Formally Verified Approximate Policy Iteration

Maximilian Schäffeler[1], Mohammad Abdulaziz[2]

[1]Technische Universität München, Germany
[2]King's College London, United Kingdom
maximilian.schaeffeler@tum.de, mohammad.abdulaziz@kcl.ac.uk

Abstract

We present a methodology based on interactive theorem proving that facilitates the development of verified implementations of algorithms for solving factored Markov Decision Processes. As a case study, we formally verify an algorithm for approximate policy iteration in the proof assistant Isabelle/HOL. We show how the verified algorithm can be refined to an executable, verified implementation. Our evaluation on benchmark problems shows that it is practical. As part of the development, we build verified software to certify linear programming solutions. We discuss the verification process and the modifications we made to the algorithm during formalization.

Code — https://github.com/schaeffm/fmdp_isabelle

Introduction

Markov Decision Processes (MDPs) are models of probabilistic systems, with applications in AI, model checking, and operations research. In AI, for instance, given a description of the world in terms of states and actions that can change those states in a randomised fashion, one seeks a *policy* that determines the actions chosen in every state, with the aim of accruing maximum *reward*. There is a large number of methods to solve MDPs, most notably, value and policy iteration, which compute policies with optimality guarantees.

In many applications in AI or autonomous systems (Lahijanian et al. 2010; Junges et al. 2018), obtaining an optimal policy is safety-critical, with the goal of e.g. minimizing the number of accidents. One important aspect here is the assurance that the output of the MDP solving system is correct. Such assurance is currently attained to some degree by testing and other software engineering methods. However, the best guarantee can be achieved by mathematically proving the MDP solver and the underlying algorithm correct. A successful way of mathematically proving correctness properties of (i.e. formally verifying) pieces of software is using Interactive Theorem Provers (ITPs), which are formal mathematical systems that one can use to devise machine-checked proofs. Indeed, ITPs have been used to prove correctness properties of compilers (Leroy 2009), operating systems kernels (Klein et al. 2009), model checkers (Esparza et al. 2013), planning systems (Abdulaziz and Lammich 2018; Abdulaziz and Koller 2022; Abdulaziz and Kurz 2023), and, most related to the topic of this work, algorithms to solve MDPs (Schäffeler and Abdulaziz 2023). A challenge with using ITPs to prove algorithms correct, nonetheless, is that they require intense human intervention. Thus for an ITP to be successfully employed in a serious verification effort, novel ideas in the design of the software to be verified as well as the underlying proof have to be made.

In this paper, we consider formally verifying algorithms for solving *factored* MDPs. A challenge to using MDPs to model realistic systems is their, in many cases, enormous size. For such systems, MDPs are succinctly represented as factored MDPs. The system's state is characterised as an assignment to a set of state variables and actions are represented in a compact way by exploiting the structure present in the system. Such representations are common in AI (Guestrin et al. 2003; Sanner 2010; Younes and Littman 2004) and in model checking (Hinton et al. 2006; Dehnert et al. 2017). Although ITPs have been used prove the correctness of multiple types of software and algorithms, including algorithms on MDPs, algorithms on factored MDPs are particularly challenging. The root of this difficulty is that the succinctness of the representation comes at a cost. Naively finding a solution for a factored MDP could entail the construction of structures exponentially bigger than the factored MDP. This necessitates using advanced data structures, heuristics and computational techniques, to avoid that full exponential blow up.

Our main contribution is that we develop a methodology based on using the Isabelle *locale* system, to structure the MDP solving algorithm into parts amenable to verification. To enable this methodology, we build a formal mathematical library allowing the specification of algorithms for solving factored MDPs and their properties, like algorithms for planning under uncertainty and probabilistic model checking. We also develop a number of reusable building blocks to be used in other algorithms, e.g. a certificate checker for linear programming solutions. Potential targets for formalization include probabilistic model checking, planning and reinforcement learning algorithms (Hartmanns et al. 2023; Keller and Eyerich 2012). We describe the methodology in terms of the verification of Guestrin et al.'s approximate pol-

icy iteration algorithm in the Isabelle theorem prover.

This algorithm computes approximate policies, i.e. suboptimal policies with guarantees on their optimality, for one type of factored MDPs. The algorithm we consider combines scoped functions and decision lists, which are data structures that exploit the factored representation, linear programming, in addition to probabilistic reasoning and dynamic programming. The combination of this wide range of mathematical/algorithmic concepts and techniques is what makes this algorithm particularly hard from a verification perspective. To get an idea of the scale, we advise the reader to look at Fig. **??**, which shows the hierarchy of concepts and definitions which we had to develop (aka *formalize*) within Isabelle/HOL to be able to state the algorithm and prove its correctness statement. This is, of course, in addition to the notions of analysis, probabilities, and MDPs, which already exist in Isabelle/HOL. Furthermore, to be able to prove the algorithm correct, we had to design an architecture of the implementation that makes verification feasible. Our architecture mixes verification and certification: we verify the entire algorithm, and build a verified certificate checker for linear programming solutions that delivers formal guarantees. In addition to proving the algorithm correct, we obtain a formally verified implementation, that we experimentally show to be practical. Our work, as far as we are aware, is the first work on formally verifying algorithms for factored MDPs.

Background

We introduce the interactive theorem prover Isabelle/HOL, our formal development of factored MDPs relate it to existing formalizations in Isabelle/HOL.

Isabelle/HOL

An ITP is a program that implements a formal mathematical system, in which definitions and theorem statements can be expressed, and proofs are constructed from a set of axioms. To prove a fact in an ITP, the user only provides the high-level steps while the ITP fills in the details at the level of axioms. Specifically, our developments use the interactive theorem prover Isabelle/HOL (Nipkow, Paulson, and Wenzel 2002) based on Higher-Order Logic, a combination of functional programming with logic. Isabelle is highly trustworthy, as the basic inference rules are implemented in a small, isolated kernel. Outside the kernel, several tools implement proof tactics, data types, recursive functions, etc..

Our presentation of definitions and theorems here deviates slightly from the formalization in Isabelle/HOL. Specifically, we use subscript notation for list indexing and some function applications. We use parentheses for function application. For a list xs, $len(xs)$ returns the length, while $xs_{:i}$ returns the first i elements of xs. List concatenation is written as $xs{\cdot}ys$, $x :: xs$ inserts x at the front of the list xs, $map(f, xs)$ applies f to every element of xs. Finally, $\mathbb{N}_{<i}$ is short for the first i natural numbers (including 0).

Factored MDPs

Factored MDPs are compactly represented MDPs that exploit regularities in large MDPs, which can often lead to an exponential reduction in the size of the model. Common formats to store factored MDPs include JANI (Budde et al. 2017), the PRISM language (Hinton et al. 2006), and RDDL (Sanner 2010). We implement the factoring described in (Guestrin et al. 2003). In Isabelle/HOL, we define factored MDPs using *locales* (Ballarin 2014). A locale introduces a mathematical context with constants and assumptions, in which we develop our formalization. Locales can be instantiated with concrete constants and a proof that discharges the assumptions of the locale, yielding all the theorems proved within the locale. For example, we instantiate the MDP locale from Schäffeler and Abdulaziz with the factored systems introduced in this section, and are therefore able to reuse important definitions and theorems. Moreover, reducing factored MDPs to a tested and reviewed library enhances confidence in our definitions.

State Space A state of a factored MDP is an assignment of values to its n state variables. Each state variable $i \in \mathbb{N}_{<n}$ has a finite, nonempty domain X_i. In Isabelle/HOL, we implement an MDP state x as a map, i.e. a function with an explicit domain $dom(x)$. A *partial state* is a map where only a subset of the state variables are assigned, but all entries are valid, i.e. $dom(x) \subseteq \mathbb{N}_{<n}$ and $x_i \in X_i$ for $i \in dom(x)$. The set X of *states* of the MDP consists of all partial states x where $dom(x) = \mathbb{N}_{<n}$. The domain of a state x can be restricted to a set of variables Y, denoted as $x|_Y$. One partial state x is called *consistent* with another partial state t (written $x \sqsubseteq t$), if and only if $x|_{dom(t)} = t$.

Example As a running example, we use a model of a computer network with ring topology from Guestrin et al.'s original paper (see Fig. 1). In the ring, each machine is either working or broken, and states change stochastically. Each machine C_i's state of operation is characterized by a variable i, s.t. all domains $X_i = \{\mathrm{W}, \mathrm{B}\}$. In a ring with three machines, a partial state is $s \coloneqq [1 \mapsto \mathrm{W}, 3 \mapsto \mathrm{B}]$, $dom(s) = \{1, 3\}$. It holds that $s \sqsubseteq [1 \mapsto \mathrm{W}]$, but $s \not\sqsubseteq [2 \mapsto \mathrm{B}]$.

Scoped Functions For many MDPs, the transition behavior and the rewards can be computed from a combination of functions that individually only depend on a small subset of all state variables. Such *scoped functions* take a partial state as input and determine the output inspecting only variables within their scope. The algorithm we formalize expresses policy iteration using scoped functions, which avoids enumerating the full state space. Scoped functions pose a challenge for formalization, as scopes could be represented implicitly or explicitly: we can either prove that a given function has a restricted scope, or the function can store its scope explicitly as data. In Isabelle/HOL we do both: decoupling scopes from the function definition is more flexible, as one may e.g. derive multiple scopes for a single function. However, since it is in general infeasible to compute the precise scope of a function, we use explicit scopes in the executable version of the algorithm. An important operation on scoped functions is the instantiation with a partial state t. The operation $inst_t$ applied to a scoped function returns a new scoped function with reduced scope, where all input dimensions that t provides are fixed to the values of t.

Transitions In our setting, a factored MDP comes with a finite set of actions A, and a default action $d \in A$. For each action $a \in A$, transition probabilities $\mathcal{P}_i^a : X \to \mathbb{P}(X_i)$ with $scope(\mathcal{P}_i^a) \subseteq \mathbb{N}_{<n}$ determine the evolution of variable i. Here, $\mathbb{P}(X)$ denotes the set of probability distributions over a finite set X. The set $effects_a$ defines the state variables where the behavior of a differs from the default action (i.e. variables i s.t. $\mathcal{P}_i^a \neq \mathcal{P}_i^d$). The combined transition probabilities between two states x and x' are defined as $\mathcal{P}^a(x, x') \coloneqq \prod_{i<n} \mathcal{P}_i^a(x, x'_i)$.

Example In the ring topology domain, for each machine there is an action to restart it, in which case it is guaranteed to work in the next step. On the other hand, the default action d is to do nothing. In general, the probability of a machine working in the next step depends on its own state and the state of the predecessor, e.g. $scope(\mathcal{P}_2^d) = \{1, 2\}$. The exact conditional probability distribution for the MDP's evolution under the default action is shown in Fig. 1. Using an explicit representation to model this factored action, we would need 8 transitions, each consisting of a distribution over 8 possible successor states. This is in contrast to three tables in the factored case. A common way to model the transition behavior is using dependency graphs as shown in Fig. 1.

Rewards The actions $a \in A$ define scoped reward functions $R_i^a : X \to \mathbb{R}$ for $i < r_a$. The reward for selecting a is a sum of those reward functions: $R^a(x) \coloneqq \sum_{i<r_a} R_i^a(x)$. We assume that the first r_d reward functions are the same for all actions. Now, given a policy $\pi : X \to A$ we are interested in the discounted expected total reward $\nu_\pi(x) \coloneqq \mathbb{E}_{\omega \sim \mathcal{T}(\pi,x)} \left[\sum_i \gamma^i R^{\pi(\omega_i)}(\omega_i)\right]$ with discount factor $\gamma < 1$ and trace space $\mathcal{T}$. Our goal is to achieve the optimal reward $\nu^*(x) \coloneqq \sup_{\pi \in \Pi} \nu_\pi(x)$. For a value estimate $v : X \to \mathbb{R}$ and an action a, the one-step lookahead is defined as

$$Q_v^a(x) \coloneqq R^a(x) + \gamma \sum_{x' \in X} \mathcal{P}^a(x, x') \cdot v(x').$$

For each state x, the maximum lookahead w.r.t. all actions is $Q_v^*(x)$. A policy π is called *greedy* if Q_v^π and Q_v^* are equal for all states. The Bellman error, denoted by $\|v - Q_v^\pi\|$, is the maximum difference between Q_v^π and v, over all states in X, i.e. the L_∞ distance.

Example In our example, a factored representation of the rewards R_i^d is $R_i^d(\mathrm{W}) = 1$ and $R_i^d(\mathrm{B}) = 0$, for all $1 \leq i \leq 3$. This gives rise to an exponentially smaller representation compared to an explicitly represented MDP, where the reward function for d would have 8 entries.

Linear Value Functions

Even if all reward and transition functions are scoped functions, the value function ν_π may still be unstructured, i.e. computing ν_π might require the construction of an exponentially big mapping (Guestrin et al. 2003). However, the value of a policy can be approximated as the weighted sum of m basis functions $h_i : X \to \mathbb{R}$. Given weights w_i for each h_i, the value of a state x is defined as a weighted sum $\nu_w(x) \coloneqq \sum_{i<m} w_i h_i(x)$. Note that the efficiency of the algorithm we verify here crucially depends on the fact that

(a) (b)

$X \mid_{scope(\mathcal{P}_1^d)}$	$\mathcal{P}_1^d(X, X'_1 \mapsto \mathrm{W})$
$X_1 \mapsto \mathrm{W}, X_3 \mapsto \mathrm{W}$	0.9
$X_1 \mapsto \mathrm{B}, X_3 \mapsto \mathrm{W}$	0.2
$X_1 \mapsto \mathrm{W}, X_3 \mapsto \mathrm{B}$	0.7
$X_1 \mapsto \mathrm{B}, X_3 \mapsto \mathrm{B}$	0.1

(c)

Figure 1: (a) Ring network of 3 machines. (b) Variable dependencies of the default action. (c) Probabilities of C_1 working in the next step, for every state of C_1 and C_3.

h_i is a scoped function and $scope(h_i) \subset \mathbb{N}_{<n}$ for most $i \in \mathbb{N}_{<n}$.

For each action choice a and basis function h_i, we can compute its expected evaluation g_i^a, defined as $\sum x' \in X.\ \mathcal{P}^a(x, x') \cdot h_i(x')$, in the successor state. As g is independent of the concrete weights, it can be computed once for a set of basis functions, and is then cached for efficiency. In Isabelle/HOL, we prove that $g_i^a(x)$ has a structured representation with scope $\Gamma_i^a \coloneqq \bigcup_{j \in scope(h_i)} .\ scope(\mathcal{P}_j^a)$. This also leads to an efficient computation of the Q functions:

$$Q_w^a(x) \coloneqq Q_{\nu_w}^a(x) = R^a(x) + \gamma \sum_{i<m} w_i g_i^a(x).$$

Example The value functions in the ring domain can be approximately represented using the basis function $h_0 = 1$ and one function per machine: $h_i = 1$ if $X_i = \mathrm{W}$ and 0 otherwise, for $1 \leq i \leq 3$. Note that since these basis functions have very limited scopes, the accuracy of the best possible approximation of the value function is also limited.

Approximate Policy Iteration

Approximate Policy Iteration (API) is a variant of policy iteration that exploits structure in MDPs to scale to large systems (Guestrin et al. 2003). API is an iteration algorithm, where each iteration consists of three parts: policy evaluation, policy improvement and Bellman error computation. The algorithm terminates when either a timeout t_{max} is reached, the error dips below a threshold ϵ, or the weights assigned to the basis functions converge. In Isabelle/HOL, the algorithm is implemented as a function $api(t, \pi, w)$ (Alg. 1), that takes as inputs a time step t, weights for the basis functions w and a policy π. The initial call to the algorithm is $api(0, \pi^0, w^0)$ where $w_0 = 0$ and π^0 is some greedy policy w.r.t. w^0. An iteration of API first uses the current policy π to first compute updated weights w', then a new greedy policy π', and finally the Bellman error err of π'. If the termination condition is met, the algorithm returns the current iteration, weights and policy, as well as the error and whether the weights converged. Otherwise, api is called recursively.

Algorithm 1 (Approximate Policy Iteration)**.**

$api(t, \pi, w) \coloneqq$
  if $t \geq t_{max}$ **or** $err \leq \epsilon$ **or** $w_=$
    then $(t, \pi', w', err, w_=)$
    else $api(t+1, \pi', w')$
  where $w' \coloneqq upd\_w(\pi)$
    $\pi' \coloneqq greedy\_\pi(w')$
    $err \coloneqq factored\_err(\pi', w')$
    $w_= \coloneqq w' = w$

We structure and decouple the algorithm using locales. Conceptually, this usage of locales is similar to using modules in programming languages. Here, we use locales to postulate the existence of three functions (upd_w, $greedy\_\pi$, $factored\_err$) along with their specifications:

Specification 1 (upd_w)**.** A decision list policy a list representation of policies where each entry (called branch) is a pair (t, a) of a partial state and an action. To select an action in a state x, we search the list for the first branch where x is consistent with t. Now fix a decision list policy π, let $w' = upd\_w(\pi)$. Then $\nu_{w'}$ is the best possible estimate of ν_π: $\|\nu_{w'} - \nu_\pi\| = \inf_w \|\nu_w - \nu_\pi\|$.

Specification 2 ($greedy\_\pi$)**.** For all weights w, $greedy\_\pi(w)$ is a greedy decision list policy for ν_w.

Specification 3 ($factored\_err$)**.** Given weights w and a greedy decision list policy π for ν_w, $factored\_err$ determines the Bellman error: $factored\_err(\pi, w) = \|Q^*_w - \nu_w\|$.

For now, we merely state specifications for the algorithms API builds upon, only later will we show how to implement the specifications concretely. This approach keeps the assumptions on individual parts of the algorithm explicit and permits an easier exchange of implementations, e.g. in our developments one may swap the LP certification algorithm for a verified LP solver implementation. It also facilitates gradual verification of software: the correct behavior of the software system can be proved top-down starting from assumptions on each component. In the following sections we show that these specifications have efficient implementations. See **??** for an overview of all components.

The choice of the specifications and the decomposition of the algorithm into components is roughly based on the presentation of the algorithm by Guestrin et al.. We identified the above specifications after multiple iterations. There is usually a trade-off between the simplicity of the specification and the complexity of the implementation: the smaller the internal complexity, the more complex the external complexity, i.e. the interactions between algorithms.

Within the context of the locale, assuming all specifications, we can derive the same error bounds as presented by Guestrin et al.. One exemplary important observation is that if the weights converge during API, then in the last step the Bellman error equals the approximation error. This leads to the following *a posteriori* optimality bound:

Theorem 1. *Let* $api(w_0, \pi_0) = (t', \pi, w, err, True)$. *Then* $(1-\gamma)\|\nu^* - \nu_w\| \leq 2\gamma \cdot err$.

Policy Improvement

Given weights w, the policy improvement phase determines a greedy policy w.r.t. ν_w. The policy takes the form of a decision list, where each element is a pair of a partial state and an action. The main idea for an efficient computation is to only consider actions better than the default action. This notion is made precise by the bonus function δ_a (Alg. 2) with scope $\mathbf{T}_a$. Guestrin et al. do not include Γ^d in $\mathbf{T}_a$, which we assume to be an oversight in the definition, since the behavior of the default action does influence the bonus. Unless components cancel out, the scope of a function difference is the union of the scopes of both arguments. Finally, we concatenate the branches π_a for every action but d, add the default action as a fallback and sort the decision list policy by decreasing bonus. Here, the empty map with no entries is $\bot_M$, and $dom(\bot_M) = \emptyset$. We can show that $greedy\_\pi$ satisfies Spec. 2, since action selection proceeds in the order of decreasing bonus.

Algorithm 2 (Decision List Policy)**.**

$greedy\_\pi \coloneqq$
  $sort\_\pi((\bot_M, d, 0) :: concat([\pi_a \mid a \in A - \{d\}])$
  where $\pi_a \coloneqq [(x, a, \delta_a(x)) \mid \delta_a(x) > 0, x \in X_{|\mathbf{T}_a}]$
    $\delta_a \coloneqq Q^a_w - Q^d_w$
    $\mathbf{T}_a \coloneqq scope(R^a) \cup \bigcup_{i \in \mathbf{I}_a} \Gamma^a_i \cup \Gamma^d_i$
    $\mathbf{I}_a \coloneqq \{i < m \mid effects_a \cap scope(h_i) \neq \emptyset\}$

Factored Bellman Error

The Bellman error $\|Q^*_w - \nu_w\|$ is an indicator of the degree of optimality of a policy. An inefficient computation would enumerate every state, and return the maximum error. However, for a decision list policy, we can compute the error incurred by each branch separately. The total error then equals the maximum error of any branch (Alg. 3). For now, assume that we have a function $branch\_err$ that computes the error for a single branch, i.e. the maximum error for any state that selects the respective branch. These are all states that are consistent with the current branch t, but did not match any prior branch $t' \in ts$ of the policy, formally $X_{(t,ts)} \coloneqq \{x \in X.\ t \sqsubseteq x \wedge \forall t' \in ts.\ t' \not\sqsubseteq x\}$. Hence, we also need to pass the prefix of the decision list policy to $branch\_err$. Note that if the branch is selected by no state its error defined as $-\infty$. We show that if Spec. 4 is met and π is a greedy policy w.r.t. w, then $factored\_err$ satisfies Spec. 3.

Algorithm 3 (Factored Bellman Error)**.**

$factored\_err \coloneqq \sup_{i < len(\pi)} branch\_err(\pi_i, map(fst, \pi_{:i}))$

Specification 4 ($branch\_err$)**.** Given a prefix of a policy π, i.e. the current branch (t, a), and a list of partial states ts from prior branches, $branch\_err(t, a, ts) = \sup_{x \in X_{(t,ts)}} |Q^a_w(x) - \nu_w(x)|$.

Branch Error Consider the branch (t, a) of the policy, for a partial state t an action a. The states of all prior branches form the list of partial states ts. To find the Bellman error of

the current branch, we need to maximize $|Q^a_w(x) - \nu_w(x)|$ w.r.t. states $x \in X_{(t,ts)}$. Note that for all states x

$$Q^a_w(x) - \nu_w(x) = \sum_{i<r_a} R^a_i(x) + \sum_{i<m} w_i(h_i - \gamma g_i)(x). \quad (1)$$

Suppose we had an algorithm to efficiently compute the maximum sum of scoped functions. In that case we could determine the error of a branch. Again, we specify an algorithm max_Σ doing exactly that (Spec. 5). In Alg. 4, we call max_Σ with the functions from (1). To restrict the maximization to the states $X_{(t,ts)}$, we instantiate all functions with the partial state t. Additionally, we define the functions $\mathcal{I}'$ that evaluate to $-\infty$ on states that select a different branch of the policy. Hence, these states are ignored in the error computation. We also apply max_Σ to the negated functions to compute the absolute value of the error. Finally, we formally prove that $branch\_err$ satisfies Spec. 4.

Specification 5 (Variable Elimination)**.** For scoped functions fs, $max_\Sigma(fs) = \sup_{x \in X} \sum_{f \in fs} f(x)$.

Algorithm 4 (Branch Error)**.**

$$branch\_err \coloneqq \max(max_\Sigma(fs \cdot \mathcal{I}'), max_\Sigma(-fs \cdot \mathcal{I}'))$$
$$\textbf{where } rs \coloneqq [R^a_0, \ldots, R^a_{r_a}]$$
$$ws \coloneqq [w_0(h_0 - \gamma g_0), \ldots, w_m(h_m - \gamma g_m)]$$
$$fs \coloneqq map(inst_t, rs \cdot ws)$$

Algorithm 5 (Variable Elimination)**.**

$$max\_step(i, fs) \coloneqq (i + 1, e :: E')$$
$$\textbf{where } (E, E') \coloneqq partition(f \mapsto \mathcal{O}(i) \in scope(f), fs)$$
$$e \coloneqq x \mapsto \max_{y \in X_{\mathcal{O}(i)}} \sum_{f \in E} f(x_{\mathcal{O}(i) \mapsto y})$$

$$max_\Sigma(fs) \coloneqq \sum_{f \in fs'} f(\bot_M)$$
$$\textbf{where } (\_, fs') \coloneqq max\_step^n(0, fs)$$

Variable Elimination The specification for max_Σ can be efficiently implemented with a variable elimination algorithm (Alg. 5). In each iteration, the algorithm selects a dimension of the state space, collects all functions that depend on this dimension in a set E. It then creates a new function e that maximizes all functions in E over that dimension of the state space. The algorithm keeps track of a set of functions to maximize and the number of the current iteration. Since the number of operations performed by the algorithm varies greatly with the elimination order, the variables can be reordered with a bijection $\mathcal{O} : \mathbb{N}_{<n} \to \mathbb{N}_{<n}$. We formally prove that max_step preserves the maximum of the current list of functions, thus max_Σ meets Spec. 5.

Value Determination

After a new candidate policy is found, it is evaluated in the value determination phase. Since the exact value function can not in general be represented as a linear combination of the basis functions, we aim to find weights for the basis functions that minimize the approximation error (see Spec. 1), for which we need to solve a linear program (LP). The structure of the algorithm that finds optimal weights is analogous to the factored Bellman error computation. For each branch of the policy, we generate a set of LP constraints (according to Spec. 6) that expresses the approximation error incurred by this branch. The union of all constraints is then

$$weight\_lp \coloneqq \bigcup_{i<len(\pi)} branch\_lp(\pi_i, map(fst, \pi_{:i})).$$

Variables of the LP are the approximation error ϕ and the weights w. The LP is optimized for minimal ϕ, the values of the variables w in an optimal solution determine the new weights. Given a set of LP constraints cs, $\langle cs \rangle_{LP}$ denotes the set of feasible solutions. We show that for any optimal solution $(\phi^*, w^*) \in \langle weight\_lp \rangle_{LP}$, setting $upd\_w \coloneqq w^*$ satisfies Spec. 1. We also formally prove that the LP always has an optimal solution, since the set of potentially optimal solutions is compact.

Specification 6 ($branch\_lp$)**.** Given a partial state t, an action a, a list of partial states ts, $branch\_lp$ constructs an LP that minimizes the approximation error for the states $X_{(t,ts)}$:

$$(\phi, w) \in \langle branch\_lp(t, a, ts) \rangle_{LP} \longleftrightarrow$$
$$\forall x \in X_{(t,ts)}.\ \phi \geq |Q^a_w(x) - \nu_w(x)|.$$

LPs for Branches For each branch of the policy, we proceed similarly to the Bellman error computation: we create two constraint sets, for positive and negative errors respectively (Alg. 6). We omit the scopes for brevity. At this level, we make use of another algorithm min_lp. Its first input C is a list of m scoped functions, the second input b is another list of scoped functions. Now $min\_lp(C, b)$ creates an LP that minimizes $Cw - b$ w.r.t. w over all states (see Spec. 7). The definitions of C and b are analogous to the Bellman error computation. It then follows that $branch\_lp$ fulfills Spec. 6.

Algorithm 6 (Branch LP)**.**

$$branch\_lp \coloneqq min\_lp(C, -b \cdot \mathcal{I}') \cup min\_lp(-C, b \cdot \mathcal{I}')$$
$$\textbf{where } b \coloneqq map(inst_t, [R^a_i \mid i < r_a])$$
$$C \coloneqq map(inst_t, [h_i - \gamma g^a_i \mid i < m])$$

Specification 7 (min_lp)**.** $min\_lp(C, b)$ generates an LP that minimizes $Cw - b$ over all weights w:

$$(\phi, w) \in \langle min\_lp(C, b) \rangle_{LP} \longleftrightarrow$$
$$\forall x \in X.\ \phi \geq \textstyle\sum_{i<len(C)} w_i C_i(x) + \sum_{i<len(b)} b_i(x).$$

Factored LP Construction The algorithm min_lp resembles max_Σ, so we only point out the challenges encountered during verification. Full details can be found in the formalization. There are two significant modifications we made to the algorithm to make verification feasible. First, the original algorithm may create equality constraints that constrain variables to $-\infty$. Since these constraints are not supported by the LP solvers we use, we formally prove that one can modify the algorithm to omit such constraints without changing the set of feasible weights. Second, the combination of LP constraints in the definition of $weight\_lp$ requires some care, to avoid interactions between LP variables created in different branches. The min_lp algorithm creates new (private) LP variables and we need to make sure that these variables have distinct names for each branch. This issue was not discussed

by Guestrin et al.. The problem can be solved by adding a tag to each generated variable. The tags contain t, a, and a flag to differentiate the two invocations of min_lp in each branch. For distinct tags p and p' we can then show that the solutions to the union of two constraint sets are equivalent to the intersection of the solution spaces of the individual constraint sets (concerning ϕ and w):

$$\langle min\_lp(p, C, b) \cup min\_lp(p', C', b') \rangle_{LP} = \langle min\_lp(p, C, b) \rangle_{LP} \cap \langle min\_lp(p', C', b') \rangle_{LP}.$$

We show that min_lp creates an LP that is equivalent to the explicit (potentially exponentially larger) LP that has a constraint for each MDP state. It immediately follows that min_lp satisfies Spec. 7, which completes our correctness proof of API. With this approach to algorithm verification using loosely coupled locales, min_lp and max_Σ are not tied to MDPs and are thereby reusable components.

Code Generation

We now discuss the process of deriving a verified efficiently executable version from the verified abstract algorithm discussed above. To do so, we follow the methodology of program refinement (Wirth 1971), where one starts with an abstract, potentially non-executable version of the algorithm and verifies it. Then one devises more optimised versions of the algorithm, and only proves the optimizations correct in this latter step, thus separating mathematical reasoning from implementation specific reasoning. This approach was used in most successful algorithm verification efforts (Klein et al. 2009; Esparza et al. 2013; Kanav, Lammich, and Popescu 2014). In this work, the three most important stages are the initial abstract algorithm, an implementation with abstract data structures, and finally an implementation with concretized data structures. As a last step, we export verified code for API in the programming language Scala.

Refinement using Locales Our implementation of stepwise refinement is based on Isabelle/HOL locales. For each locale of the abstract algorithm, we define a corresponding locale where we define the executable version of the algorithm. Finally, we relate the abstract version of the MDP to the concrete version. For each definition, we then show that corresponding inputs lead to corresponding outputs, i.e. our abstract algorithm and the implementation behave the same. At this point in the refinement, data structures remain abstract interfaces, with the concrete implementations chosen only later. We use the data structures provided by the Isabelle Collections Framework (Lammich and Lochbihler 2019) for code generation and we extend them with a data structure for scoped functions, represented as a pair of a function and a set for its scope. The data structure also provides an operation to evaluate a function on its full scope for memoization.

Certification of LP Solutions An implementation of API depends on efficient LP solvers. In our verified implementation, we use precise but unverified LP solvers and certify their results. This avoids implementing a verified, optimized LP solver but retains formal guarantees – the tradeoff here is that the unverified LP solver might return solutions that cannot be certified. At the cost of performance, it is also possible to connect the formalization to an existing simplex implementation for Isabelle/HOL (Spasić and Marić 2012). To achieve formal guarantees, the LP has to be solved exactly, i.e. using rational numbers. Two potential candidates for precise LP solvers are QSopt_ex (Applegate et al. 2007) and SoPlex (Bestuzheva et al. 2023). For larger LPs in our setting, SoPlex demonstrated more consistent performance.

We certify optimality using the dual solution and the strong duality of linear programming. In our formalization we also formally prove that infeasibility and unboundedness can be certified similarly using farkas certificates or unbounded rays. The linear program is first preprocessed to a standard form: variable bounds and equality constraints are reduced to inequality constraints. The constraints of the resulting LP are of the form $Ax \leq b$, with no restrictions on x. During benchmarking of the certification process, the normalization operation usually applied to rationals after each operation proved to be very costly. For certification, we represent rational numbers as pairs that are never normalized, which leads to faster certificate checking in our experiments.

The exported Scala program takes an arbitrary function from linear programs to their solutions as input. If this function returns invalid solutions, they are rejected by the certificate checker, so there are no assumptions we need to place on the LP solver. However, there is the implicit assumption that the LP solver is deterministic. Since we are working in a fragment of a functional programming language, calling the LP solver twice on the same problem should lead to the same solution. In theory, a nondeterministic LP solver could be misused to lead to inconsistencies. As we do not compare LP solutions in our algorithm and SoPlex is actually deterministic, this problem does not impact our verified software. A more general solution to the problem could be the use of memoization or to model the nondeterminism with monads.

	Ring				Star	
n	$\mathbf{t(s)}$	$\mathbf{t_{LP}(s)}$	**Constrs**	**Vars**	$\mathbf{t(s)}$	$\mathbf{t_{LP}(s)}$
1	0.27	0.02	74	41	0.34	0.03
3	0.89	0.15	1258	693	0.50	0.05
5	1.89	0.46	4378	2455	0.69	0.08
7	3.78	0.98	9418	5305	0.98	0.14
9	6.74	1.82	16378	9243	1.30	0.22
11	12.44	3.36	25258	14269	1.52	0.29
13	20.95	5.31	36058	20383	1.76	0.36
15	34.69	8.67	48778	27585	2.28	0.50
17	58.30	16.86	63418	35875	2.89	0.64
19	92.19	30.25	79978	45253	3.69	0.80

Table 1: Evaluation on the ring and star domains. The first column denotes the number of clients. For each topology, the first two columns give the total running time and time spent in the LP solver. For the ring domain, we also show the number of LP constraints and variables generated.

Experimental Evaluation We show the practicality of our verified implementation by applying it to both the ring and star topologies from (Guestrin et al. 2003). Note that all numerical computations have to be performed with infinite

precision, which substantially impacts the performance. We run our implementation on an Intel i7-11800H CPU and set the discount factor to 0.9 in all our experiments. In all runs, the weights converged after max. 5 iterations. The results of the experiments (Table 1) show that the algorithm can deal with ring networks of half a million states and 20 actions. For larger networks, the precise mode of the LP solver SoPlex cannot find a rational solution. The experiment shows that our implementation of linear programming certification can process linear programs with tens of thousands of constraints. For the simpler star topology, we can handle systems with 2^{40} states in 45s.

Discussion

Our work combines and integrates a wide range of tools and formalization efforts to produce a verified implementation of API. In Isabelle/HOL, we build on the formal libraries for LPs (Thiemann 2022), MDPs (Hölzl 2017), linear algebra, analysis (Hölzl, Immler, and Huffman 2013), probability theory (Hölzl and Heller 2011), and the collections framework (Lammich and Lochbihler 2019). Furthermore, we certify the results computed by precise LP solvers. In total, our development is comprised of approximately 20,000 lines of code, two thirds of which concern code generation. We show how to facilitate locales, powerful automation in Isabelle/HOL, and certification to develop complex formally verified software. We also make the case that the process of algorithm verification provides a detailed understanding of the algorithm: all hidden assumptions are made explicit, while we modularize the algorithm into components with precisely specified behaviour.

The methodology presented here can be applied to a large number of algorithms since the algorithm we verified is a seminal algorithm for solving factored MDPs, combining a large number of concepts that are widely used in many contexts, like AI (Delgado, Sanner, and De Barros 2011; Osband and Roy 2014; Deng, Devic, and Juba 2022) and model checking (Hinton et al. 2006; Dehnert et al. 2017).

Applications of interactive theorem provers in verification and formalizing mathematics have been recently attracting a lot of attention (Avigad and Harrison 2014; Avigad 2023; Massot 2021). In most applications, especially in computer science (Esparza et al. 2013; Klein et al. 2009) and AI (Bagnall and Stewart 2019; Selsam, Liang, and Dill 2017), the emphasis is on the difficulty of the proofs, whether that is due to many cases or complex constructions, etc., and how theorem proving helped find mistakes or find missing cases in the proofs. A distinct feature of this work is that its complexity comes from the large number of concepts it combines, shown in Fig. **??**, which is a more prevalent issue in formalizing pure mathematics. Our project has contributed a better restructuring of the algorithm and untangling of the different concepts, leading to better understandability.

Multiple directions can be considered to extend and build on our work. An alternative to using exact LP solvers is to use their highly optimized floating point counterparts that do not calculate exact solutions. In this case, one needs to certify the error bound derived from a dual solution to the LP. Then the error bound has to be incorporated in the error analysis of the algorithm to obtain formal guarantees. Moreover, we may initialize the algorithm with weights computed by an unverified, floating point implementation of API to reduce number of iterations performed by the verified implementation. In Isabelle/HOL, some of the correctness proofs for code generation can be automated using tools to transfer theorems. Furthermore, the ergonomics for instantiating locales and inheriting from other locales could be improved for situations with many locale parameters.

Related Work Several formal treatments of MDPs have been developed in the theorem provers Isabelle/HOL (Hölzl 2017; Schäffeler and Abdulaziz 2023; Chevallier and Fleuriot 2021; Hartmanns, Kohlen, and Lammich 2023) and Coq (Vajjha et al. 2021). All developments verify algorithms for explicitly represented MDPs, which limits their practical applicability to solve large MDPs. We adapt and integrate the implementation of Schäffeler and Abdulaziz with our formalization. The algorithm we verified in Isabelle/HOL was first presented by (Guestrin et al. 2003). An approach to LP certification certification with Isabelle/HOL has been done before as part of the Flyspeck project (Obua and Nipkow 2009), where the feasibility of a solution to a linear program is checked by Isabelle/HOL. The work presents a method that uses dual solutions produced by floating-point LP solvers to find bounds on the objective value of an LP. The tool Marabou for neural network verfication uses Farkas vectors to produce proofs for its results (Isac et al. 2022).

Acknowledgements

This project was funded in part by the Deutsche Forschungsgemeinschaft – 378803395 (ConVeY).

References

Abdulaziz, M.; and Koller, L. 2022. Formal Semantics and Formally Verified Validation for Temporal Planning. In *The 36th AAAI Conference on Artificial Intelligence (AAAI)*.

Abdulaziz, M.; and Kurz, F. 2023. Formally Verified SAT-Based AI Planning. In *The 37th AAAI Conference on Artificial Intelligence (AAAI)*.

Abdulaziz, M.; and Lammich, P. 2018. A Formally Verified Validator for Classical Planning Problems and Solutions. In *The 30th International Conference on Tools with Artificial Intelligence (ICTAI)*.

Applegate, D. L.; Cook, W.; Dash, S.; and Espinoza, D. G. 2007. Exact Solutions to Linear Programming Problems. *Operations Research Letters*.

Avigad, J. 2023. Mathematics and the Formal Turn. arxiv:2311.00007.

Avigad, J.; and Harrison, J. 2014. Formally Verified Mathematics. *Commun. ACM*.

Bagnall, A.; and Stewart, G. 2019. Certifying the True Error: Machine Learning in Coq with Verified Generalization Guarantees. In *The 33rd AAAI Conference on Artificial Intelligence (AAAI)*.

Ballarin, C. 2014. Locales: A Module System for Mathematical Theories. *J. Autom. Reason.*

Bestuzheva, K.; Besançon, M.; Chen, W.-K.; Chmiela, A.; Donkiewicz, T.; van Doornmalen, J.; Eifler, L.; Gaul, O.; Gamrath, G.; Gleixner, A.; Gottwald, L.; Graczyk, C.; Halbig, K.; Hoen, A.; Hojny, C.; van der Hulst, R.; Koch, T.; Lübbecke, M.; Maher, S. J.; Matter, F.; Mühmer, E.; Müller, B.; Pfetsch, M. E.; Rehfeldt, D.; Schlein, S.; Schlösser, F.; Serrano, F.; Shinano, Y.; Sofranac, B.; Turner, M.; Vigerske, S.; Wegscheider, F.; Wellner, P.; Weninger, D.; and Witzig, J. 2023. Enabling Research through the SCIP Optimization Suite 8.0. *ACM Trans. Math. Softw.*

Budde, C. E.; Dehnert, C.; Hahn, E. M.; Hartmanns, A.; Junges, S.; and Turrini, A. 2017. JANI: Quantitative Model and Tool Interaction. In *The 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems*.

Chevallier, M.; and Fleuriot, J. D. 2021. Formalising the Foundations of Discrete Reinforcement Learning in Isabelle/HOL. *CoRR*.

Dehnert, C.; Junges, S.; Katoen, J.-P.; and Volk, M. 2017. A Storm Is Coming: A Modern Probabilistic Model Checker. In *The 29th International Conference on Computer Aided Verification*.

Delgado, K. V.; Sanner, S.; and De Barros, L. N. 2011. Efficient Solutions to Factored MDPs with Imprecise Transition Probabilities. *Artificial Intelligence*.

Deng, Z.; Devic, S.; and Juba, B. 2022. Polynomial Time Reinforcement Learning in Factored State MDPs with Linear Value Functions. In *The 25th International Conference on Artificial Intelligence and Statistics*.

Esparza, J.; Lammich, P.; Neumann, R.; Nipkow, T.; Schimpf, A.; and Smaus, J.-G. 2013. A Fully Verified Executable LTL Model Checker. In *25th International Conference on Computer Aided Verification (CAV)*.

Guestrin, C.; Koller, D.; Parr, R.; and Venkataraman, S. 2003. Efficient Solution Algorithms for Factored MDPs. *J. Artif. Intell. Res.*

Hartmanns, A.; Junges, S.; Quatmann, T.; and Weininger, M. 2023. A Practitioner's Guide to MDP Model Checking Algorithms. In *TACAS (1)*, volume 13993 of *Lecture Notes in Computer Science*, 469–488. Springer.

Hartmanns, A.; Kohlen, B.; and Lammich, P. 2023. Fast Verified SCCs for Probabilistic Model Checking. In *The 21st International Symposium on Automated Technology for Verification and Analysis (ATVA)*.

Hinton, A.; Kwiatkowska, M. Z.; Norman, G.; and Parker, D. 2006. PRISM: A Tool for Automatic Verification of Probabilistic Systems. In *The 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*.

Hölzl, J. 2017. Markov Chains and Markov Decision Processes in Isabelle/HOL. *J. Autom. Reason.*

Hölzl, J.; and Heller, A. 2011. Three Chapters of Measure Theory in Isabelle/HOL. In *2nd International Conference on Interactive Theorem Proving (ITP)*.

Hölzl, J.; Immler, F.; and Huffman, B. 2013. Type Classes and Filters for Mathematical Analysis in Isabelle/HOL. In *4th International Conference on Interactive Theorem Proving (ITP)*.

Isac, O.; Barrett, C.; Zhang, M.; and Katz, G. 2022. Neural Network Verification with Proof Production. In *The 22nd Conference on Formal Methods in Computer-Aided Design (FMCAD)*.

Junges, S.; Jansen, N.; Katoen, J.; Topcu, U.; Zhang, R.; and Hayhoe, M. M. 2018. Model Checking for Safe Navigation Among Humans. In McIver, A.; and Horváth, A., eds., *Quantitative Evaluation of Systems - 15th International Conference, QEST 2018, Beijing, China, September 4-7, 2018, Proceedings*, volume 11024 of *Lecture Notes in Computer Science*, 207–222. Springer.

Kanav, S.; Lammich, P.; and Popescu, A. 2014. A Conference Management System with Verified Document Confidentiality. In *The 26th International Conference on Computer Aided Verification (CAV)*.

Keller, T.; and Eyerich, P. 2012. PROST: Probabilistic Planning Based on UCT. In McCluskey, L.; Williams, B. C.; Silva, J. R.; and Bonet, B., eds., *Proceedings of the Twenty-Second International Conference on Automated Planning and Scheduling, ICAPS 2012, Atibaia, São Paulo, Brazil, June 25-19, 2012*. AAAI.

Klein, G.; Elphinstone, K.; Heiser, G.; Andronick, J.; Cock, D.; Derrin, P.; Elkaduwe, D.; Engelhardt, K.; Kolanski, R.; Norrish, M.; Sewell, T.; Tuch, H.; and Winwood, S. 2009. seL4: Formal Verification of an OS Kernel. In *22nd ACM Symposium on Operating Systems Principles 2009 (SOSP)*.

Lahijanian, M.; Wasniewski, J.; Andersson, S. B.; and Belta, C. 2010. Motion planning and control from temporal logic specifications with probabilistic satisfaction guarantees. In *IEEE International Conference on Robotics and Automation, ICRA 2010, Anchorage, Alaska, USA, 3-7 May 2010*, 3227–3232. IEEE.

Lammich, P.; and Lochbihler, A. 2019. Automatic Refinement to Efficient Data Structures: A Comparison of Two Approaches. *J Autom Reasoning*.

Leroy, X. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM*.

Massot, P. 2021. Why Formalize Mathematics?

Nipkow, T.; Paulson, L. C.; and Wenzel, M. 2002. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*.

Obua, S.; and Nipkow, T. 2009. Flyspeck II: The Basic Linear Programs. *Ann Math Artif Intell*.

Osband, I.; and Roy, B. V. 2014. Near-optimal reinforcement learning in factored MDPs. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 1*, NIPS'14, 604–612. Cambridge, MA, USA: MIT Press.

Sanner, S. 2010. Relational Dynamic Influence Diagram Language (RDDL): Language Description.

Schäffeler, M.; and Abdulaziz, M. 2023. Formally Verified Solution Methods for Infinite-Horizon Markov Decision Processes. In *The 37th AAAI Conference on Artificial Intelligence (AAAI)*.

Selsam, D.; Liang, P.; and Dill, D. L. 2017. Developing Bug-Free Machine Learning Systems With Formal Mathematics. In *The 34th International Conference on Machine Learning (ICML)*.

Spasić, M.; and Marić, F. 2012. Formalization of Incremental Simplex Algorithm by Stepwise Refinement. In *The 18th International Symposium on Formal Methods*.

Thiemann, R. 2022. Duality of Linear Programming. *Arch. Formal Proofs*.

Vajjha, K.; Shinnar, A.; Trager, B. M.; Pestun, V.; and Fulton, N. 2021. CertRL: Formalizing Convergence Proofs for Value and Policy Iteration in Coq. In *The 10th International Conference on Certified Programs and Proofs (CPP)*.

Wirth, N. 1971. Program Development by Stepwise Refinement. *Commun. ACM*.

Younes, H. L.; and Littman, M. L. 2004. PPDDL1. 0: The Language for the Probabilistic Part of IPC-4. In *Proc. International Planning Competition*.