A* with h^{max} Definitely Finds Optimal Plans – Formally Verifying a Planner Based on Heuristic Search

Gregor Behnke, Simone Kilian, Malvin Gattinger

University of Amsterdam, Institute for Logic Language and Computation, The Netherlands
g.behnke@uva.nl, simone.kilian@student.uva.nl, b.r.m.gattinger@uva.nl

Abstract

Research in automated planning relies on both proofs of theoretical properties and the practical implementation of methods to solve planning problems. Both strands can suffer from the same problem: inadvertant errors either due to oversight, human faults, or the sheer complexity of the problem that caused not all situations to be correctly considered. As a consequence published theorems might be wrong or well-known and often used planners may contain bugs.

One remedy is rigorous formalisation. Both the theory of planning and implementations of planning algorithms should be formalised so that a theorem prover can verify their correctness. While previous formalisations of planning focussed, e.g., on plan validation and SAT-based planning, we provide a Lean 4 formalisation of heuristic search algorithms in general, as well as some well known planning heuristics including h^{max}, h^+, and abstraction heuristics.

Introduction

Over the course of multiple decades of research in automated planning, researchers have created a multitude of planning systems – starting with the original STRIPS (Fikes and Nilsson 1971), continuing with Graphplan (Blum and Furst 1997), HSP (Bonet and Geffner 2001), FF (Hoffmann and Nebel 2001), to modern planners like FastDownward (Helmert 2006), Madagascar (Rintanen, Heljanko, and Niemelä 2006), or SymBA (Torralba et al. 2014). All these planners were carefully implemented and the papers introducing them typically contain human-readable proofs of correctness, completeness, and if applicable optimality of these planners, e.g., via proving admissibility of the employed heuristics. By construction, these planners produce a plan that can be checked by an external tool for correctness w.r.t. the input planning problems. The traditional validator VAL (Howey and Long 2003) and a later developed formally verified verifier (Abdulaziz and Lammich 2018) can only check whether the plan produced by the planner is *some executable and goal-achieving plan*, but not whether it is optimal. Similarly, unsolvability of planning problems cannot be witnessed by a single plan. For both optimality and unsolvability more complex certificates and corresponding checkers have been developed (Eriksson, Röger, and Helmert 2017, 2018).

In all these cases, the only formal statement that can be made about the planner is that it produced a valid or optimal plan or proved unsolvability for that particular given planning problem for which it produced a plan or certificate respectively. The natural instinct is to generalise many successfully validated plans (resp. certificates) together with a written proof of soundness, completeness, and optimality to the statement that the planner itself is sound. As we all know, this can be a devilish fallacy: almost no planner is without an error in its implementation. In some rare cases even the theoretical results underlying them may turn out to be false under closer inspection of proofs written in natural language. Notably, the A* search algorithm was shown to be incomplete for infinite graphs even with the perfect heuristic (Yousefi et al. 2025), running against the common conception that it is sound, complete, and optimal in general.

One way to remedy this issue, is to provide a fully formalised theory of planning together with a fully formalised implementation of a planner. Such a formalisation naturally has to contain proofs of the statements relevant for the formalisation and for the correctness of the planner. A (interactive) theorem prover can then be utilised to check the formalisation and its proofs, yielding a high degree of trust in both the proven theorems and the planner's implementation. Within planning, this endeavour has been undertaken for creating a formally verified planners based on a translation to boolean satisfiability (Abdulaziz and Kurz 2023) and for fully observable probabilistic planning (Schäffeler and Abdulaziz 2023, 2025).

As an additional benefit, formalising results previously obtained and proven by human-understandable proofs typically requires a more precise formalisation of these results. These formalisations may reveal imprecisions or additional requirements that have been obscured by the vagueness inherent in proofs written in human language – and simultaneously increase the trust in these results when newer research is build on top of them.

In this paper, we provide the first substantial step towards creating a formally verified planner that uses heuristics. We provide our formalisation in the interactive theorem prover Lean 4. To achieve this goal, multiple components are needed:

1. A formalised theory of STRIPS, FDR, or FTS planning
2. A formalisation of PDDL and its semantics
3. A formally verified parser and grounder

4. A formally verified search engine
5. Formally verified heuristics

In this paper, we tackle components 1, 4, and 5 – while we leave 2 and 3 for future work. Prior work by Abdulaziz and Kurz (2023) already formalised PDDL, and provided a formally verified parser and grounder for it. These formalisation were done in Isabelle/HOL and are thus not directly compatible with the Lean 4 formalisation that we provide here. Recently, Nicodemus (2026) provided a formalisation of STRIPS planning in Lean 4. It comes with a parser for the STRIPS format used by Eriksson, Röger, and Helmert's (2018) planner producing unsolvability certificates. This format is somewhat similar to FastDownward's internal representation of FDR problems. We have used Nicodemus's formalisation of STRIPS planning as the basis for our work.

In the remainder of this paper, we first introduce the basic notions for STRIPS planning and theorem proving in Lean 4. We then describe our formalisation of search algorithms for weighted graphs. In it, we formally prove the soundness, completeness, and optimality (with an admissible heuristic) of the A* algorithm (Hart, Nilsson, and Raphael 1968), which to our knowledge is the first fully formalised proof of it's correctness. We then turn to connecting the formalisation of STRIPS planning to our implementation of A* which yields a sound, complete, and optimal planner, given an admissible heuristic. We then describe our formalisation of some of the known heuristics of planning, notably the h^+ and h^{max} heuristics (Bonet and Geffner 2001), abstraction heuristics in general and Pattern Database (PDB) heuristics (Culberson and Schaeffer 1998; Edelkamp 2001) in particular as well as the notion of (non-negative) cost partitioning (Katz and Domshlak 2010; Yang et al. 2008).

Some of the proofs in our formalisation were generated using AI tools, in particular ARISTOTLE (Achim et al. 2025) and we describe our use of it at the end of our paper.

We split our formalisation into two parts: one only containing search algorithms on finite graphs in general (Behnke, Kilian, and Gattinger 2026b) and one containing the planning related concepts (Behnke, Kilian, and Gattinger 2026a). Up-to-date versions of these formalisations will be made available at https://github.com/galvusdamor/lean4-search-algorithms and https://github.com/galvusdamor/lean4-planner, respectively.

Preliminaries: Planning

We consider grounded STRIPS planning (Fikes and Nilsson 1971). Let V be a set of propositional state variables. A state s is a subset of V. An action a is a triple $(pre, add, del) \in (2^V)^3$. We write $pre(a)$ to denote the pre in that triple (resp. $add(a)$ and $del(a)$). The action a is applicable in the state s if $pre(a) \subseteq s$. Applying a in s is possible if a is applicable in s and yields the state $a[s] = (s \setminus del(a)) \cup add(a)$[1]. Applicability and application of sequences of actions π are defined inductively: If $\pi = (a)$, then π is applicable in s if a is and $\pi[s] = a[s]$. If $\pi = a \circ \pi'$, that is it starts with a and continues with π', then π is applicable in s if a is applicable in s and π' is applicable in $a[s]$. Further, if π is applicable, then $(a \circ \pi')[s] = \pi'[a[s]]$.

A planning problem $\Pi = (V, A, I, G)$ comprises a set of propositional variables V, a set of actions A, an initial state $I \subseteq V$, and a goal description $G \subseteq V$. Any state g with $G \subseteq g$ is a goal state. A plan π for Π is a sequence of actions that is applicable in I and for which $G \subseteq \pi[I]$.

Lean 4

Lean is a proof assistant originally started in 2013. The current Lean 4 was released in 2021 (Moura and Ullrich 2021) and has gained substantial traction since then. Lean's semantics is based on dependent type theory and the calculus of inductive constructions (Coquand and Huet 1988), similar to Rocq (formerly Coq). This stands in contrast to Isabelle/HOL which has been used for formalisations of planning in the past (e.g. (Abdulaziz and Kurz 2023)) and is based on the logic for computable functions. This difference leads to Lean having a smaller trusted core than other languages – as it has less code that cannot be verified using Lean itself and whose correctness needs to be trusted.

Lean applies a uniform view on functions, lemmas, and theorems: all of them have a type – for functions this is the type of the function (e.g. $\mathbb{N} \mapsto \mathbb{N}$, written as `Nat -> Nat`), while for lemmas and theorems this is their statement by virtue of the Curry-Howard correspondence nowadays also called propositions as types (Howard 1980). Lean thus allows use to define inductive data-structures (e.g., for plans and paths in graphs), but also the statements of theorems.

Functions, lemmas, and theorems are then defined or proved, respectively, by providing a term of the respective type. For functions this means to provide its implementation while for lemmas and theorems this means to provide a proof. The sole objective of Leans's trusted core is type checking: to determine whether the provided term is of the promised type. Lean 4 itself provides a more human readable way of defining proof-terms (called *tactic mode*), while limited proof automation exists, e.g., in the form of Aesop's proof search (Limperg and From 2023) and the more recent `grind` tactic. Recently Harmonic made an LLM-based proof search agent – ARISTOTLE – publicly available (Achim et al. 2025).

A formalisation in Lean 4 means to (1) define the types involved in the statements to be proven, (2) formalise the functions and theorems about them that are to be proven, and (3) provide proof-terms for all required theorems. To trust such a formalised proof, it is sufficient to ascertain whether the statement of the defined functions and theorems correctly reflects what is intended to be proven, while checking the proof itself is the task to be performed by Lean.

Lean 4 itself provides only the most basic structures and theorems, e.g., `Prop` (the type of propositions), `Nat` (natural numbers), `List`, and `Array`, but not even `Set`. The Lean community has curated additional formalisations which are often used as the common basis for further formalisations – the MATHLIB (The mathlib Community 2020). It e.g. defines the type `Set E` as `E -> Prop`.

Lean 4's community has recently started a separate formalisation of computer science – CSLIB (Barrett et al.

[1] I.e. adding takes precedence

```
structure Digraph (V : Type*) where -- defined in mathlib
  Adj : V -> V -> Prop

structure WeightedDiGraph (V : Type) (E : Type) [FinEnum V] extends Digraph V where
  Payload : (u : V) -> (v : V) -> (Adj u v) -> E
  instDecAdj : DecidableRel Adj

def NatGraph (V : Type) [FinEnum V] : Type := WeightedDiGraph V Nat
```

Figure 1: Basic definitions of `Digraph`, `WeightedDiGraph`, and `NatGraph`.[2]

2026). Work on it is still in its infancy and thus does not provide sufficient theory to be used in our project, yet. CSLIB does however contain a formalisation of labelled transition systems, which might be useful for our project in the future.

Formalising Search

Search algorithms used in planning are at their core algorithms on graphs. In particular, the most commonly used search algorithm for cost-optimal search-based planning, A*, can be understood to be executed on a graph whose vertices are the states of the given planning problem Π and whose edges are induced by Π's actions. We aim to separate the general algorithmic of search – which is independent of planning – and the particulars of search for planning. Hence we first developed a formalisation of search algorithms on graphs in Lean 4. Graph algorithms and in particular search algorithms have been the subject of formalisations in the past. Notably, formalised and provably correct implementations of Dijkstra's algorithm (Nordhoff and Lammich 2012) and Maximum Flow (Edmonds-Karp and Push-Relabel; (Lammich and Sefidgar 2019)), among other graph algorithms, exist. As of writing this paper, we are not aware of any formalisation of the A* algorithm (Hart, Nilsson, and Raphael 1968) – which is one of our contributions.

Formalising Graphs

MATHLIB already contains two formalisations of graphs. The first, `Graph`, implements undirected graphs, while the second, `Digraph`, implements directed graphs. Unfortunately, the types of `Graph` and `Digraph` are not suitably connected in MATHLIB, i.e., results cannot be transferred easily from one to the other. Further, most of graph theory is formalised for `Graph` and not for `Digraph` at the moment. As planning is mostly concerned with directed graphs, we based our formalisation on `Digraph` at the cost of having to translate already existing formalisations of graph concepts from `Graph` to `Digraph`. A `Digraph` (see Figure 1) is defined over a type of vertices V by specifying its adjacency relation: given two vertices u and v `Adj` returns the proposition of whether u and v are adjacent.

Neither `Graph` nor `Digraph` assign any weight or label to their edges. We therefore introduced the new type `WeightedDiGraph`, which a `Digraph` whose edges are labelled by terms of some type E. For graphs used in planning, we will most often choose $E = \mathbb{N}_0$, but allowing any type leaves generality for future work (e.g. if $E = \mathbb{Z}$ is necessary for general cost partitioning (Pommerening et al. 2015)). To access the label, we require a proof of the existence of the edge `Adj u v` to be provided. In general, Lean 4 allows to consider digraphs whose vertices are from an arbitrary type – but for executable algorithms we need to ensure that it is at least possible to (1) enumerate all vertices of the graph and that (2) the adjacency relation of the graph is decidable. For formalising search algorithms we thus require the vertex-type V to be `FinEnum` (finitely enumerable) and `Adj` to be `DecidableRel` (a decidable relation). Our formalisation of `WeightedDiGraph` is given in Figure 1. We further define the `NatGraph` whose edges are labelled with natural numbers – their edge cost.

For our formalisation objective, we reason mostly about paths in graphs. While they have been defined for `Graph` in MATHLIB, no definition for `Digraph` exists in MATHLIB. As MATHLIB does for `Graph`, we distinguish between walks and paths (Figure 2). A `Walk u v` is an inductively defined sequence of edges that leads from vertex u to vertex v. The type `Walk u v` already *requires* a proof that the edges traversed by the walk actually exist. A `Path u v` is then a walk in which no vertex occurs more than once – i.e. whose list of visited vertices (the support) does not contain duplicates (`List.Nodup`). Based on these definitions, we can formalise and prove simple lemmas about walks, paths, and their composition. We further introduced the notion of the cost of an edge for `NatGraph`s as well as the notion of the cost of a walk (and path) as the sum of the costs of all edges on the walk. This allows us to define what a cheapest or optimal path p from a vertex u to a vertex v is: one for which all other paths from u to v have a equal or higher cost (Figure 2).

Formalising Search Algorithms

Our formalisation is not limited to the A* algorithm, but also formalises Depth-First (DFS) and Breadth-First search (BFS), and Dijkstra's algorithm (Dijkstra 1959). Each of them takes as their input a (possibly weighted) graph, a starting vertex, and a goal vertex. To allow for re-using implementations and proofs between all four search algorithms, we developed a common search-algorithm framework that can capture all of them. At its core, each algorithm is viewed as a fix-point algorithm iterating over a current *search state*. A *search state* in this context is the full configuration that the search algorithm is currently in – which vertices have been visited, which vertices are still to be processed, and the cur-

[2]This and all future excerpts of Lean 4 code were simplified to accommodate their inclusion in this paper.

```
variable (G : WeightedDiGraph V E) -- global variable G
inductive Walk : V -> V -> Type
  | nil {u : V} : Walk u u -- the empty walk
  | cons {f w t : V} (h : (G.Adj f w)) (p : Walk w t) : Walk f t

abbrev Path (u v : V) := { p : G.Walk u v // List.Nodup p.support }

variable (F : NatGraph V)
def edgeCost {u v : V} (h : F.Adj u v) : Nat := F.Payload u v h
def cost {u v : V} : (F.Walk u v) -> Nat
  | .cons adj rest => (F.edgeCost adj) + cost rest
  | .nil => 0
def cost {u v : V} (p : F.Path u v) : Nat := cost p.val

def is_cheapest {u v : V} (p : F.Path u v) : Prop :=
  ∀ p' : G.Path u v, p.cost ≤ p'.cost
```

Figure 2: Definitions of walks, paths, and cost.

```
variable (G : WeightedDiGraph V E)
structure search_state (V E : Type) [FinEnum V] (D : Type) [FValueComp D]  where
    visited : Finset V
    distance : V -> D
    mother : visited -> V
    fringe : List V
```

Figure 3: Definitions of a search state.

rently known "distance"[3] of each vertex from the start vertex. Our framework uses the common type `search_state` for the search states of all search algorithms (Figure 3). The type of the "distance" is left open as a type variable `D`, which is required to have a well-founded less-than operation (i.e. no infinite descent is possible) via `FValueComp`. For BFS and DFS, `D` is simply `Nat`, while Dijkstra's algorithm and A* use `Nat` $\times$ `Nat`. These pairs encode both the cost of the shortest path to a vertex and the number of edges in that path. This additional information was helpful for tie-breaking in case of edges with cost zero and helps to simplify proofs.

In a `search_state` we have a finite set of already `visited` vertices, the `distance`, the list `fringe` of still to be processed vertices, and `mother` providing for every visited vertex its predecessor on a path from the start.

A search algorithm is then defined by a function `expand`, which takes a current `search_state` and performs one expansion step. The search algorithm iterates `expand` until either the `fringe` is empty or the first element of `fringe` is a goal state. To operate, `expand` receives the current `search_state`, the front element of the fringe, and the remainder of the fringe and returns the next `search_state` (see `search_expand` in Figure 4). We provide an abridged formalisation of this process and the resulting function `search_recurse` in Figure 4. Note that this means that `expand` has to rearrange the `fringe` so that the vertex to be expanded next is at the start. Our formalisation comprises definitions of `expand` for DFS, BFS, Dijkstra's algorithm, and A*. The latter additionally requires a heuristic – which is simply a function from V to $\mathbb{N}_0$.

[3]For DFS this is the search depth of each vertex.

The main function from our search-algorithm framework is `search_exe`. It takes a start state and goal states and an `expand` function and returns `Option (F.Path start goal)`, i.e. either `none` if no path was found or some path from the start to a goal. It does so by calling `search_recurse` and extracting the necessary information from its return values.

Implementing A* in Lean 4 The implementation of A*'s `expand` function comprises approximately 40 lines of Lean 4 code.Both A* and Dijkstra's algorithm share the same `expand` function – where Dijkstra's algorithm passes the zero-heuristic. This `expand` function is called `hsearch_expand`. Overall, the function performs the following steps assuming that H is the current front element of the fringe and F is the remainder of the fringe:

1. Determine the set NV of neighbours of H that are not yet visited or for which the distance in the current `search_expand` is larger than the distance via H
2. Mark all vertices in NV as visited
3. Update the distances for all vertices in NV
4. Let any vertex in NV now have H as its mother
5. Let the new fringe be $NV \cup F$, sorted by `distance` plus heuristic.

Using the `expand` function defined this way, the overall A* algorithm can then be defined as given in Figure 4.

Proving Correctness of Search Algorithms

Most of the work in formalising search algorithms lies in proving their correctness. This task can be split into three sub-tasks: proving soundness, i.e., that the returned path is indeed a path, proving completeness, i.e., that if a path exists

```
abbrev search_expand (F : WeightedDiGraph V E) (D : Type) [FValueComp D] : Type :=
    search_state V E D -> V -> List V -> search_state V E D

def search_step (F : WeightedDiGraph V E) (goal : V)
    (prior_state : search_state V E D) (expand : search_expand V E D)
    : search_state V E D × (Option Bool) :=
  match prior_state.stack with
    | [] => (prior_state, some false) -- goal not found
    | (s :: xs) => if s = goal then (prior_state, some true)
                               else (expand prior_state s xs, none)

def search_recurse (F : WeightedDiGraph V E) (goal : V)
    (prior_state : search_state V E D) (expand : search_expand V E D)
    : search_state V E D × Bool :=
  let next := search_step F goal priorState expand
  let nextState := next.fst
  let result : Option Bool := next.snd
  if result_none : result = none
  then search_recurse F goal nextState expand
  else <nextState,result.get (by apply Option.isSome_iff_ne_none.mpr;exact result_none)>

def astar (F : NatGraph V) (heur : V -> Nat) (start : V) (goal : V):
    Option (F.Path start goal) :=
  let start_state := WeightedDiGraph.search_state_initial start <0,0>
  search_exe F start goal start_state (hsearch_expand heur)
```

Figure 4: Definitions generic search algorithm. Modified for readability.

one is actually returned, and proving optimality, which only applies to BFS (w.r.t. path length), Dijkstra's, and A*.

In formalisations of algorithms (see e.g. (Abdulaziz 2025) for Dijkstra in Isabelle/HOL), correctness proofs often use invariants to obtain their main result. We also first define the invariants kept by the `expand` function, prove that they are indeed invariants under application of `expand`, and then use them to extract the main result.

The invariants necessary to prove soundness and completeness are the same for all four search algorithms we implemented. Our proof-framework provides one centralised definition of these six invariants and a generalised proof that if these invariants hold for the `expand` function, the resulting search algorithm is sound and complete. This abstraction allows for a substantial simplification of the proofs: the main results only need to be proven once, while the algorithm-specific proofs only have to show simple properties w.r.t. one execution of the `expand` function. Specifically these invariants are (for the formalisation see Figure 5):

1. the start vertex is visited
2. all vertexes in the fringe are visited
3. all visited vertexes have a mother has is visited
4. for all visited vertexes v except start, the mother of v is adjacent to v
5. for all visited vertexes v except start, the mother's distance is strictly smaller than the distance of v
6. all visited vertexes v are either in the fringe or all their neighbours are also visited

Together the invariants ensure that whenever a vertex v is visited there is a path from the start vertex to v. If the search finds a goal vertex, that vertex is in the fringe, thus it is visited (invariant Nr. 2), and thus a path from the start vertex to that goal exists. Conversely, if the search terminates without finding a goal vertex, invariants Nr. 1 and 6 ensure that all vertices reachable from the start vertex are visited. Since this does not include a goal vertex (otherwise a path would have been found), no path from the start vertex to a goal can exist.

What remains is to prove that all six invariants are kept by the `expand` functions of BFS, DFS, Dijkstra's, and A* – which is in generally a simple and straight-forward proof.

Proving Optimality

Our main new formalised result is that A* is optimal given an admissible heuristic. A heuristic in general is a function `V -> Nat`, i.e., a function that returns for every vertex of the graph a natural number.[4] It is admissible (formalised in Figure 6), if for every vertex v and every path p from that vertex to the goal of the search, the cost of p is at least has high as the value of the heuristic for v.

Our main result w.r.t. A* search is given in Figure 6 – it states that given a `NatGraph`, a start and a goal vertex, a heuristic and a proof of its admissibility, as well as the fact that the invocation of A* returned some path, that that path is a cheapest, i.e., optimal path. Using the definition of a path being the cheapest path from Figure 2, we can then cleanly define the main result `astar_is_optimal` (Figure 6).

For `astar_is_optimal`, we formalised the proof originally given by Hart, Nilsson, and Raphael (1968). Helpfully their proof already stated an invariant (their Lemma 1) that we could use for a formalised proof.

[4]For simplicity, we restricted both the edge costs of graphs and the values that can be returned by heuristics to natural numbers.

```
1 start ∈ s.visited
2 ∀ x : V, x ∈ s.fringe -> x ∈ s.visited
3 ∀ x : s.visited, s.mother x ∈ s.visited
4 ∀ x : s.visited, x ≠ start -> G.Adj (s.mother x) x
5 ∀ x : s.visited, x ≠ start -> s.distance (s.mother x) < s.distance x
6 ∀ x : s.visited, x ∈ s.fringe ∨ ∀ y : V, (G.Adj x y) -> y ∈ s.visited
```

Figure 5: Basic invariants for all search algorithms.

```
1 abbrev admissible (F : NatGraph V) (heur : V -> Nat) (goal : V) :=
2   ∀ v : V, ∀ p : F.Path v goal, p.cost ≥ (heur v)
3 theorem astar_is_optimal (F : NatGraph V) (start : V) (goal : V) (heur : V -> Nat)
4     (is_admissible : g.admissible heur goal)
5     (returned_path : Option.isSome (astar F heur start goal)):
6     ((astar F heur start goal).get returned_path).is_cheapest := <proof omitted>
```

Figure 6: Theorem stating the optimality of A*.

Lemma 1 (Hart, Nilsson, and Raphael (1968))**.** *For any non-closed vertex v and for any optimal path p from the start vertex to v, there exists an open vertex v' on p so that the current estimate of the distance between the start vertex and v' is their true distance.*

In our formalisation, the current estimate of this distance is `distance v'`, which leads to the invariant in Figure 7, lines 4–6. From this invariant, Hart, Nilsson, and Raphael then conclude in a corollary that for an admissible heuristic and if the search has not terminated yet, any optimal path from the start vertex to the goal has one vertex currently in the fringe and its f-value is less or equal to the cost of that optimal path. The f-value here refers to the sum of the current distance estimate and the heuristic, i.e., `distance v' + heur v'`. In the proof of the corollary, Hart, Nilsson, and Raphael tacitly assume that for every optimal path there always exists some vertex v on the path that is not yet closed. While this seems intuitive for a human reader, in a formalisation we need to prove this statement too and thus add it as a further invariant. One can now conclude (as done by Hart, Nilsson, and Raphael in Theorem 1, case 3) that if A$\star$ terminates with finding a path to the goal vertex, no cheaper path p' can exist: If it would exist, take the cheapest among them. The corollary requires one vertex of that path p' to be in the fringe with an f-value lower than the cost of an optimal path. This contradicts the fact that the fringe is sorted by f-value and the vertex with the lowest value on the stack was the goal.

As part of the proof of Hart, Nilsson, and Raphael's main invariant we further need to show that A*'s `expand` ensures that the stored `distance` values satisfy the Bellman equation. Lastly, we require two additional technical invariants that state that the fringe is sorted according to $f(v) = h(v) + g(v)$ and that the distance of the start vertex is always $(0, 0)$ (i.e. cost zero with zero edges). The full formalisation of all five invariants can be found in Figure 7. Two of them (starting with `hsearch`) are also used when proving optimality of Dijkstra's algorithm.

Formalisations of DFS, BFS, and Dijkstra's

In addition to the formalisation of A* search, we also provided formalisation of DFS, BFS, and Dijkstra's algorithm and proofs of their soundness, correctness, and optimality for BFS and Dijkstra's – mostly as a reference.

Formalising Planning

Based on our formalisation of search on graphs, we developed a formalisation STRIPS planning via search and some well-known heuristics. Since planning problems typically have more than one goal state (as every state satisfying the goal condition is considered a goal), we created a formalised version of A* search that finds the optimal path to any of a given list of goals – via a reduction to the case with only one goal. In the reduction, one adds a new "super"-goal vertex that can be reached from every goal vertex with an edge of cost zero. Soundness, completeness, and optimality follow easily from the same properties for A* with only one goal.

A prior thesis (Nicodemus 2026) formalised STRIPS planning in Lean 4 with the computational efficiency of the representation in mind.[5] However, this formalisation is limited as it was created only to reason about the unsolvability of planning problems and thus misses tools to reason about the cost of plans and any method of finding plans. We adopted this definition of STRIPS planning and provided additional functionality and proofs where necessary. It defines, e.g., types for STRIPS problems with n propositional variables (`STRIPS n`), states (`State' n`), actions (`Action n`), and plans for a STRIPS problem $prob$ from the state $init$ (`Plan prob init`).

To connect STRIPS planning to the formally specified search algorithms, we defined the graph structure that a planner actually searches on: the problem's transition system. Given a planning problem $\Pi = (V, A, I, G, C)$, its unlabelled transition system $\Theta = (V_\Theta, E_\Theta, C_\Theta)$ is a graph whose vertex set V_Θ is 2^V, its edge set E_Θ comprises all pairs (s, s') for which an action $a \in A$ exists that is applicable in s and for which $s' = a[s]$ and whose cost function C_Θ returns for any edge the minimum cost of $c(a)$ of such

[5] Available at https://github.com/AmosNico/validator

```
variable (F : NatGraph V)
abbrev vertex_open (s : search_state F (Nat × Nat)) (v : V) := v ∈ s.fringe
abbrev vertex_closed (s : search_state F (Nat × Nat)) (v : V) := v ∈ s.visited ∧ v ∉ s.
    fringe
abbrev astar_invar (start : V) (s : search_state F (Nat × Nat)) :=
  ∀ v : V, ¬(vertex_closed s v) -> ∀ p : F.Path start v, p.is_cheapest ->
    ∃ v' ∈ p.support, (vertex_open s v') ∧ F.cost_is start v' (s.distance v').1
abbrev astar_path_invar (start : V) (goal : V) (s : search_state F (Nat × Nat)) :=
  ∀ p : F.Path start goal, ∃ v' ∈ p.support, (vertex_open s v')
def add_heur (v : V) (p : Nat × Nat) (heur : V -> Nat) : Nat × Nat := <p.1 + heur v, p.2>
abbrev astar_stack_sorted (s : search_state F (Nat × Nat)) :=
  List.Pairwise (fun u v =>
    (add_heur u (s.distance u) heur = add_heur ∨ (s.distance v) heur) ∨ (add_heur u (s.
    distance u) heur < add_heur ∨ (s.distance v) heur)) s.fringe
abbrev hsearch_invar_on_stack_or_all_neighbours_max_order (s : search_state F (Nat × Nat))
    := ∀ x : s.visited, x ∉ s.fringe ->
    ∀ y : V, (adj : F.Adj x y) -> (s.distance y).1 ≤ (s.distance x).1 + F.edgeCost adj
abbrev hsearch_invar_start_dist_zero_zero (start : V) (s : search_state F (Nat × Nat)) :=
    s.distance start = (0,0)
```

Figure 7: Definitions of invariants for A*.

```
abbrev heur_admissible {n : Nat} (prob : STRIPS n) (heur : State' n -> Nat):=
  ∀ v : State' n, ∀ plan : Plan prob (convertState v), plan.path.cost ≥ (heur v)
```

Figure 8: Definitions of admissibility for graphs in general and planning in particular.

actions. Once we defined this translation in Lean 4, we can create a planner by invoking the A* algorithm for multiple goals in Θ with the initial state I and any g with $g \subseteq G$ to be a goal. This only returns a path in Θ, i.e., without actions, but not a plan for Π. We added a translation step that given a path from I to a goal state in Θ extracts the intended plan.

Soundness and completeness of this planner follow immediately from the corresponding theorems for A*.

To prove optimality, the underlying A* search algorithm requires a proof of admissibility of the provided heuristic with respect to the graph that the A* search is executed on. From the point-of-view of planning, it would be more convenient to prove admissibility w.r.t. the planning problem only and not as required w.r.t. the transition system induced by the planning problem. We provide a simpler planning-only definition of admissibility `heur_admissible` (see Figure 8) together with a proof that `heur_admissible` implies that the heuristic is admissible w.r.t. the transition system, i.e., that it satisfies `admissible` from Figure 6. To prove that our planner is optimal given an admissible heuristic, we needed lastly to show that given a path Θ, our plan extraction selects the cheapest possible action at any step and that no cheaper plan can exist if the given path was optimal in Θ.

Formalising Heuristics

In addition to the pure base mechanics and properties of the search-based planner, we also formalised several admissible heuristics. So far, we defined and proved admissibility of the following heuristics:

- The perfect heuristic h^*
- The perfect delete relaxation heuristic h^+
- The zero heuristic h^0.
- The $h^{max} = h^1$ heuristic (Bonet and Geffner 2001)
- Abstraction Heuristics in general
- PDB Heuristics by proving that PDBs are abstractions

We provide proofs that goal-aware and admissible heuristics are consistent (allowing for potentially easier proofs of admissibility) and that any non-negative cost-partitioning of admissible heuristics is admissible.

Formal Proofs Generated by LLMs

As noted before, proving a theorem in Lean requires to provide a term of the type that corresponds to the theorem's statement. Traditionally, this means to either provide the proof-term itself manually or to use the tactic mode of Lean to obtain the proof-term. Correctness of the proof-term is then determined by Lean's kernel, independently from how the proof term was written or generated. Hence we may also obtain the proof-term from some other source and still obtain a valid proof, as long as it is accepted by Lean's kernel. Importantly, we do *not have to trust* that other source.

In particular, this separation of proof generation and proof checking enables the use of large language models (LLMs) to generate proof-terms. In our work, we have used Harmonic's LLM ARISTOTLE (Achim et al. 2025) to assist in creating some of the proofs. ARISTOTLE is an LLM specifically trained to generate proofs in Lean 4.

A* search We used ARISTOTLE to prove the main invariant of A* search as stated in Lemma 1 of Hart, Nilsson, and Raphael (1968) – once all other necessary invariants were already proven. ARISTOTLE was provided with a PDF copy of the paper by Hart, Nilsson, and Raphael, the implementation of A* as well as proofs of soundness, completeness,

and optimality for BFS, Dijkstra, and DFS (there without optimality), as well as the the instruction to prove lemma 1, in which it succeeded. All other elements of the described search algorithms and their proofs were written by hand.

Simple Planning Heuristics In order to test the current abilities of ARISTOTLE, we tasked it to prove the admissibility of various heuristics. In all cases, the implementation of the heuristic (or heuristic technique) was created manually by a human. ARISTOTLE proved without further input the admissibility of h^* and h^+, abstraction heuristics in general, the fact that syntactic projections onto variables (i.e. PDBs) are abstractions and thus yield admissible abstraction heuristics, as well as the statement that any non-negative cost-partitioning over admissible heuristics is admissible.

The h^{max} Heuristic We also tasked ARISTOTLE to prove the admissibility of the h^{max} heuristic, given as a dynamic programming implementation (Bonet and Geffner 2001). ARISTOTLE failed to finalize a proof after seven invocations, each exhausting the compute budget (whose value is unknown at the time of writing). At this point, its generated output did not change substantially any more and it seemed to have entered a cycle in which no progress can be made.

As a second attempt, we opted to provide a high-level skeleton of the argument used to prove the admissibility of h^1max. We split the proof into two parts. Firstly, we provided a characterisation of h^{max} as a regression heuristic. This is precisely the original definition of h^{max} by Bonet and Geffner (2001) – i.e. their definition of $g_s^{max}(C)$. To precisely capture this definition in our formalisation, we introduced the operation of replacing the goal of a problem Π with a given state K. We write $\Pi[K]$ to denote this new problem. Translated to our notation, we defined the following equations that are satisfied by h^{max}_Π for any state:

$$h^{max}_\Pi(s) = \begin{cases} 0 & \text{, if } G \subseteq s \\ \max\limits_{g \in G} h^{max}_{\Pi[\{g\}]}(s) & \text{, if } |G| > 1 \\ \arg\min\limits_{a \in A, g \in add(a)} c(a) + h^{max}_\Pi(pre(a)) & \text{, if } |G| = \{g\} \end{cases}$$

Note that in the second case (the max case), we transition from one planning problem to another with a different goal. We then stated formally, that any heuristic that satisfies these regression-based equations is admissible.

Secondly, we stated that the dynamic programming-based implementation of h^{max} satisfies the regression-based definition and is thus admissible. With this structural help, ARISTOTLE was able to complete the proof of admissibility of h^{max} using eight invocations of all but the last ran until the compute budget was exhausted.

Empirical Test

The planner we formalised is an actual runnable planner, meaning that the Lean code can be compiled and executed on actual STRIPS planning problems. Instructions for how to run this planner can be found in the respective repository. We did not implement PDDL-level parsing or grounding and thus rely on an external tool for this step. For this purpose, we used a modified version (Eriksson, Röger, and Helmert 2018) of Fast Downward (Helmert 2006), which is able to generate a grounded STRIPS output-format that can be read in Lean 4 using Nicodemus's (2026) implementation. The provable correctness of our planner applies only w.r.t. the input provided by the external parser and grounder.

		h^{max}		h^0	
instance	$\|V\|$	time	mem	time	mem
gripper p01	24	Stack Overflow			
miconic s1-0	6	0.2	153	0.1	124
miconic s2-0	12	11.0	155	17.5	133
miconic s3-0	18	Stack Overflow			
movie p01	14	1443.4	160	1097.4	136
psr-small p01	13	28.6	156	17.9	134
psr-small p02	13	Stack Overflow			
zenotravel p01	18	Stack Overflow			

Table 1: Runtime (in seconds) and memory usage (in MB) for selected planning problems from the IPC benchmarks. $|V|$ denotes the number of propositional states variables that each benchmark problem has.

We ran a proof-of-concept test with small planning problems from the International Planning Competition (IPC).[6] Our formalised implementation of A* was not created with the intent of being a particularly efficient implementation, but rather to provide a provably correct version of the algorithm. Consequently, the implementation is set up in a way that makes proving its correctness as easy as possible. For example, when expanding a vertex v, all search algorithms have to iterate over all states u and only then check whether v and u are adjacent. This inefficiency is mainly caused by MATHLIB's definition of adjacency, which is a test given two vertices and not, e.g., a list of neighbours given a vertex v. Furthermore, our implementation of A* does not maintain a priority queue of vertices in the fringe, but fully sorts the entire fringe after every expansion. Overall, the implementation has a runtime complexity of $\mathcal{O}(|V|^2)$ (for V being the vertices). We aim to improve the algorithm's practical efficiency in future implementations.

If used to solve a planning problem, our implementation requires to enumerate all states of the given STRIPS problem Π, i.e., $2^{|V|}$ many states. This turns the search into an $\times(2^{|V|})$ algorithm for planning. Due to its strictly exponential runtime, we can only solve the smallest planning problems. In practical terms, the limit is somewhere between 10 and 20 propositional state variables. We selected a few planning problems from the IPC that are sufficiently small. We ran our planner with both the h^{max} heuristic and the constant zero heuristic (h^0), see Table 1. In most cases the planner does not run into memory exhaustion, but into a stack overflow – as the search algorithm is a recursive algorithm.

Formalisation

At the time of submission of this paper, the formalisation for search algorithms and planning concepts comprised

[6] https://github.com/AI-Planning/classical-domains/.

10.377 lines of code. Of these, 2.319 are specific to planning, 935 are helper lemmas on core concepts (e.g. lists or well-founded recursion), 1.218 formalise weighted directed graphs, and 5.087 lines formalised search algorithms. Of the current formalisation 2.025 lines of code were written by ARISTOTLE.

Conclusion and Future Work

We presented the first fully formalised implementation of A* and proved its soundness, completeness, and optimality. Using A* we provided a sound, complete, and optimal STRIPS planner and several admissible heuristics for it, including h^{max}. The empirical performance of the planner is still limited, but could be improved in future work. Further, we plan to expand our formalisation to capture more heuristics, e.g., to include LM-cut (Helmert and Domshlak 2009), h^m in general (Haslum and Geffner 2000), or merge-and-shrink (Helmert et al. 2014).

References

Abdulaziz, M. 2025. *Graph Algorithms*, 273–298. New York, NY, USA: Association for Computing Machinery, 1 edition. ISBN 9798400731570.

Abdulaziz, M.; and Kurz, F. 2023. Formally Verified SAT-Based AI Planning. In *Proc. AAAI 2023*, 15073–15081.

Abdulaziz, M.; and Lammich, P. 2018. A formally verified validator for classical planning problems and solutions. In *Proc. ICTAI 2018*, 474–479. IEEE.

Achim, T.; Best, A.; Bietti, A.; Der, K.; Fédérico, M.; Gukov, S.; Halpern-Leistner, D.; Henningsgard, K.; Kudryashov, Y.; Meiburg, A.; Michelsen, M.; Patterson, R.; Rodriguez, E.; Scharff, L.; Shanker, V.; Sicca, V.; Sowrirajan, H.; Swope, A.; Tamas, M.; Tenev, V.; Thomm, J.; Williams, H.; and Wu, L. 2025. Aristotle: IMO-level Automated Theorem Proving. arXiv:2510.01346.

Barrett, C.; Chaudhuri, S.; Montesi, F.; Grundy, J.; Kohli, P.; de Moura, L.; Rademaker, A.; and Yingchareonthawornchai, S. 2026. CSLib: The Lean Computer Science Library. arXiv:2602.04846.

Behnke, G.; Kilian, S.; and Gattinger, M. 2026a. galvusdamor/lean4-planner: v4.27.0-hsdip2026. Zenodo, DOI: 10.5281/zenodo.20686788.

Behnke, G.; Kilian, S.; and Gattinger, M. 2026b. galvusdamor/lean4-search-algorithms: v4.27.0-hsdip2026. Zenodo, DOI: 10.5281/zenodo.20510135.

Blum, A.; and Furst, M. L. 1997. Fast Planning Through Planning Graph Analysis. 90(1–2): 281–300.

Bonet, B.; and Geffner, H. 2001. Planning as Heuristic Search. 129(1): 5–33.

Coquand, T.; and Huet, G. 1988. The calculus of constructions. *Information and Computation*, 76(2): 95–120.

Culberson, J. C.; and Schaeffer, J. 1998. Pattern Databases. 14(3): 318–334.

Dijkstra, E. W. 1959. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1: 269–271.

Edelkamp, S. 2001. Planning with Pattern Databases. In *Proc. ECP 2001*, 84–90.

Eriksson, S.; Röger, G.; and Helmert, M. 2017. Unsolvability Certificates for Classical Planning. In *Proc. ICAPS 2017*, 88–97.

Eriksson, S.; Röger, G.; and Helmert, M. 2018. A Proof System for Unsolvable Planning Tasks. In *Proc. ICAPS 2018*, 65–73.

Fikes, R. E.; and Nilsson, N. J. 1971. STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving. 2: 189–208.

Hart, P. E.; Nilsson, N. J.; and Raphael, B. 1968. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2): 100–107.

Haslum, P.; and Geffner, H. 2000. Admissible Heuristics for Optimal Planning. In *Proc. AIPS 2000*, 140–149.

Helmert, M. 2006. The Fast Downward Planning System. 26: 191–246.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In *Proc. ICAPS 2009*, 162–169.

Helmert, M.; Haslum, P.; Hoffmann, J.; and Nissim, R. 2014. Merge-and-Shrink Abstraction: A Method for Generating Lower Bounds in Factored State Spaces. *Journal of the ACM*, 61(3): 16:1–63.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. 14: 253–302.

Howard, W. A. 1980. The Formulae-as-Types Notion of Construction. In Seldin, J. P.; and Hindley, J. R., eds., *To H. B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, 479–490. Academic Press.

Howey, R.; and Long, D. 2003. VAL's Progress: The Automatic Validation Tool for PDDL2.1 used in the International Planning Competition. In *ICAPS 2003 Workshop on the Competition*.

Katz, M.; and Domshlak, C. 2010. Optimal admissible composition of abstraction heuristics. 174(12–13): 767–798.

Lammich, P.; and Sefidgar, S. R. 2019. Formalizing network flow algorithms: A refinement approach in isabelle/hol. *Journal of Automated Reasoning*, 62(2): 261–280.

Limperg, J.; and From, A. H. 2023. Aesop: White-Box Best-First Proof Search for Lean. ISBN 9798400700262.

Moura, L. d.; and Ullrich, S. 2021. The Lean 4 Theorem Prover and Programming Language. In Platzer, A.; and Sutcliffe, G., eds., *Automated Deduction – CADE 28*.

Nicodemus, A. 2026. *Formalizing Unsolvability Certificates for Automated Planning in Lean 4*. Master's thesis, Institute for Logic, Language and Computation, University of Amsterdam, Amsterdam, The Netherlands.

Nordhoff, B.; and Lammich, P. 2012. Dijkstra's Shortest Path Algorithm. *Archive of Formal Proofs*. https://isa-afp.org/entries/Dijkstra_Shortest_Path.html, Formal proof development.

Pommerening, F.; Helmert, M.; Röger, G.; and Seipp, J. 2015. From Non-Negative to General Operator Cost Partitioning. In *Proc. AAAI 2015*, 3335–3341.

Rintanen, J.; Heljanko, K.; and Niemelä, I. 2006. Planning as satisfiability: parallel plans and algorithms for plan search. 170(12–13): 1031–1080.

Schäffeler, M.; and Abdulaziz, M. 2023. Formally Verified Solution Methods for Markov Decision Processes. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, 15073–15081.

Schäffeler, M.; and Abdulaziz, M. 2025. Formally Verified Approximate Policy Iteration. 26659–26667.

The mathlib Community. 2020. The Lean Mathematical Library. In *Proc. SIGPLAN' CPP 2020*, CPP 2020. New Orleans, LA, USA: ACM.

Torralba, Á.; Alcázar, V.; Borrajo, D.; Kissmann, P.; and Edelkamp, S. 2014. SymBA*: A Symbolic Bidirectional A* Planner. In *IPC-8 Planner Abstracts*, 105–109.

Yang, F.; Culberson, J.; Holte, R.; Zahavi, U.; and Felner, A. 2008. A General Theory of Additive State Space Abstractions. 32: 631–662.

Yousefi, M.; Schmautz, M.; Haslum, P.; and Bercher, P. 2025. How Good is Perfect? On the Incompleteness of A* for Total-Order HTN Planning. In *Proceedings of the 35th International Conference on Automated Planning and Scheduling (ICAPS 2025)*, 112–120. AAAI Press.