATPs) or *satisfiability solvers*, collectively referred to in this paper as *automated provers*. These automated provers operate on a subset of first-order logic (FOL), and try to prove a proposition by refuting its negation without human intervention [3]. One such tool that utilises this approach is Why3, a platform for deductive program verification [4–6].

As part of a larger project on automated formal verification tools, this paper provides an overview of the utility of *Why3* in automated software verification. We present a case study of Why3, by verifying the correctness of the A star algorithm (A*), a heuristic-based shortest-path-finding algorithm [7]. In the process, we seek to outline the capabilities and limitations of Why3 as an automated software verification tool. The primary motivation for choosing A* stems from its close relation to Dijkstra's shortest-path-finding algorithm, which has been previously verified in Why3 by Jean-Christophe Filliâtre [8, 9]. Additionally, we believe the complexity of A* provides a good test of the capabilities of Why3.

The outline of this paper is as follows. In Section 2, we introduce the Why3 tool and explain how it works. Section 3 formalises the specification of A* and correctness criteria we endeavoured to implement. Section 4 outlines the implementation of the specification from Section 3 in WhyML, Why3's native language. Section 5 documents observations we made on Why3, as well as the results collected from our implementation. In Section 6 we outline our considerations towards maintaining responsible research practices, such as the reproducibility of our results and how one might go about doing so. Finally, Section 7 discusses our conclusions and suggestions for future work.

1.1 An Introduction to Automated Verification

Formal verification is a method of verifying correctness of a program through formal mathematical reasoning. In 1998, Tony Hoare proposed a way to formally define programming languages, allowing for formal reasoning about the results of programs [10]. A major practical limitation of this approach was that these proofs needed to be done by hand. As such, various systems have been developed to partially automate this approach. The three major categories of these systems, collectively referred to as provers, are:

- *Interactive Theorem Provers* – These include systems like Rocq [11], and Isabelle [12]. These systems provide an expressive language, which is used by the user to write proofs. These proofs are then verified by a computer.
- *ATPs* – These include systems like Vampire [13], E Theorem Prover [14], and SPASS [15]. These systems operate on a subset of FOL. Through logical reasoning, they attempt to prove a proposition by deriving a contradiction from its negation.
- *SMT Solvers* – This family of solvers includes Alt-Ergo [16], CVC5 [17], and Z3 [18]. Similarly to ATPs, they operate on FOL with some extensions and operate on a negation of the given premise. They search for a model satisfying a set of clauses.

Although interactive theorem provers usually provide a more expressive language, they still require the user to write a proof almost entirely by hand. On the other hand, automated provers like ATPs and SMT solvers, although requiring much less manual proving, lack in their expressivity. Why3 utilises multiple provers, a notion we elaborate on in Section 2.

2 The Why3 Platform

Why3 is a modular proof system developed for deductive program verification using a high degree of automation [3–6]. It was introduced in 2010 as a replacement for an earlier Why version, providing a more expressive language and an API to allow for easier extension [3]. Why3 is under active development by INRIA in the Toccata project [4].

2.1 Introduction to Why3

Why3 provides an expressive language (WhyML) for specification and programming. To verify programs, Why3 offloads the proofs to external theorem *provers*, which can be interactive or automated (ATPs and SMT Solvers) [19]. It does so by translating WhyML programs into a purely logical language (called Why) [3]. Since provers don't necessarily support the same logical constructs as Why3, Why code is converted into a subset logic, interpretable by the given prover, through *logical transformations*. Of particular interest for this paper is Why3's ability to prove things almost entirely automatically.

2.2 Interfacing with External Provers

Why3's proving process revolves around proving *verification conditions* (VCs), sometimes referred to as *proof tasks* [19]. These are generated from various parts of the code through

specific keywords, which we discuss in more detail in Section 2.3. A task has a *context*, consisting of the predicates, functions, and lemmas that were specified before, and a *goal* that must be proven. To interface with a given prover, Why3 uses a *driver* file [20]. This is a text file that contains information, such as how to pretty-print the prover's output, or which axioms and theories are built-in to the prover [19]. Most importantly a *driver* specifies how to translate Why logic into a subset which can be understood by the prover. This is done through sequentially applying Why3's *logical transformations*. For example, `unfold` is used to unfold a predicate/function definition within a goal or premise. These transformations can also be used by the user to guide the proving process. In fact, this is often necessary to help the automated provers succeed [3]. A commonly occurring example is the `induction_pr` transformation, which attempts to split the task into base and inductive cases based on some premise.

2.3 The WhyML Language

Why3 is based on a custom extension of first-order logic (FOL), by introducing constructs like polymorphism, inductive predicates, algebraic data types, and recursive definitions [19]. These constructs can be grouped into three main categories:

Logical Expressions In Table 1 we describe some of the logical constructs WhyML provides for expressing logic. Besides basic logical operators, WhyML also provides more advanced constructs. One notable example is `function`, which is used to define logical functions. These functions can be defined recursively, like `path_weight` in Why3's standard library in the `graph` module [21]. Another notable construct supported by WhyML is inductive predicates, defined by a base and inductive case via the `inductive` keyword. An example of this is the definition of `path` in Listing 2.

Keyword(s)	Description
`/\`, `\/`, `not`	Conjunction, disjunction, negation.
`->`, `<->`	Implication and bi-implication.
`forall`, `exists`	Universal and existential quantifiers.
`function`	Used to define logical functions.
`predicate`	Used to define simple predicates.
`inductive`	Used to define inductive predicates.
`constant`	Introduces some arbitrary constant.
`axiom`	Introduces a logical axiom.

Table 1: Common WhyML keywords used in logic.

Program and Type Expressions WhyML provides various programming constructs, some of which are listed in Table 2. This includes if statements, loops, and pattern matching. A notable construct is `let..in..`, which can be used to introduce program methods, like the `astar_code` in Listing 7 on line 1. WhyML also supports the definition of data types using the `type` keyword. This includes simple constructor-based definitions, but also record types with `mutable` fields.

Keyword(s)	Description
`&&`, `||`, `not`	AND, OR, and NOT operators.
`for..to..do..done`	For loop.
`while..do..done`	While loop.
`if..then..else..end`	If statement.
`begin..end`	Code block.
`let..in..`	Let binding.
`match..with..end`	Used for pattern matching.
`ghost`	Marks code as "ghost code", which is only added for verification purposes.
`type`	Introduces a new data type.
`abstract`	Makes all fields in a record accessible only in ghost code.
`mutable`	Makes a record field mutable.

Table 2: Common WhyML keywords used in programs.

Verification Expressions As mentioned in Section 2.2, Why3 works on the basis of proving individual verification conditions (VCs). These are generated when using specific keywords in WhyML, listed in Table 3. The simplest of these is the `goal` keyword, which declares a VC. For example, `goal: forall n: int. n * n >= 0` generates a VC with the goal $\forall n \in \mathbb{Z}.\ n^2 > 0$. Similarly, `lemma` specifies a goal which can be used as a premise in later proofs. Other interesting keyword are `variant` (used to prove the termination of loops) and `invariant` (generates a proof that the a premise hold initially and is preserved after an arbitrary iteration of the loop).

A key feature of verification expressions in Why3 is the ability for users to define specifications of logic or programs without providing an implementation. This is often done via the `val` keyword along with the `requires` and `ensures` keywords. In this paper, we refer to this feature as the *specification-only* approach.

Keyword(s)	Description
`assert`	Asserts that a statement holds on this line.
`requires`	Introduces a function pre-condition.
`ensures`	Introduces a function post-condition.
`diverges`	Marks a function as nonterminating.
`writes`	Lists variables mutated by this function.
`invariant`	Introduces a loop invariant.
`variant`	Introduces a decreasing variant which proves loop termination.
`old`	Used in condition to reference the initial state of a mutable variable.

Table 3: Common WhyML keywords used for verification.

2.4 Applications of Why3

The Toccata project has a large repository of various proofs in Why3, including data structures, puzzles, and algorithms [9]. It also contains a proof of Dijkstra's algorithm, which was the inspiration for proving A* in this paper. Outside of this, there are also various other programs verified using Why3, like a subset of the Go concurrent programming language [3]. Why3 is also used for the verification of Ada programs by the SPARK language [22, 23].

3 Formal Specification of A*

In the following section, we specify the formal definition and properties of A* which we attempted to prove using Why3.

3.1 Graph and Paths

A *finite weighted directed graph* is defined as a tuple $G = (V, E)$, where V is the finite set of vertices, and E is the set of directed edges. We also define a $successors$ function, such that,

$$\forall a, b \in V.\ b \in successors(a) \leftrightarrow (a, b) \in E \tag{1}$$

A $Path(a, l, b)$ through the graph G from vertex a to vertex b is defined by,

$$\begin{aligned} &l = (n_0, n_1, ..., n_{m-1}, n_m), \\ &(\forall i \in [0, m].\ n_i \in V),\ \ n_0 = a,\ \ n_m = b, \\ &(\forall i \in [1, m].\ (n_{i-1}, n_i) \in E) \end{aligned} \tag{2}$$

The *path weight* of l is given by the following function,

$$\mathbf{w}(l) = \sum_{i=1}^{m} w(n_{i-1}, n_i) \tag{3}$$

where the *edge weights* are $w(n_{i-1}, n_i) > 0$. By extension, we can define $ShortestPath(a, l, b)$ via the condition,

$$Path(a, l, b) \wedge (\forall l'.Path(a, l', b) \rightarrow \mathbf{w}(l) \leq \mathbf{w}(l')) \tag{4}$$

3.2 Heuristic and Distance Functions

A* uses a function $\tilde{f}(n)$, which gives the estimated distance from the source vertex s to the destination vertex d, through the vertex n. It is defined as:

$$\begin{aligned} &\tilde{f}(n) = \tilde{g}(n) + \tilde{h}(n) \\ &\exists l_s.\ \tilde{g}(n) = \mathbf{w}(l_s) \wedge Path(s, l_s, n) \\ &\exists l_d.\ \tilde{h}(n) = \mathbf{w}(l_d) \wedge Path(n, l_d, d) \end{aligned} \tag{5}$$

We refer to $\tilde{g}(n)$ as the *distance function* and $\tilde{h}(n)$ as the *heuristic function*. Their non-estimated counterparts are $f(n)$, $g(n)$ and $h(n)$, where $g(n)$ is the weight of the shortest path from s to n, and $h(n)$ is the weight of the shortest path from n to d. Formally, this is:

$$\begin{aligned} &f(n) = g(n) + h(n) \\ &\exists l_s.\ g(n) = \mathbf{w}(l_s) \wedge ShortestPath(s, l_s, n) \\ &\exists l_d.\ h(n) = \mathbf{w}(l_d)n \wedge ShortestPath(n, l_d, d) \end{aligned} \tag{6}$$

The distance function is updated within the iterations of A* as it expands new nodes. The heuristic function is a parameter, and is derived from the problem domain. The heuristic must be *positive*, *admissible*, and *consistent*. These properties are defined as follows:

$$\begin{aligned} positive(\tilde{h}) &\leftrightarrow \forall n.\tilde{h} \geq 0 \\ admissible(\tilde{h}) &\leftrightarrow \forall n.\tilde{h}(n) \leq h(n) \\ consistent(\tilde{h}) &\leftrightarrow (\forall (a, b) \in E.\ \tilde{h}(a) \leq w(a, b) + \tilde{h}(b)) \\ &\quad \wedge \tilde{h}(t) = 0 \end{aligned} \tag{7}$$

3.3 OPEN and CLOSED Sets

A* flags vertices as *open* and can later flag them as *closed* [7]. In order to keep track of which nodes are open and which are closed, it defines two sets: *OPEN* and *CLOSED*. A* uses the *OPEN* set to retrieve the open vertex with the smallest $\tilde{f}(n)$.

3.4 The A* Algorithm

A* is an *informed search algorithm* which finds the weight of the shortest path from a given source s to a given destination d. It functions similarly to Dijkstra's algorithm, but it chooses vertices slightly differently [24]. Dijkstra prioritises vertices based only on $\tilde{g}(n)$, while A* extends this with $\tilde{h}(n)$ to estimate the rest of the distance to the destination, allowing it to prioritise vertices based on the entire distance through the given vertex $\tilde{f}(n) = \tilde{g}(n) + \tilde{h}(n)$. The algorithm follows the pseudocode in Algorithm 1, and functions as follows [7]:

1. Mark s as "open" and calculate $\tilde{f}(s)$ (lines 2-4).
2. Select the open node n with the smallest $\tilde{f}(n)$ (line 6).
3. If $n = d$, close n and terminate (lines 7-9).
4. Otherwise, mark n closed. For each successor of n, calculate $\tilde{f}(n)$ and mark as open each successor not already marked closed (lines 11-16). Then, go to Step 2 (line 5).

Algorithm 1: The A* Algorithm

```
1  function AStar(s, d)
2      OPEN ← {s}
3      CLOSED ← ∅
4      g̃(s) ← 0
5      while OPEN ≠ ∅ do
6          n ← arg min_{v∈OPEN} f̃(v)
7          OPEN ← OPEN \ {n}
8          CLOSED ← CLOSED ∪ {n}
9          if v = dst then
10             return g̃(u)
11         for u ∈ successors(n) do
12             new_g̃ ← g̃(n) + edge_weight(n, u)
13             if u ∉ CLOSED then
14                 if u ∉ OPEN or new_g̃ < g̃(u) then
15                     g̃(u) ← new_g̃
16                     OPEN ← OPEN ∪ {u}
17     return No such path exists.
```

3.5 Properties to Prove

We sought to prove the correctness of our implementation by proving the following properties, which we know should hold for the algorithm.

Optimal Substructure A subset of a shortest path, must also be a shortest path:

$$\begin{aligned} &ShortestPath(n_0, (n_0, ..., n_a, n_b, ..., n_m), n_m) \\ &\rightarrow ShortestPath(n_0, (n_0, ..., n_a), n_a) \\ &\quad \wedge ShortestPath(n_b, (n_b, ..., n_m), n_m) \end{aligned} \tag{8}$$

Consistency Implies Admissibility A consistent heuristic is also an admissible heuristic [7]:

$$consistent(\tilde{h}) \rightarrow admissible(\tilde{h}) \tag{9}$$

Termination and Completeness A* terminates. Either no path exists and all reachable nodes were closed, or the result r is the shortest distance:

$$\begin{aligned} &(\forall n, l.\ Path(s, l, n) \leftrightarrow n \in CLOSED) \\ &\vee (\exists l.\ ShortestPath(s, l, d) \wedge (r = \mathbf{w}(l))) \end{aligned} \tag{10}$$

Optimal Efficiency A* closes all nodes n where $\tilde{f}(n)$ is less than the weight of the shortest path from s to d:

$$\forall n.\ \tilde{f}(n) < f(d) \leftrightarrow n \in CLOSED \tag{11}$$

Minimal Expansion At any given moment, the shortest path has been found for all closed vertices. This can be expressed as the following invariant:

$$\forall n \in CLOSED.\ g(n) = \tilde{g}(n) \tag{12}$$

Open Optimum For any shortest path from the source (s) to some n, if n is not closed by A*, then there must exist an open vertex n' on this path, where the $\tilde{g}(n') = g(n')$. This is the "Lemma 1" proven in the original paper of A* [7], and can be defined by the invariant:

$$\begin{aligned} &\forall b \notin CLOSED.\ \forall l = (n_0, ..., n_m). \\ &\quad ShortestPath(s, l, b) \\ &\quad \rightarrow (\exists i.\ \tilde{g}(n_i) = g(n_i) \wedge n_i \in OPEN) \end{aligned} \tag{13}$$

4 Implementation of A* in WhyML

In this section, we describe key aspects of our implementation, available on GitHub[1] and in Appendix C.

4.1 Graph and Paths

To define a graph in WhyML, we used the same approach as in the proof for Dijsktra's algorithm [9]. The exact implementation can be seen in Listing 1, where `graph: fset vertex` (line 5) is the finite set of vertices and `successors` (lines 7-9) in the *successors* function of the graph. It is marked with the `ghost` keyword, such that it can be used in logical expression. We also define `successors_impl` (lines 11-12), which functions like `successors` but can be used directly in program code. This is required since WhyML restricts where different function types can be used (logic vs. programs). Listing 1 also shows the definition of the `edge` predicate (line 14), which is defined to link the standard library definition of paths to our implementation. Additionally, it shows the `Positive_weight` axiom (line 20) which ensures weights (as defined in the standard library code) are always greater than zero.

```
1  type vertex
2
3  clone set.SetImp with type elt = vertex
4
5  constant graph: fset vertex
6
7  val ghost function successors (x: vertex): fset vertex
8      ensures { subset result graph }
9      ensures { not mem x result }
10
11 val successors_impl (x: vertex): set
12     ensures { result = successors x }
13
14 predicate edge (a b: vertex) = mem b (successors a)
15
16 clone graph.IntPathWeight with
17     type vertex = vertex,
18     predicate edge = edge
19
20 axiom positive_weight: forall a, b. weight a b > 0
```

Listing 1: Definitiont of paths taken from the graph.Path module in the Why3 Standard Library [21].

In order to avoid redefining concepts and lemmas from scratch, we used the definition of paths provided in Why3's Standard Library [21]. Paths are therefore defined inductively as a list of vertices, as seen in Listing 2. The base case is a nil path (`Path_empty`, lines 2-3), and the inductive case defines extending paths from the front (`Path_cons`, lines 4-6). It is important to note that, as defined in the standard library, the list never contains the final vertex on the path. In the context of our implementation, this allows easier appending of paths by simply appending the two lists. The standard library also defines a `path_weight` function, which is defined recursively over a list of vertices and returns the cumulative weight of the edges between them [21].

```
1  inductive path vertex (list vertex) vertex =
2    | Path_empty:
3        forall x: vertex. path x Nil x
4    | Path_cons:
5        forall x y z: vertex, l: list vertex.
6        edge x y -> path y l z -> path x (Cons x l) z
```

Listing 2: Definitiont of paths taken from the graph.Path module in the Why3 Standard Library [21].

We defined shortest paths using the aforementioned `path` predicate and `path_weight` function, as can be seen in Listing 3. We say that the list must be a valid path (line 2), and for all other paths, their weight does not exceed this path's weight (lines 3-4) – exactly as defined in Equation (4).

```
1  predicate shortest_path (a:vertex) (l:list vertex) (b:vertex) =
2      path a l b /\
3      (forall l'. path a l' b ->
4          path_weight l b <= path_weight l' b)
```

Listing 3: Definitiont of shortest paths in WhyML.

We also introduced two useful predicates for reasoning about paths and shortest paths using their path weight rather then a list of nodes. The main motivation behind this is from the proof for Dijkstra's shortest path algorithm mentioned before [9]. The definition of these predicates can be seen in

[1] https://github.com/kmneumann/Why3-A-Star.git

Listing 4. We utilise the `exists` keyword to say that there exists a path of the given weight (lines 1-2). Similarly, we define a predicate like this for shortest paths (lines 4-5).

```
1  predicate path_with_len (a b:vertex) (d:int) =
2      exists l. path a l b /\ (path_weight l b = d)
3
4  predicate shortest_path_with_len (a b:vertex) (d:int) =
5      exists l. shortest_path a l b /\ (path_weight l b = d)
```

Listing 4: Definitons of path (top) and shortest path (bottom) based on their distance rather a list of nodes.

4.2 Heuristic and Distance Functions

In our implementation, we chose to have the *heuristic* function be a parameter of the A* algorithm. This allowed for greater control in pre-/post-conditions. In order to reason about this function, we defined four predicates seen in Listing 5. The first is the `positive` predicate (lines 1-2), which ensures the function only returns positive values. The second predicate (`admissible` on lines 4-5) defines admissibility of a function for a specific destination node. It does so using `path` and `path_weight`, mentioned previously. Lastly, we defined `consistent` and `path_consistent` (lines 7-11 and 13-17), which both give alternate but equivalent definitions of consistency. The `consistent` predicate defines it traditionally as mentioned in Section 3. On the other hand, `path_consistent` defines consistency over a whole path rather than a single edge. This definition seemed to help the provers reason about the implication of consistency and path finding, which we discuss further in Section 5. These two definitions are equivalent; a fact we were able to prove with a lemma in our implementation.

```
1  predicate positive (f: vertex -> int) =
2      forall n. f n >= 0
3
4  predicate admissible (f: vertex -> int) (dst: vertex) =
5      forall a, l. path a l dst -> f a <= path_weight l dst
6
7  predicate consistent (f: vertex -> int) (dst: vertex) =
8      f dst = 0 /\
9      (forall a b:vertex.
10       edge a b ->
11       f a <= weight a b + f b )
12
13 predicate path_consistent (f: vertex -> int) (dst: vertex) =
14     f dst = 0 /\
15     (forall a b:vertex, l: list vertex.
16       path a l b ->
17       f a <= path_weight l b + f b )
```

Listing 5: The *positive* (lines 1-2), *admissible* (lines 4-5) and *consistent* (lines 7-11 and 13-17) predicates defined in WhyML.

The *distance* function was expressed exactly like in the proof for Dijkstra's algorithm [9]. We took from this proof the definition of a mutable map type (called `mutMap`), which we used to keep track of the currently found shortest distance from the source to a given vertex (i.e. $\tilde{g}(n)$). The functions we utilised in the code for A*, described later in this section, is the `[]` functions for querying value for a given key, and the `[] <-` function which assigns a value to a given key. The WhyML module defining these functions can also be seen in Appendix A.

4.3 OPEN and CLOSED Sets

The $OPEN$ and $CLOSED$ sets is defined using the `set.SetImp` module in Why3's Standard Library. It provides an implementation of a mutable set. Additionally, for the $CLOSED$ set, we defined functions that allow us to retrieve the vertex with the smallest $\tilde{f}(n)$ score. These are expressed in WhyML as seen in Listing 6. The `min` predicate (lines 1-3) is used to assert that a given vertex is a vertex with the smallest $\tilde{f}(n)$ in the given set, calculated using the distance map (`d`) and the heuristic function (`h`). Then, we used the `val` keyword to define the `get_min` function via a specification-only approach (lines 5-9). It ensures that the returned vertex is the minimum and that it is removed from the `open` set. This is modelled almost exactly like the `visited` set in the proof for Dijkstra's algorithm [9].

```
1  predicate min (m:vertex) (q:set) (d:mutMap int) (h:vertex->int) =
2      mem m q /\
3      (forall x: vertex. mem x q -> d[m] + h m <= d[x] + h x)
4
5  val get_min (open:set) (d:mutMap int) (h:vertex->int) : vertex
6      writes   { open }
7      requires { not is_empty open }
8      ensures  { min result (old open) d h }
9      ensures  { open = remove result (old open) }
```

Listing 6: Definition of a function (bottom) to take the minimum vertex from a set. The predicate (top) is used to define minimum.

4.4 The A* Algorithm

Due to WhyML supporting various common programming constructs, we were able to express the algorithm almost entirely like in the original pseudocode shown in Algorithm 1. Our implementation, as seen in Listing 7, deviates from the pseudocode definition in only a few minor ways. Firstly, the function returns an `option`, rather than an integer (end of line 1). This is done so that we can encode the case where no path from source to destination exists (line 23). Secondly, as mentioned before, we chose to pass the heuristic function as a parameter to the function (`h: vertex -> int` on line 1).

For the sake of readability, we removed the preconditions, postconditions, invariants, variants, and assertions from Listing 7. We cover those specifically in the next subsection.

```
1  let astar_code (src dst: vertex) (h: vertex -> int): option int
2    = let closed: set = empty () in
3      let open: set = singleton src in
4      let d: mutMap int = create 0 in
5      while not is_empty open do
6          let u = get_min open d h in
7          if eq u dst then begin
8              return Some d[u]
9          end;
10         add u closed;
11         let su = successors_impl u in
12         while not is_empty su do
13             let y = choose_and_remove su in
14             let x = d[u] + weight u y in
15             if not mem y closed then begin
16                 if not (mem y open) || x < d[y] then begin
17                     add y open;
18                     d[y] <- x
19                 end
20             end
21         done
22     done;
23     return None
```

Listing 7: WhyML implementation of A*, without verification code.

4.5 Properties to Prove

We expressed the properties of A* listed in Section 3 using lemmas, variants, invariants, assertions, and pre-/post-conditions. Some properties are expressed through more than one of these constructs. In this section, we detail how we introduced each property into our implementation.

Besides the properties listed below, we also defined additional lemmas, invariants, variants, and pre/post-conditions to facilitate proving. For example, we defined a pre-condition that the heuristic must be consistent via the `requires` keyword. To improve readability of code, we grouped many invariants into single predicates (like `inv`, `inv_succ`, and `inv_succ2`). The code base has comments describing what each expresses in more detail, as visible in Appendix C (lines 215-262). Most of these were taken from the proof for Dijkstra's algorithm [9], and annotated by us.

Optimal Substructure We express this property, as seen in Listing 8, by showing that any shortest path from s to t consisting of three sections (s-a, a-b, and b-t) implies that the section from a to b is also a shortest path from a to b.

```
1  lemma optimal_substructure_property:
2      forall s, t, a, b, l1, l2, l_ab.
3      shortest_path s (l1 ++ (Cons a l_ab) ++ (Cons b l2)) t ->
4      shortest_path a (Cons a l_ab) b
```

Listing 8: Optimal substructure property of paths expressed in WhyML.

Consistency Implies Admissibility We expressed this, as seen in Listing 9, by an implication for all vertices `dst` and functions `f` using the the `consistent` and `admissible` predicates described before. We also ensured, through implication, that the only functions to consider are positive functions.

```
1  lemma consistent_implies_admissible:
2      forall dst: vertex, f: (vertex -> int). positive f ->
3      consistent f dst -> admissible f dst
```

Listing 9: Consistency implies admissibility expressed in WhyML.

Termination and Completeness This property was defined directly in the `astar_code` using post-conditions and assertions. The post condition is defined using the `returns` keyword as seen in Listing 10. It states the properties that must hold if `None` or `Some` are returned – there is no path if `None` is returned (line 4), or this is the shortest path is `Some` is returned (line 3). We also `assert` right before returning `None` that `forall v, l. path src l v -> mem v closed`, which ensures all nodes with a path from the source have been closed.

```
1  returns  { result ->
2      match result with
3      | Some n -> shortest_path_with_len src dst n
4      | None -> forall l. not path src l dst
5      end
6    }
```

Listing 10: Completeness post-condition for A* expressed in WhyML.

Optimal Efficiency We use an assertion before returning the shortest distance (line 10 in Listing 7) to prove this property. We assert that for all vertices v with a shortest path of weight s, if $f(v) = s + h(v)$ is strictly less than $f(n)$ of the destination node, then this node must have been closed, seen in Listing 11.

```
1  assert {
2      forall v:vertex, s:int.
3      shortest_path_with_len src v s ->
4      s + h v < d[u] + h u -> mem v closed
5    }
```

Listing 11: Optimal efficiency property expressed as a WhyML assertion.

Minimal Expansion We prove this property by adding an assertion after retrieving the next open node (after line 6 in Listing 7). We assert that the `d[u]` is the weight of the shortest path using the `shortest_path_with_len` predicate described before. Additionally, we add an invariant to both loops to ensure that for all members of `closed`, we already found the shortest path (expressed as a clause in the `inv` predicate seen in Appendix C on line 241).

Open Optimum This is expressed with an invariant on the outer loop, as seen in Listing 12. The invariant states that for any path from source for which the destination (n) is not yet closed (line 2), there must exists an open vertex u (line 4), which is on this shortest path (lines 5-7), and A* already found the shortest distance from source to u (line 8).

```
1  invariant {
2      forall n, l. shortest_path src l n -> not (mem n closed) ->
3          (exists u, l1, l2.
4              mem u open /\
5              shortest_path src l1 u /\
6              shortest_path u l2 n /\
7              l = l1 ++ l2 /\
8              d[u] = path_weight l1 u)
9    }
```

Listing 12: "Lemma 1" of the A* proof expressed in WhyML.

5 Results and Observations

In this section we give an overview of the proof generated from our implementation as well as the observations we made in the process.

5.1 Statistics on Generated Proof

To prove the properties mentioned before, we used Why3 with three SMT provers (Alt-Ergo, Z3, CVC5) and one ATP prover (E Theorem Prover). We elaborate on how to reproduce our results in Section 6. We predominately relied on applying transformations, and running the `Auto_Level_1` strategy provided by Why3, which runs the three SMT solvers with a time-limit of 5 seconds. If the aforementioned strategy failed, we ran Eprover (E Theorem Prover) before attempting further transformations.

From the initial 17 goals (which includes 16 lemmas and the `astar_code`), the transformations generated a further 181 sub-goals, totalling 198 proof goals. Of the 181 sub-goals, 128 are *leaf-goals* (VCs without any further sub-goals, which are directly offloaded to the provers). The number leaf-goals proven, minimum time, maximum time, and mean time taken to prove a goal (in seconds) for each prover can be seen listed in Table 4. As can be seen, all leaf-goals were proven well within 5 seconds, mostly using CVC5 and Z3.

Prover	# of Goals	Min	Max	Mean
Alt-Ergo 2.6.2	21	0.01 s	1.73 s	0.10 s
CVC5 1.2.1	53	0.01 s	2.66 s	0.34 s
Eprover 2.0	2	0.07 s	0.37 s	0.22 s
Z3 4.15.0	52	0.00 s	0.69 s	0.03 s
Total	128	0.00 s	2.66 s	0.17 s

Table 4: The number of leaf-goals proven, and the minimum, maximum, and mean time taken per leaf-goal by each prover. (*Leaf-goals* are VCs without any further sub-goals, which are directly offloaded to the provers.)

We used 70 transformations in the proof. The frequency of use for each type of transformation can be seen in Table 5. The most common transformation used was `assert` (used 18 times), while the least common was `induction` (each used only once). The following are short descriptions of what each transformation does, which are explained more thoroughly in the Why3 Manual [20]:

- `assert` – Adds the given premise to the context, and generates an additional sub-goal to prove said premise holds.
- `unfold` – Substitutes instances of the predicate/function with the given name with its definition. Optionally, `in` can be used to do it in a hypothesis rather than the goal.
- `destruct_rec` – Recursively destructs the top-most logic symbol and stops if a symbol of a different type is found. If necessary, generates additionally sub-goals to prove that part of the premise holds, for example, the antecedent, A, must be proven when destructing the implication, $A \rightarrow B$.
- `split_vc` – Splits the goal into small verification conditions.
- `inst_rem` – Introduces a new hypothesis by substituting the variable in the top-most `forall` by the given values in a given hypothesis. It also removes the original hypothesis after substituting.
- `exists` – Replaces the top-most term in an `exists` with the given term and generates a goal to prove this holds.
- `case` – Generates sub-goals based on a given premise split – one where it holds, and one where it does not.
- `induction_pr` – Attempts to split the current goal into a base case and inductive cases based on some premise, such as an inductive predicate.
- `introduce_premises` – Moves variables in `forall` and antecedents from implications from the goal into the context.
- `induction` – Creates goals for the base case and inductive case for a proof by induction on a given integer term.

5.2 Usability

Below we list our observation related to the usability and user-friendliness of Why3.

Transformation	# of Uses
`assert`	18
`unfold`	9
`destruct_rec`	9
`split_vc`	8
`inst_rem`	6
`exists`	6
`case`	6
`induction_pr`	4
`introduce_premises`	3
`induction`	1
Total	70

Table 5: The number of uses for each type of Why3 transformation in the proof of A*.

High Expressivness of WhyML: Due to WhyML providing various common programming constructs (such as while loops, for loops, if statements, and code blocks), translating the A* pseudocode in Algorithm 1 required little effort. Almost no changes needed to be made to write the pseudocode in WhyML, except the use of while loop instead of a for loop for the inner loop, as seen in Listing 7.

Simple Installation Process: The installation process of the Why3 platform was simple through OPAM [25] (OCaml Package Manager). Through OPAM, it was possible to install Why3, its IDE, as well as Alt-Ergo, with one command: `opam install why3 why3-ide alt-ergo`.

Why3 IDE Lacks The "Undo" Feature: In the version of Why3 used in this project (see Section 6 for list of versions), its IDE does not allow undoing changes to the source file. This was a recurring impediment during development, since changes or deletions were permanent and impossible to undo.

Limited/Out-Dated Documentation: The main sources of documentation of the Why3 platform are the Why3 Manual [20] and the Standard Library documentation [21], both of which lack information. The manual, although extensive, misses documentation on certain transformations, such as `induction_pr` and `inversion_pr`. Additionally, it gives little information on what the various WhyML language constructs actually do, with only the syntax grammar available for most. The standard library, although extensive, mostly provides direct code rather than explanation of its function, with some comments. However, not all modules have comments or explanations.

5.3 Automation

We outline below our observation on the proof automation feature of the Why3 platform.

Fast, Consistent and Reproducible Proof Generation: As seen in Table 4, all the provers find proofs in well under 5 seconds. We observed little cases where providing more time to provers was necessary. The provers either timed-out

after 30 seconds, or found a solution within the 5 seconds. Additionally, we observed that rerunning the provers never provided differing results, meaning prover output stayed consistent. Lastly, Why3's proof sessions and `replay` command allowed for easy reproducibility of the results.

ATPs are Useful Despite Out-Dated Versions: Of the two ATPs we tried, Why3 supports outdated versions of these provers – supports EProver 2.0 while latest is 3.2, and supports 0.6 while latest is 4.9. Despite this, we found two cases (as seen in Table 4) where EProver was the only prover to succeed within 20 seconds (Alt-Ergo, Z3, and CVC5 timed-out).

Manual Proving is Still a Large Component: As seen in Table 5, the proof required 72 manual transformations, with the most frequently used transformation being `assert`. Additionally, various lemmas, outside the ones proving the properties mentioned in Section 3, were provided which aided the provers in finding a proof without the need for manual transformations. For example, the `path_zero` lemma states that if a path has weight zero, it must be a nil-path.

5.4 Program Verification

This section lists out observations pertaining to program verification in WhyML.

Possibility of Principle of Explosion: We have observed in our attempts that it is possible to prove contradictions in Why3, which lead to the principle of explosion. We proved that our attempted implementation of A* returns shortest path despite having an inadmissible heuristic, and generates a proof significantly faster than previous and later attempts. This is a known flaw, and is usually caused by to providing contradictory axioms, which Why3 assumes are true [3].

Pre/Post-Loop States are Linked Through Invariants: When proving that the Open Optimum property (expressed in Section 4) is preserved after a single loop iteration, we noticed that there is no relation in the task context between the `open` set before the inner loop and after the inner loop. We relied on the invariants to reason about the new state of `open` in order to prove this property. Through experimentation, we observed that it was non-trivial to generate a proof that the `open` set before the loop must be a subset of the set after the loop, a premise which must hold due to the inner loop only adding elements to the set.

6 Considerations for Responsible Research

We sought to uphold responsible research practices in this project. As such, we dedicated this section to detailing our considerations in that regard. We provide useful information for reproducing our results (Section 6.1) and how we used AI for this project (Section 6.2).

6.1 Reproducibility

To ensure reproducibility, the code developed in this project, as well as the proof generated using Why3, is publicly available on GitHub[2]. The repository contains a `README.md`, which details its contents and useful information for reproducibility - how to run, versions of software used, and commands to replicate proofs. Additionally, the code is annotated with comments to improve clarity.

In this project we used various software, with different versions. We sought to use the most up to date versions. For the provers, we were limited by the versions supported by Why3. The following versions were used:

- Why3 1.8.0
- Alt-Ergo 2.6.2
- CVC5 1.2.1
- Eprover 2.0
- Z3 4.15.0

Why3 provides "sessions" which save the versions of provers, transformations, as well as proofs generated by the provers for a given file. This allows for easier replication of results, as one can used the commands `why3 replay astar`. More information regarding this can be found in the `README.md`. We also recommend consulting the Why3 Manual for more useful commands [20].

There are two factors that we identified which might impact reproducibility of our results. Firstly, although provers are mostly deterministic, some, like Z3, are known to use randomness for some of their heuristics. Although this did not seem to impact determinism during this project, it is, nonetheless, something to note. Secondly, provers have preset time and memory limits. The results in this study were generated on a MacBook Pro with the M1 Max and 32 GB of RAM. As can been seen in Section 5, all proofs ran far below the 20 second limit. If however, this seems to be an issue, we suggest increasing the time and memory limits available to provers as described in the Why3 Manual [20].

The results shown in Table 4 were generated using Why3's `--provers-stats` and `--session-stats` flags [20]. The results in Table 5, however, were generates using the unix command:

```
why3 session info --session-stats ./astar |
grep -oE 'transformation +[a-zA-Z_]+' |
grep -oE '[a-zA-Z_]+$' |
sort |
uniq -c |
sort -nr
```

6.2 Use of AI in This Project

ChatGPT was used in the writing of this report to help with finding appropriate LaTeX packages, such as the `listings` package for code blocks. Additionally, we used ChatGPT at the begining of the project to generate a different perspective on how to go about implementing A* in WhyML. However, it failed to provide useful information for our implementation. We used the following prompt for this:

> Can you show me an implementation of the A* algorithm in WhyML?

We also used LLMs to help in understanding Rocq [11] strategies as an introduction to the transformations provided by Why3.

[2] https://github.com/kmneumann/Why3-A-Star.git

7 Conclusion

7.1 Main Conclusions

We have successfully demonstrated and implementation A* in WhyML (Why3's native language). Using this, we were able to prove various properties which define the correctness of A*. This process led us to discover a number of capabilities and limitations of the platform as a tool for automated software verification.

The biggest advantage of Why3 that we found is it's highly expressive language. Our implementation, seen in Listing 7, followed almost exactly the pseudocode we defined in Algorithm 1, in most part due to this expressivity. Similarly, the properties we defined could be expressed with WhyML's support for logical expressions. Another strength of this tool was the speed, consistency, and reproducibility of the proofs it generates. We think its mainly due to the variety of automated provers which Why3 utilises. Lastly, we found the installation process to be quite simple, which can greatly improve more wide spread adaptation of this tool, both in education and in industry.

We did, however, encounter some limitations. Why3's Manual lacked explanations [20], particularly on the language and the transformations used during the proving process. This created a steep learning curve in the initial phase of this project. We also found that, although a lot of automation is available, proofs still involve manual proving by the user, which could make it difficult to incorporate Why3 in a developer's toolkit. Lastly, we found that it is possible to accidentally prove correctness by introducing contradictory axioms (principle of explosion). This fact impacts the validity of proofs, particularly for large scale projects.

Despite these limitations, and considering the age of the tool, Why3's features provide a powerful platform for formal verification of programs. Although writing proof directly in WhyML still requires significant effort, Why3 provides a powerful interface to interact with automated provers. Therefore, the approach taken by tools like SPARK [22], where Why3 is used as a back-end, seems the most promising use of this platform.

7.2 Future Work

A simple extension to this paper could be trying to use our implementation to run the algorithm on a specific example. The Why3 Manual mentions how one can run WhyML programs in Section 9 [20]. Additionally, it could be possible to verify different variations of A* in different contexts, to provide a better understanding of the shortcomings of our implementation.

It is also important to note that, in this paper, we analysed Why3 based only on a single case study of A*. This approach was chosen due on account of our limited experience with formal verification and time constraints. An interesting extension to this paper would be a larger, more thorough experiment to test Why3's limitations. Additionally, this case study doesn't use all available constructs provided by WhyML, which could be tested more exhaustively in a larger experiment.

A Implementation of a Mutable Map in WhyML

The following is an implementation of a mutable map in WhyML, created by Jean-Christophe Filliâtre in the verification of Dijkstra's Algorithm [9]. It is also available on GitHub: https://github.com/kmneumann/Why3-A-Star.git.

```
(** {2 Mutabe Map Mudole}

  This definition was taken from the Why3 proof of Dijkstra's algorithm.

  We use it for the distance function in the A* algorithm.

*)

module ImpmapNoDom

  use map.Map
  use map.Const

  type key

  type mutMap 'a = abstract { mutable contents: map key 'a }

  (** initialises a map that with the given value for all keys *)
  val function create (x: 'a): mutMap 'a
    ensures { result.contents = const x }

  (** retrives the value stored at the given key *)
  val function ([]) (m: mutMap 'a) (k: key): 'a
    ensures { result = m.contents[k] }

  (** logic function that assigns the given value to the given key *)
  val ghost function ([<-]) (m: mutMap 'a) (k: key) (v: 'a): mutMap 'a
    ensures { result.contents = m.contents[k <- v] }

  (** program function that assigns the given value to the given key *)
  val ([]<-) (m: mutMap 'a) (k: key) (v: 'a): unit
    writes { m }
    ensures { m = (old m)[k <- v] }

end
```

B Verification of Dijkstra's Algorithm in WhyML

The following is the verification of Dijkstra's Algorithm in WhyML, created by Jean-Christophe Filliâtre [9]. This proof was the primary inspiration for verifying A* in this paper.

```
(* Dijkstra's shortest path algorithm.

   This proof follows Cormen et al's "Algorithms".

   Author: Jean-Christophe Filliatre (CNRS) *)

module ImpmapNoDom

  use map.Map
  use map.Const

  type key

  type t 'a = abstract { mutable contents: map key 'a }

  val function create (x: 'a): t 'a
    ensures { result.contents = const x }

  val function ([]) (m: t 'a) (k: key): 'a
    ensures { result = m.contents[k] }

  val ghost function ([<-]) (m: t 'a) (k: key) (v: 'a): t 'a
    ensures { result.contents = m.contents[k <- v] }

  val ([]<-) (m: t 'a) (k: key) (v: 'a): unit
    writes { m }
    ensures { m = (old m)[k <- v] }

end

module DijkstraShortestPath

  use int.Int
  use ref.Ref

  (** The graph is introduced as a set v of vertices and a function g_succ
     returning the successors of a given vertex.
     The weight of an edge is defined independently, using function weight.
     The weight is an integer. *)

  type vertex

  clone set.SetImp with type elt = vertex
  clone ImpmapNoDom with type key = vertex

  constant v: fset vertex

  val ghost function g_succ (_x: vertex) : fset vertex
    ensures { subset result v }

  val get_succs (x: vertex): set
    ensures { result = g_succ x  }

  val function weight vertex vertex : int (* edge weight, if there is an edge *)
    ensures { result >= 0 }

  (** Data structures for the algorithm. *)

  (* The set of already visited vertices. *)

  val visited: set

  (* Map d holds the current distances from the source.
     There is no need to introduce infinite distances. *)

  val d: t int

  (* The priority queue. *)

  val q: set

  predicate min (m: vertex) (q: set) (d: t int) =
    mem m q /\
    forall x: vertex. mem x q -> d[m] <= d[x]

  val q_extract_min () : vertex writes {q}
    requires { not is_empty q }
    ensures  { min result (old q) d }
    ensures  { q = remove result (old q) }

  (* Initialisation of visited, q, and d. *)

  val init (src: vertex) : unit writes { visited, q, d }
    ensures { is_empty visited }
    ensures { q = singleton src }
    ensures { d = (old d)[src <- 0] }

```

```
88      (* Relaxation of edge u->v. *)
89
90      let relax u v
91        ensures {
92          (mem v visited /\ q = old q /\ d = old d)
93          \/
94          (mem v q /\ d[u] + weight u v >= d[v] /\ q = old q /\ d = old d)
95          \/
96          (mem v q /\ (old d)[u] + weight u v < (old d)[v] /\
97              q = old q /\ d = (old d)[v <- (old d)[u] + weight u v])
98          \/
99          (not mem v visited /\ not mem v (old q) /\
100             q = add v (old q) /\
101             d = (old d)[v <- (old d)[u] + weight u v]) }
102     = if not mem v visited then
103         let x = d[u] + weight u v in
104         if mem v q then begin
105           if x < d[v] then d[v] <- x
106         end else begin
107           add v q;
108           d[v] <- x
109         end
110
111     (* Paths and shortest paths.
112
113        path x y d =
114           there is a path from x to y of length d
115
116        shortest_path x y d =
117           there is a path from x to y of length d, and no shorter path *)
118
119     inductive path vertex vertex int =
120       | Path_nil :
121           forall x: vertex. path x x 0
122       | Path_cons:
123           forall x y z: vertex. forall d: int.
124           path x y d -> mem z (g_succ y) -> path x z (d + weight y z)
125
126     lemma Length_nonneg: forall x y: vertex. forall d: int. path x y d -> d >= 0
127
128     predicate shortest_path (x y: vertex) (d: int) =
129       path x y d /\ forall d': int. path x y d' -> d <= d'
130
131     lemma Path_inversion:
132       forall src v:vertex. forall d:int. path src v d ->
133       (v = src /\ d = 0) \/
134       (exists v':vertex. path src v' (d - weight v' v) /\ mem v (g_succ v'))
135
136     lemma Path_shortest_path:
137       forall src v: vertex. forall d: int. path src v d ->
138       exists d': int. shortest_path src v d' /\ d' <= d
139
140     (* Lemmas (requiring induction). *)
141
142     lemma Main_lemma:
143       forall src v: vertex. forall d: int.
144       path src v d -> not (shortest_path src v d) ->
145       v = src /\ d > 0
146       \/
147       exists v': vertex. exists d': int.
148         shortest_path src v' d' /\ mem v (g_succ v') /\ d' + weight v' v < d
149
150     lemma Completeness_lemma:
151       forall s: set.
152       (* if s is closed under g_succ *)
153       (forall v: vertex.
154          mem v s -> forall w: vertex. mem w (g_succ v) -> mem w s) ->
155       (* and if s contains src *)
156       forall src: vertex. mem src s ->
157       (* then any vertex reachable from s is also in s *)
158       forall dst: vertex. forall d: int.
159       path src dst d -> mem dst s
160
161     (* Definitions used in loop invariants. *)
162
163     predicate inv_src (src: vertex) (s q: set) =
164       mem src s \/ mem src q
165
166     predicate inv (src: vertex) (s q: set) (d: t int) =
167       inv_src src s q /\ d[src] = 0 /\
168       (* S and Q are contained in V *)
169       subset s v /\ subset q v /\
170       (* S and Q are disjoint *)
171       (forall v: vertex. mem v q -> mem v s -> false) /\
172       (* we already found the shortest paths for vertices in S *)
173       (forall v: vertex. mem v s -> shortest_path src v d[v]) /\
174       (* there are paths for vertices in Q *)
175       (forall v: vertex. mem v q -> path src v d[v])
176
177     predicate inv_succ (_src: vertex) (s q: set) (d: t int) =
178       (* successors of vertices in S are either in S or in Q *)
179       forall x: vertex. mem x s ->
```

```
    forall y: vertex. mem y (g_succ x) ->
    (mem y s \/ mem y q) /\ d[y] <= d[x] + weight x y

  predicate inv_succ2 (_src: vertex) (s q: set) (d: t int) (u: vertex) (su: set) =
    (* successors of vertices in S are either in S or in Q,
       unless they are successors of u still in su *)
    forall x: vertex. mem x s ->
    forall y: vertex. mem y (g_succ x) ->
    (x<>u \/ (x=u /\ not (mem y su))) ->
    (mem y s \/ mem y q) /\ d[y] <= d[x] + weight x y

lemma inside_or_exit:
    forall s, src, v, d. mem src s -> path src v d ->
      mem v s
      \/
      exists y. exists z. exists dy.
        mem y s /\ not (mem z s) /\ mem z (g_succ y) /\
        path src y dy /\ (dy + weight y z <= d)

  (* Algorithm's code. *)

  let shortest_path_code (src dst: vertex)
    requires { mem src v /\ mem dst v }
    ensures  { forall v: vertex. mem v visited ->
                 shortest_path src v d[v] }
    ensures  { forall v: vertex. not mem v visited ->
                 forall dv: int. not path src v dv }
  = init src;
    while not is_empty q do
      invariant { inv src visited q d }
      invariant { inv_succ src visited q d }
      invariant { (* vertices at distance < min(Q) are already in S *)
                  forall m: vertex. min m q d ->
                  forall x: vertex. forall dx: int. path src x dx ->
                  dx < d[m] -> mem x visited }
      variant   { cardinal v - cardinal visited }
      let u = q_extract_min () in
      assert { shortest_path src u d[u] };
      add u visited;
      let su = get_succs u in
      while not is_empty su do
        invariant { subset su (g_succ u) }
        invariant { inv src visited q d  }
        invariant { inv_succ2 src visited q d u su }
        variant   { cardinal su }
        let v = choose_and_remove su in
        relax u v;
        assert { d[v] <= d[u] + weight u v }
      done
    done

end
```

C Verification of The A* Algorithm in WhyML

The following is the verification of A* in WhyML as described in Section 4. This code is also available on GitHub: https://github.com/kmneumann/Why3-A-Star.git.

```
(** {2 A* Algorithm Module}

  This module implements the specification of A* listed in the paper.

*)

module AStar

  use int.Int
  use list.List
  use list.Append
  use option.Option

  (** {6 Graph & Paths Definitions}

    Below is the code defininng the graph and paths, as described in the paper.
  *)

  type vertex

  clone set.SetImp with type elt = vertex

  (** set of all vertices in the graph *)
  constant graph: fset vertex

  (** logic (ghost) definition of the `successors` function *)
  val ghost function successors (x: vertex): fset vertex
    ensures { subset result graph }
    ensures { not mem x result }

  (** program definition of the `successors` function *)
  val successors_impl (x: vertex): set
    ensures { result = successors x }

  (** `edge` predicate used to link the above graph definition to `graph.IntPathWeight` *)
  predicate edge (a b: vertex) = mem b (successors a)

  (** imports the definition of `path` and `path_weight` from the standard library *)
  clone graph.IntPathWeight with
    type vertex = vertex,
    predicate edge = edge

  (** ensures edge weights can only be positive *)
  axiom positive_weight: forall a, b. weight a b > 0

  (** definition of `shortest_path` *)
  predicate shortest_path (a: vertex) (l: list vertex) (b: vertex) =
    path a l b /\
    (forall l'. path a l' b -> path_weight l b <= path_weight l' b)

  (** alternate definition of `path` via its weight rather than a list *)
  predicate path_with_len (a b: vertex) (d: int) =
    exists l. path a l b /\ (path_weight l b = d)

  (** alternate definition of `shortest_path` via its weight rather than a list *)
  predicate shortest_path_with_len (a b: vertex) (d: int) =
    exists l. shortest_path a l b /\ (path_weight l b = d)

  (** predicate used to ensure that the edges in `graph` never lead outside of it *)
  predicate closed_under_succ (v: fset vertex) =
    forall n. mem n v -> forall m. edge n m -> mem m v


  (** {6 Lemmas on Graph & Paths}

    Below are lemmas about the properties of the graph and paths on it.
  *)

  (** paths always have positive weight *)
  lemma path_nonneg:
    forall x, y, l. path x l y -> path_weight l y >= 0

  (** shortest paths always have positive weight *)
  lemma shortest_path_nonneg:
    forall x, y, l. shortest_path x l y -> path_weight l y >= 0

  (** a shortest path consiting of 2 parts, implies both parts are also shortest paths *)
  lemma shortest_path_decomposition:
    forall x y z: vertex, l1 l2: list vertex.
    shortest_path x (l1 ++ Cons y l2) z -> shortest_path x l1 y /\ shortest_path y (Cons y l2) z

  (** if a list is not a shortest path, it either isn't a path or there is a path shorter than it *)
  lemma shortest_path_negation:
    forall a, b, l. not (shortest_path a l b) ->
    not (path a l b) \/ (exists l'. path a l' b /\ path_weight l' b < path_weight l b)

  (** the weight of a path is equal to the sum of the weights of the sub-paths on it *)
```

```
lemma path_weight_sub_path:
  forall x y z: vertex, l1 l2 l3: list vertex.
  path_weight (l1 ++ (Cons x l2) ++ (Cons y l3)) z =
  path_weight l1 x + path_weight (Cons x l2) y + path_weight (Cons y l3) z

(** a sub-path on a shortest path must also be a shortest path *)
lemma optimal_substructure_property:
  forall s, t, a, b, l1, l2, l_ab.
  shortest_path s (l1 ++ (Cons a l_ab) ++ (Cons b l2)) t ->
  shortest_path a (Cons a l_ab) b

(** paths can be joined with an edge to make another valid path *)
lemma sub_path:
  forall x, a, b, z, l1, l2.
      path x l1 a -> path b l2 z -> edge a b -> path x (l1 ++ (Cons a l2)) z

(** a path is either the nil-path, or it is two paths joined by an edge *)
lemma sub_path_inversion:
  forall x z: vertex, l: list vertex. path x l z ->
     (x = z /\ l = Nil)
  \/ (exists a, b, l1, l2.
      path x l1 a /\ path b l2 z /\ edge a b /\ l = l1 ++ (Cons a l2))

(** a path from some `src` in the set `s` to some `v` outside of the set `s` must contain an edge which exists this set `s` *)
lemma inside_or_exit_path:
  forall s, src, v, l. mem src s -> path src l v -> not (mem v s) ->
    (exists y, z, l1, l2.
      mem y s /\ not (mem z s) /\ edge y z /\
      path src l1 y /\ path z l2 v /\ l = l1 ++ Cons y l2)

(** a shortest path from some `src` in the set `s` to some `v` outside of the set `s` must contain an edge which exists this set `s` *)
lemma inside_or_exit_shortest_path:
  forall s, src, v, l. mem src s -> shortest_path src l v -> not (mem v s) ->
    (exists y, z, l1, l2.
      mem y s /\ not (mem z s) /\ edge y z /\
      shortest_path src l1 y /\ shortest_path z l2 v /\ l = l1 ++ Cons y l2)

(** if the exists a path from `a` to `b`, there must exist a shortest path as well *)
lemma path_imples_exists_shortest_path:
  forall a, b, l. path a l b -> (exists l'. path_weight l' b <= path_weight l b /\ shortest_path a l' b)

(** a path of weight zero must necessarily be the nil-path *)
lemma path_zero:
  forall a, b, l. path a l b -> path_weight l b = 0 -> l = Nil /\ a = b

(** a path of weight zero must necessarily be the nil-path *)
lemma main_lemma:
  forall src v: vertex. forall l: list vertex.
  path src l v -> not (shortest_path src l v) ->
  v = src /\ l <> Nil
  \/
  (exists v': vertex. exists l': list vertex. shortest_path v' l' v /\ edge src v' /\ path_weight (Cons src l') v < path_weight l v)

(** this lemma was defined in the proof of Dijkstra's algorithm *)
lemma completeness_lemma:
  forall s: set.
  (* if s is closed under `successors` *)
  closed_under_succ s ->
  (* and if s contains src *)
  forall src: vertex. mem src s ->
  (* then any vertex reachable from s is also in s *)
  forall dst, l. path src l dst -> mem dst s


(** {6 Heuristic Function Definitions}

  Here we define predicates about the Heuristic function.
*)

(** the heuristic function must never return a negative value *)
predicate positive (f: vertex -> int) =
  (forall n. f n >= 0)

(** the heuristic function must be admissible *)
predicate admissible (f: vertex -> int) (dst: vertex) =
  forall a, l. path a l dst -> f a <= path_weight l dst

(** the heuristic function must be consistent *)
predicate consistent (f: vertex -> int) (dst: vertex) =
  f dst = 0 /\
  (forall a b:vertex.
    edge a b ->
    f a <= weight a b + f b )

(** alternate definition of consistentcy *)
predicate path_consistent (f: vertex -> int) (dst: vertex) =
  f dst = 0 /\
  (forall a b:vertex, l: list vertex.
    path a l b ->
    f a <= path_weight l b + f b )


```

```
(** {6 Lemmas on Heuristic Function Properties}

  Here we define lemmas on the propertiesof a heuristic function (defined above).
*)

(** the two alternate definitions of consistency are equivelent *)
lemma consistent_is_path_consistent:
  forall dst: vertex, f: (vertex -> int). positive f -> consistent f dst <-> path_consistent f dst

(** a consistent heuristic is also and admissible heuristic *)
lemma consistent_implies_admissible:
  forall dst: vertex, f: (vertex -> int). positive f -> consistent f dst -> admissible f dst


(** {6 Distnace Function and the CLOSED and OPEN Set Definitions}

  Here we define functions useful for dealing with the CLOSED and OPEN sets.
*)

(** imports the mutable map function we use for the distance function *)
clone ImpmapNoDom with type key = vertex

(** predicate defining a vertex with the smallest f(n) value in a set *)
predicate min (m: vertex) (q: set) (d: mutMap int) (h: vertex -> int) =
  mem m q /\
  (forall x: vertex. mem x q -> d[m] + h m <= d[x] + h x)

(** program function which removes and returns the vertex in the given set with the smalles f(n) value *)
val get_min (open: set) (d: mutMap int) (h: vertex -> int) : vertex
  writes   { open }
  requires { not is_empty open }
  ensures  { min result (old open) d h }
  ensures  { open = remove result (old open) }


(** {6 Predicates Used in Loop Invariants}

Below are predicates used to define the loop invariants of the A* algorithm.

They are added so that the actual code is easier to follow.

Note that for these predicates: `s` is the CLOSED set and `q` is the OPEN set
*)

(** the source is either closed or open *)
predicate inv_src (src: vertex) (s q: set) =
  mem src s \/ mem src q

(** these invariants hold for both the outer and inner loops *)
predicate inv (src dst: vertex) (s q: set) (d: mutMap int) =
  (* The source is either closed or open: *)
  inv_src src s q /\
  (* The distance from src to src is zero: *)
  d[src] = 0 /\
  (* We have not yet closed the destination vertex: *)
  not mem dst s /\
  (* CLOSED and OPEN are subsets of the graph: *)
  subset s graph /\ subset q graph /\
  (* CLOSED and OPEN are disjoint: *)
  disjoint s q /\
  (* We already found the shortest paths for vertices in CLOSED: *)
  (forall v: vertex. mem v s -> shortest_path_with_len src v d[v]) /\
  (* There are paths for vertices in OPEN: *)
  (forall n. mem n q \/ mem n s -> path_with_len src n d[n]) /\
  (* For every open vertex `n`, there is a vertex in closed from which `n` succeeds: *)
  (forall n. mem n q -> (exists u. mem n (successors u) /\ mem u s /\ d[u] + weight u n = d[n]) \/ n = src) /\
  (* No node is both open and closed at the same time: *)
  (forall n. not (mem n q /\ mem n s))

(** an invariant that holds for the outer loop *)
predicate inv_succ (_src: vertex) (s q: set) (d: mutMap int) =
  (* successors of vertices in CLOSED are either in CLOSED or in OPEN: *)
  forall x: vertex. mem x s ->
  forall y: vertex. mem y (successors x) ->
  (mem y s \/ mem y q) /\ d[y] <= d[x] + weight x y

(** an invariant that holds for the inner loop *)
predicate inv_succ2 (_src: vertex) (s q: set) (d: mutMap int) (u: vertex) (su: set) =
  (* successors of vertices in CLOSED are either in CLOSED or in OPEN, unless they are successors of `u` still in `su` *)
  forall x: vertex. mem x s ->
  forall y: vertex. mem y (successors x) ->
  (x<>u \/ (x=u /\ not (mem y su))) ->
  (mem y s \/ mem y q) /\ d[y] <= d[x] + weight x y


(** {6 The A* Algorithm}

Below is the impletation of the A* algorithm as described in the paper.
*)

let astar_code (src dst: vertex) (h: vertex -> int): option int
  requires { consistent h dst /\ positive h }
```

```
    requires { closed_under_succ graph }
    requires { mem src graph /\ mem dst graph }
    returns  { result ->
      match result with
      | Some n -> shortest_path_with_len src dst n
      | None -> forall l. not path src l dst
      end
     }
  = let closed: set = empty () in
    let open: set = singleton src in
    let d: mutMap int = create 0 in
    while not is_empty open do
      invariant { inv src dst closed open d }
      invariant { inv_succ src closed open d }
      invariant {
          forall n, l. shortest_path src l n -> not (mem n closed) ->
            (exists u, l1, l2.
              mem u open /\
              shortest_path src l1 u /\
              shortest_path u l2 n /\
              l = l1 ++ l2 /\
              d[u] = path_weight l1 u)
        }
      variant   { cardinal graph - cardinal closed }
      let u = get_min open d h in
      assert { shortest_path_with_len src u d[u] };
      if eq u dst then begin
       assert { forall v:vertex, s:int. shortest_path_with_len src v s -> s + h v < d[u] + h u -> mem v closed };
       return Some d[u]
      end;
      add u closed;
      let su = successors_impl u in
      while not is_empty su do
        invariant { subset su (successors u) }
        invariant { inv src dst closed open d  }
        invariant { inv_succ2 src closed open d u su }
        variant   { cardinal su }
        let y = choose_and_remove su in
        let x = d[u] + weight u y in
        if not mem y closed then begin
          if not (mem y open) || x < d[y] then begin
              add y open;
              d[y] <- x
          end
        end;
        assert { d[y] <= d[u] + weight u y }
      done;
      assert { forall m. edge u m -> mem m closed \/ mem m open };
    done;
    assert { forall v, l. path src l v -> mem v closed };
    return None


end
```

References

[1] D. Edsger W., "Notes on structured programming (EWD 249)," in *Structured Programming*, 1972. [Online]. Available: https://www.cs.utexas.edu/~EWD/transcriptions/EWD02xx/EWD249/EWD249.html (visited on 01/05/2025).

[2] S. Edwards, L. Lavagno, E. A. Lee and A. Sangiovanni-Vincentelli, "Design of embedded systems: Formal models, validation, and synthesis," in *Readings in Hardware/Software Co-Design*, G. De Micheli, R. Ernst and W. Wolf, Eds., San Francisco: Morgan Kaufmann, 1st Jan. 2002, pp. 86–107, ISBN: 18759661. DOI: 10.1016/B978-155860702-6/50009-0. [Online]. Available: https://www.sciencedirect.com/science/article/pii/B9781558607026500090.

[3] M. Schoolderman, "Verification of goroutines using why3," Master's Thesis, Radboud University, Nijmegen, Jul. 2016, 90 pp. [Online]. Available: https://www.cs.ru.nl/masters-theses/2016/M_Schoolderman__Verification_of_Goroutines_using_Why3.pdf.

[4] *Why3*. [Online]. Available: https://www.why3.org/ (visited on 01/05/2025).

[5] F. Bobot, J.-C. Filliâtre, C. Marché and A. Paskevich, "Why3: Shepherd your herd of provers," presented at the Boogie 2011: First International Workshop on Intermediate Verification Languages, 2011, p. 53. DOI: 10/document. [Online]. Available: https://inria.hal.science/hal-00790310 (visited on 26/04/2025).

[6] J.-C. Filliâtre and A. Paskevich, "Why3 — where programs meet provers," in *Programming Languages and Systems*, ISSN: 1611-3349, Springer, Berlin, Heidelberg, 2013, pp. 125–128, ISBN: 978-3-642-37036-6. DOI: 10.1007/978-3-642-37036-6_8. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-642-37036-6_8 (visited on 26/04/2025).

[7] P. E. Hart, N. J. Nilsson and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, Jul. 1968, ISSN: 2168-2887. DOI: 10.1109/TSSC.1968.300136. [Online]. Available: https://ieeexplore.ieee.org/document/4082128 (visited on 01/05/2025).

[8] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, no. 1, pp. 269–271, 1st Dec. 1959, ISSN: 0945-3245. DOI: 10.1007/BF01386390. [Online]. Available: https://doi.org/10.1007/BF01386390.

[9] "Gallery of verified programs." (2012-2025), [Online]. Available: https://toccata.gitlabpages.inria.fr/toccata/gallery/index.en.html (visited on 26/04/2025).

[10] C. A. R. Hoare, "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, no. 10, pp. 576–580, Oct. 1969, ISSN: 0001-0782. DOI: 10.1145/363235.363259. [Online]. Available: https://doi.org/10.1145/363235.363259.

[11] *Rocq*. [Online]. Available: https://rocq-prover.org (visited on 02/06/2025).

[12] *Isabelle*. [Online]. Available: https://isabelle.in.tum.de/ (visited on 02/06/2025).

[13] *Vampire*. [Online]. Available: https://vprover.github.io/ (visited on 02/06/2025).

[14] *The E Theorem Prover*. [Online]. Available: https://wwwlehre.dhbw-stuttgart.de/~sschulz/E/E.html (visited on 02/06/2025).

[15] *SPASS*. [Online]. Available: https://www.mpi-inf.mpg.de/departments/automation-of-logic/software/spass-workbench/classic-spass-theorem-prover (visited on 02/06/2025).

[16] *Alt-Ergo*. [Online]. Available: https://alt-ergo.ocamlpro.com (visited on 02/06/2025).

[17] *CVC5*. [Online]. Available: https://cvc5.github.io/ (visited on 02/06/2025).

[18] *Z3 Prover*. [Online]. Available: https://github.com/Z3Prover/z3 (visited on 02/06/2025).

[19] J.-C. Filliâtre, "One Logic To Use Them All," in *CADE 24 - the 24th International Conference on Automated Deduction*, M. P. Bonacina, Ed., Lake Placid, NY, United States: Springer, Jun. 2013. [Online]. Available: https://inria.hal.science/hal-00809651.

[20] "The Why3 Platform." (2025), [Online]. Available: https://why3.gitlabpages.inria.fr/why3/index.html (visited on 10/06/2025).

[21] "Why3 standard library," Why3 Standard Library. (2025), [Online]. Available: https://www.why3.org/stdlib/ (visited on 23/05/2025).

[22] *SPARK*. [Online]. Available: https://www.adacore.com/about-spark (visited on 10/06/2025).

[23] J. M. Cohen and P. Johnson-Freyd, "A formalization of core why3 in coq," *Proc. ACM Program. Lang.*, vol. 8, no. POPL, Jan. 2024. DOI: 10.1145/3632902. [Online]. Available: https://doi.org/10.1145/3632902.

[24] P. Amit. "Amit's A* Pages." (1997), [Online]. Available: https://theory.stanford.edu/~amitp/GameProgramming/ (visited on 22/06/2025).

[25] *Opam*. [Online]. Available: https://opam.ocaml.org/ (visited on 10/06/2025).