Formally Verified Solution Methods for Markov Decision Processes

Maximilian Schäffeler[1], **Mohammad Abdulaziz**[1,2]

[1]Technische Universität München, Germany
[2]King's College London, United Kingdom

Abstract

We formally verify executable algorithms for solving Markov decision processes (MDPs) in the interactive theorem prover Isabelle/HOL. We build on existing formalizations of probability theory to analyze the expected total reward criterion on finite and infinite-horizon problems. Our developments formalize the Bellman equation and give conditions under which optimal policies exist. Based on this analysis, we verify dynamic programming algorithms to solve tabular MDPs. We evaluate the formally verified implementations experimentally on standard problems, compare them with state-of-the-art systems, and show that they are practical.

Introduction

Despite the impressive advances in the capabilities of different types of AI systems, it is becoming clear that one major hurdle to their wide adoption is the lack of trustworthiness of these systems. This has prompted researchers to study techniques to boost the trustworthiness of AI systems in different areas, like machine learning (Selsam, Liang, and Dill 2017; Katz et al. 2017), planning (Abdulaziz, Gretton, and Norrish 2019; Abdulaziz and Lammich 2018), and model-checking (Esparza et al. 2013). However, one of the areas of AI where there is still a lot to be done regarding trustworthiness is software for solving Markov decision processes (MDPs). MDPs are models for systems where a decision-maker selects actions with random outcomes to maximize long-term rewards. Such systems with uncertainty occur in wide-ranging areas, e.g. planning, reinforcement learning, model checking, and operations research. Depending on the area, the questions one asks about the model might be different, e.g. in planning the aim might be to find a policy that chooses an action for a robot in every state, with some optimality guarantees on the accrued rewards, while in model-checking the aim might be, for instance, to find what the expected delays of a real-time system are. Furthermore, a lot of safety-critical systems could be modeled as MDPs, e.g. a policy (aka strategy) could be used to navigate an autonomous vehicle. Like with other AI systems, for such applications, it is strongly desirable to have trustworthy programs to reason about and solve MDPs.

A methodology with notable success in developing provably correct software involves the use of interactive theorem provers (ITPs). Success stories of the use of ITPs to develop trustworthy software include a formally verified OS kernel (Klein et al. 2009), a verified compiler for C (Leroy 2009), and verified implementations of a multitude of algorithms (Nipkow, Eberl, and Haslbeck 2020).

In this work, we study the application of ITPs to the development of trustworthy software for reasoning about and solving MDPs using iterative methods. However, using ITPs to develop verified implementations of these iterative methods brings its own particular set of challenges, compared to developing other types of verified algorithms. First, at a mathematical level, formal proofs of correctness of MDP solving algorithms in ITPs need a combination of diverse and significantly non-trivial formal mathematical libraries and concepts, e.g. probabilities, limits, (co)recursion, and probabilistic transition systems. Second, at an implementation level, multiple challenges exist, e.g. should the algorithm be implemented imperatively or functionally, what kind of implementation of numerics should be used, etc. Last, an even bigger challenge is the overall architecture of the verified system: should it be a verified implementation of an iterative method, or should we use an unverified system to perform the computation and produce a certificate, which is later validated using a formally verified certificate checker.

In addressing those challenges, multiple factors play in, like the feasibility of the verification, the reusability of the results, and the efficiency or practicality of the verified implementation. Although previous authors (Vajjha et al. 2021; Chevallier and Fleuriot 2021) have addressed the problem of verifying iterative methods for MDPs using ITPs, all of them focused on the abstract mathematical challenges.

In this paper, we comprehensively tackle all three challenges. At the abstract mathematical level, using the ITP Isabelle/HOL (Nipkow, Paulson, and Wenzel 2002) and following the exposition of Puterman (Puterman 1994), we develop a general formalization of MDPs with rewards. Based on that, we first formally prove correct the backward induction algorithm for finite-horizon MDPs. Second, for infinite-horizon problems, we model four fundamental iteration-based methods in Isabelle/HOL, namely, value iteration, policy iteration, modified policy iteration, and splitting-based methods. We prove the correctness of all methods against

a formal specification of expected total discounted reward. Compared to previous attempts (Vajjha et al. 2021; Chevallier and Fleuriot 2021), at a mathematical level, our formalization is more reusable. We also show the correctness of the algorithms against a simpler definition of expected total discounted reward.

As our second contribution, we devise the first executable verified implementations of the four algorithms of which we are aware. In the process, we fix a mistake in a textbook correctness proof of Gauss-Seidel value iteration. We experimentally evaluate our implementations of the four algorithms on standard probabilistic planning problems and show that they are practical. Our implementations use efficient data structures and can solve planning problems with millions of transitions.

Finally, we experimentally show that combining our verified infinite-horizon implementations with an unverified implementation yields significant performance improvements. In particular, we show that one can use a fast floating-point implementation to perform all the iterations and then use the formally verified implementation for the last iteration, effectively having the best of both worlds.

Background

Isabelle/HOL

An ITP is a program that implements a formal mathematical system, in which definitions and theorem statements are written, and a set of axioms or derivation rules, using which proofs are constructed. To prove a fact in an ITP, the user provides high-level steps of a proof, and the ITP fills in the details, at the level of axioms, culminating in a formal proof. We performed the formalization in this paper using the interactive theorem prover Isabelle/HOL (Nipkow, Paulson, and Wenzel 2002), a theorem prover for Higher-Order Logic. Roughly speaking, Higher-Order Logic can be seen as a combination of functional programming with logic. Isabelle/HOL supports the extraction of the functional fragment to executable code (Haftmann and Nipkow 2007).

Isabelle is designed for trustworthiness: following the Logic for Computable Functions (LCF) approach (Milner 1972), a small kernel implements the inference rules of the logic, and, using encapsulation features of abstract data types, it guarantees that all theorems are actually proved by this small kernel. Around the kernel, there is a large set of tools that implement proof tactics and high-level concepts like algebraic data types and recursive functions. Bugs in these tools cannot lead to invalid proofs, but only to error messages when the kernel refuses a proof.

The formalization, our verified implementation, and detailed benchmarks are available online.[1]

Probability Theory

Probability theory was formalized in Isabelle/HOL primarily by Hölzl based on a library of measure theory (Hölzl 2013). Let Ω be the sample space of the probability space P. We denote the expectation of a random variable $X : \Omega \to \mathbb{R}$ by $\mathbb{E}_{\omega\sim P}[X(\omega)]$, it is defined via the Lebesgue integral. We write $\mathcal{P}(\Omega)$ for the set of all probability spaces over Ω.

[1] https://github.com/schaeffm/mdps-isabelle-hol

Bounded Functions

Our formal analysis of MDPs makes use of bounded functions. For a set X and a complete normed vector space V, the space of bounded functions $X \to_b V$ contains all functions where the image of X is bounded w.r.t. the norm on V. We define a pointwise addition and scaling operation, and a partial order for bounded functions. The uniform norm makes bounded functions a complete normed vector space.

The set of bounded linear functions $V \to_L W$ between normed vector spaces consists of all linear transformations that perform bounded scaling. For finite-dimensional vector spaces, bounded linear functions correspond to matrices. Equipped with the operator norm $\|A\| \coloneqq \sup_{v\in V} \|Av\|/\|v\|$, bounded linear functions also form a normed vector space. We formalize the following closed-form formula for the geometric series of contractive bounded linear functions, that is central to our analysis:

Theorem 1 (Puterman 1994, Corollary C.4)**.** *Let $A : V \to_L W$ with $\|A\| < 1$, then $\sum_{i\in\mathbb{N}} A^i = (1 - A)^{-1}$.*

Bounded linear functions are available in the Isabelle/HOL distribution as a subtype of the function type. A type for bounded continuous functions is also present in the library, which we generalize to bounded functions. Introducing special function types has the advantage that we can then show that they are instances of several vector space type classes (Haftmann and Wenzel 2006) and profit from an existing library of theorems and overloaded notation for vector spaces.

Markov Decision Processes

In this section we give an overview of our formalization of the mathematical concepts needed for specifying the different algorithms on MDPs and their correctness criteria. We closely follow the exposition in (Puterman 1994, Chapters 2-6). However, the most notable outcome of our formalization is that we give a more explicit construction of the trace space of the MDP and we also correct mistakes in the proofs.

Markov decision processes (Bellman 1957) model systems where an agent acts in an environment that exhibits randomized behavior. The dynamics of MDPs evolve over a succession of epochs, during each of which the agent analyzes the current state of the environment, chooses an action, obtains a reward, and transitions to a new state. The goal of the agent is to optimize the action selection to maximize the accrued rewards. We distinguish between finite and infinite horizon problems, where the number of epochs is finite or infinite respectively. Our formalization defines MDPs on general state and action spaces, but our analysis of solution methods only covers discrete state and action spaces.

Definition 1 (Discrete Markov Decision Process with Rewards)**.** *A Markov decision process is composed of discrete sets of states S and actions A, a Markov kernel of transition probabilities $K : S \times A \to \mathcal{P}(S)$, a bounded reward function $r : S \times A \to_b \mathbb{R}$, and state rewards $r_N : S \to_b \mathbb{R}$. For*

each state-action pair, r gives a real-valued reward, while r_N denotes the final value of a state (for finite-horizon problems). Furthermore, we assume the existence of a non-empty set of enabled actions $A_s \subseteq A$ for each state $s \in S$.

We use *locales* (Ballarin 2014) to define MDPs in Isabelle/HOL. A locale can be seen as a formal mathematical context that introduces constants and assumptions, in which we develop our formalization. Locales can be instantiated, e.g. with a concrete MDP. This requires a proof that discharges the assumptions, yielding all the theorems proved within the locale. There already exists a formalization of MDPs in the Archive of Formal Proofs (Hölzl 2017). However, this formalization does not support stochastic action choice, an important component of our analysis. Hence we develop a more flexible definition of MDPs ourselves.

Construction of the Trace Space

In each epoch, the agent determines the action with a decision rule $d : S \to \mathcal{P}(A)$, i.e. it chooses a probability distribution over the enabled actions. A deterministic decision rule is a function $d : S \to A$ that respects enabled actions. In general, the decision rule employed by the agent may depend upon the history of all previous decisions and states visited. Histories are alternating sequences of states and actions, they form the set H. A policy $\pi : H \times S \to \mathcal{P}(A)$ can be seen as a mapping from histories to decision rules. We denote the set of all decision rules by D and the set of policies by Π. Markovian policies depend only on the current epoch and may thus be represented as a sequence of decision rules. Stationary policies are policies that only use a single decision rule. In our formalization, each subclass of policies has a different type. We insert explicit coercion functions to translate between different types of policies, but we omit them here for notational economy.

We now fix a policy π and an initial state. That determines the trace space of the MDP, the probability space of infinite sequences of state-action pairs as observed by the agent. The law of this stochastic process is $\kappa : H \to \mathcal{P}(S \times A)$:

$$\kappa(s) \coloneqq \textbf{do}\{\ a \leftarrow \pi(s);\ \text{return}\ (s,a)\ \} \tag{1}$$

$$\kappa(h,s,a) \coloneqq \textbf{do}\{\ s' \leftarrow K(s,a);\ a' \leftarrow \pi(h,s,a,s');\ \text{return}\ (s',a')\ \}. \tag{2}$$

The definition uses the Giry monad (Giry 1982) on probability spaces to elegantly compose a sequence of experiments. A more detailed introduction to the Giry monad and **do**-notation is given in the appendix. Specifically, the law κ shows how to extend a finite trace by a single state-action pair: we sample the next state s' using the Markov kernel K on the current state-action pair from the history, then choose an action, and finally return the next state-action pair. From the law of the stochastic process, we obtain the trace space $\mathcal{T}^\pi : S \to \mathcal{P}((S \times A)^\infty)$ via the Ionescu-Tulcea extension theorem. The theorem was formalized by Hölzl, who used it to construct trace spaces for Markov chains (Hölzl 2017).

Alternatively, the stochastic process can be viewed as a sequence of state-action distributions, one for each epoch. The state-action distribution $P^n_\pi : S \to \mathcal{P}(S \times A)$ at epoch n is obtained as a projection from $\mathcal{T}^\pi$. We prove that P^n_π adheres to the following recursive characterization:

$$P^0_\pi(s) = \textbf{do}\{\ a \leftarrow \pi(s);\ \text{return}\ (s,a)\ \} \tag{3}$$

$$P^{n+1}_\pi(s) = \textbf{do}\{\ a \leftarrow \pi(s);\ s' \leftarrow K(s,a);\ P^n_{\pi'}(s')\ \} \tag{4}$$

$$\text{where}\ \pi'((s_1,a_1),\ldots,t) = \pi((s,a),(s_1,a_1),\ldots,t).$$

From these equations we derive that for a fixed initial state $s \in S$ and an arbitrary policy π, there exists a Markovian policy π_M such that $P^n_\pi(s) = P^n_{\pi_M}(s)$ for all n. This fact allows us to restrict the search for optimal infinite-horizon policies to Markovian policies.

Finite-Horizon MDPs

For a finite horizon N, we denote the value of a policy π as $\nu^\pi_N : S \to_b \mathbb{R}$. It is defined as the expected discounted sum of rewards accrued over time, plus a final state reward:

$$\nu^\pi_N(s) \coloneqq \mathbb{E}_{\omega \sim \mathcal{T}^\pi(s)}\left[\textstyle\sum_{i<N} \lambda^i r(\omega_i) + r_N(\omega_{N,1})\right]. \tag{5}$$

Typically for finite-horizon problems the discount factor $\lambda = 1$. We show that the optimal finite-horizon value $\nu^*_N \coloneqq \sup_{\pi \in \Pi} \nu^\pi_N$ can be computed using backward induction as $\nu^*_N = u^*_0$, where $u^*_N(s) \coloneqq r_N(s)$ and for $t < N$

$$u^*_t(s) \coloneqq \sup_{a \in A_s} r(s,a) + \lambda \mathbb{E}_{K(s,a)}\left[u^*_{t+1}\right].$$

The maximizing actions in each equation constitute an optimal, Markovian, and deterministic policy.

Infinite-Horizon MDPs

We also define the infinite-horizon value $\nu^\pi : S \to_b \mathbb{R}$

$$\nu^\pi(s) \coloneqq \lim_{n \to \infty} \nu^\pi_n(s). \tag{6}$$

For infinite-horizon problems we assume $0 \le \lambda < 1$, as well as $r_N = 0$. The discount factor decreases the relevance of later rewards, and also guarantees that each policy has a finite value. We show that ν^π is a member of the space V_B of bounded functions $S \to_b \mathbb{R}$. Here, we would like to note that both existing formalizations of the expected total discounted reward criterion (Chevallier and Fleuriot 2021; Vajjha et al. 2021) avoid the construction of the trace space and, instead, use the following characterization as their definition of ν^π:

$$\nu^\pi(s) = \textstyle\sum_{i \in \mathbb{N}} \lambda^i\, \mathbb{E}_{P^i_\pi(s)}[r]. \tag{7}$$

We argue that our definition is more natural and simpler, as it is derived from the trace space of the MDP, and is the one used in the textbook exposition, e.g. Puterman 1994, Equation 5.1.3. Thus, we believe it is a superior basis for the verification of specifications in terms of ν^π. However, the definition of ν^π is difficult to work with directly, because reasoning w.r.t. the trace space is complex. We therefore show that our definition and (7) are equivalent.

We further simplify ν^π with the introduction of a vector notation. The reward vector for a decision rule d is defined as $r^d_s \coloneqq \mathbb{E}_{a \sim d(s)}[r(s,a)]$. We also define the operator $\mathcal{P}^n_\pi : V_B \to_L V_B$, a bounded linear function that is equivalent to the n-step transition probability matrix for policy π. $\mathcal{P}^n_\pi$ acts in the same way on a function as a stochastic matrix does on a vector. We prefer statements in vector notation in our

formalization, so we can prove theorems at a high level of abstraction. Now we can formulate ν^π in vector notation for Markovian policies π:

$$\nu^\pi = \textstyle\sum_i \lambda^i \mathcal{P}_\pi^i r^{\pi_i},\ (\mathcal{P}_\pi^i v)_s \coloneqq \mathbb{E}_{(s',a')\sim P_\pi^i(s)}\left[v_{s'}\right]. \quad (8)$$

Ultimately our goal is to maximize the expected total discounted reward by optimizing the policy. We define the value of the MDP $\nu^* \coloneqq \sup_{\pi\in\Pi}\ \nu^\pi$ to be the least upper bound of ν^π ranging over all policies π. As noted earlier, it suffices to consider Markovian policies for optimality.

Policy Evaluation Before searching for an optimal policy, we first tackle the problem of policy evaluation, i.e. the computation of ν^π. Specifically, we only consider a decision rule d. It is possible to compute ν^d exactly, as from Theorem 1 we can show that $\nu^d = (1 - \lambda\mathcal{P}_d)^{-1} r^d$. However, determining an exact solution becomes computationally expensive for large state spaces. A more practical alternative is the approximation using a fixed-point iteration. We define the Bellman operator $L_d(v) \coloneqq r^d + \lambda P_d v$ and observe that $\nu^d = L_d(\nu^d)$. Since L_d is a contraction mapping, the Banach fixed-point theorem establishes that ν^d is its unique fixed-point and that $\lim_{i\to\infty} L_d^i(v) = \nu^d$ for any $v \in V_B$.

Optimality Equations The Bellman operator can be used to approximate ν^*. We therefore introduce the Bellman optimality operator $\mathcal{L} : V_B \to V_B$ where

$$\mathcal{L}(v) \coloneqq \sup_{d\in D}\ L_d(v). \quad (9)$$

The proof that $\mathcal{L}$ is well-defined is missing from Puterman's book, as discussed by Chevallier and Fleuriot 2021. We can prove that the supremum is indeed well-defined since changing the action selection for one state does not influence the value of other states. Next, we prove that every fixed-point of $\mathcal{L}$ is equal to the optimal value ν^*. Since $\mathcal{L}$ is also a contraction mapping, ν^* is actually the unique fixed-point of $\mathcal{L}$ and can be computed with a fixed-point iteration.

Theorem 2 (Fixed point of $\mathcal{L}$)**.** *The optimal value ν^* is the unique fixed point of the optimality operator $\mathcal{L}$. Additionally, $\lim_{i\to\infty} \mathcal{L}^i(v) = \nu^*$ for $v \in V_B$.*

It remains to be shown under which conditions an optimal policy (one that achieves the value of the MDP) exists. A sufficient condition is the existence of the supremum in the definition of $\mathcal{L}(v)$ for every $v \in V_B$. The optimal policy is then the decision rule d^* that maximizes $L_{d^*}(\nu^*)$. If the set of enabled actions in each state is finite, the supremum in $\mathcal{L}$ is always attained. Thus finite MDPs have optimal policies that are stationary and deterministic. Finally, we show that if there exists an optimal policy, there also exists an optimal stationary and deterministic policy.

Infinite-Horizon Algorithms

A novelty of our work is that we verify optimized and executable algorithms to compute optimal policies. Verifying such executable algorithms pertains to two tasks. The first is formalizing the algorithm in the logic of the theorem prover, usually as a non-executable recursive function. This function should capture the mathematical essence of the algorithm, but excludes implementation details. Then one proves the desired mathematical properties of this function, most notably termination and that the algorithms compute optimal policies. In the second step, we formally verify executable versions of the algorithms at hand. We now discuss the first step and then later discuss the second.

From here on, we assume that both S and A are finite. This ensures the existence of optimal policies and allows us to extract executable code. We shortly present the basic value iteration and policy iteration algorithms and then give a more detailed exposition of the optimized variants, namely modified policy iteration and Gauss-Seidel value iteration. For finite-horizon MDPs, we compute optimal policies using the backward induction algorithm described earlier.

The value iteration algorithm is based on Theorem 2. It repeatedly applies the Bellman optimality operator $\mathcal{L}$ to an initial estimate of the value function and stops as soon as successive iterates are sufficiently close in distance to achieve the desired accuracy. On the other hand, the policy iteration algorithm performs a direct search in the space of deterministic decision rules (Puterman 1994, Section 6.4). It alternates between evaluating a candidate policy and improving it, until the policy stabilizes. Exact policy evaluation involves solving a system of linear equations, for which we use a verified implementation of the Gauss-Jordan algorithm (Thiemann and Yamada 2016).

Modified Policy Iteration Value iteration and policy iteration can be considered extreme instances of modified policy iteration (Algorithm 1) (Puterman 1994, Section 6.5). In each step, value iteration only approximates the value of the current policy with a single iteration of the Bellman operator, while policy iteration considers infinitely many iterations. Modified policy iteration places a varying amount of policy evaluation steps between each policy improvement phase. When the initial value estimate v is conservative, i.e. $v \leq \mathcal{L}(v)$, the algorithm terminates with a policy that is optimal upto an a priori error bound (ϵ-optimal policy). The convergence proof of modified policy iteration is based on the observation that its iterates dominate the iterates of value iteration but never exceed ν^*.

Gauss-Seidel Value Iteration Finally, Gauss-Seidel value iteration (Algorithm 2) is a variant of value iteration where the value estimates are updated in-place (Puterman 1994, Section 6.3.3). The algorithm delivers an ϵ-optimal policy. It converges at least as fast as value iteration, because updated estimates are used immediately, not only in the next iteration. In practice, often substantially fewer iterations are required until convergence. We now assume that the state space of the MDP is a subset $\{0, 1, \ldots, n\}$ of the natural numbers and each iteration proceeds from low to high states. Thus we can interpret bounded linear functions as matrices.

Puterman shows that Gauss-Seidel value iteration is an instance of a class of algorithms called splitting methods. These use regular splittings (Q_d, R_d) of the linear function $(1 - \lambda P_d) = Q_d - R_d$. Regularity of a splitting means that Q_d^{-1} and R_d are nonnegative matrices. The algorithm proceeds the same way as value iteration with the Bellman operator replaced by $G_d(v) \coloneqq Q_d^{-1}(r^d + R_d v)$. For the Gauss-Seidel method we split $P_d = P_d^L + P_d^U$ into a

strictly lower triangular part P_d^L and an upper triangular part P_d^U and choose the regular splitting $Q_d = (1 - \lambda P_d^L)$ and $R_d = \lambda P_d^U$. In-place value iteration should follow the equation $v^{n+1} = r^d + \lambda P_d^L v^{n+1} + \lambda P_d^U v^n$ for some d. Rearranging terms then directly yields

$$v^{n+1} = (1 - \lambda P_d^L)^{-1}(r^d + \lambda P_d^U v^n) = G_d(v^n). \quad (10)$$

We define the Gauss-Seidel variant of the Bellman optimality operator $\mathcal{G}(v) \coloneqq \sup_{d \in D} G_d(v)$. We show that for any regular splitting, $\mathcal{G}$ is a contraction mapping and the algorithm converges if the supremum in $\mathcal{G}(v)$ is attained for all $v \in V_B$ and $\sup_{d \in D} \|Q_d^{-1} R_d\| < 1$. The requirement that the supremum has to be attained is overlooked by Puterman, where the existence of such a decision rule is implicitly assumed. We close this proof gap for Gauss-Seidel value iteration. Assuming a total ordering on the states, as part of our formalization, we show that a decision rule that attains the supremum for all states smaller than s can always be extended to an optimal decision rule up to and including state s. Before we state the theorem, let $f(x := y)$ denote the pointwise update of the value of function f at x.

Theorem 3. *Assume that:*

1. *the decision rule d^* maximizes $G_{d^*}(v)$ for all states smaller than s, i.e. $(G_d(v))_t \leq (G_{d^*}(v))_t$, for any $t < s$ and $d \in D_D$, and*
2. *the maximizing action in s is a, i.e. for any $a' \in A_s$, $(G_{d^*(s:=a')}(v))_s \leq (G_{d^*(s:=a)}(v))_s$.*

Then $d^*(s := a)$ maximizes $G_{d^*}(v)$ for all states up to s, i.e. $(G_d(v))_t \leq (G_{d^*(s:=a)}(v))_t$, for any $t \leq s$ and $d \in D_D$.

Proof. We obtain $G_d(v) = r^d + \lambda P_d^U v + \lambda P_d^L G_d(v)$ by rearranging the definition of G_d. Since P_d^L is lower diagonal, the value of $(G_d(v))_t$ depends only on the values of d up to state t. Now, it follows from the assumptions that $d^*(s := a)$ is already maximizing for all $t < s$. Thus it suffices to show that for all deterministic decision rules d, $(G_d(v))_s \leq (G_{d^*(s:=a)}(v))_s$. With $d' \coloneqq d^*(s := d(s))$, we have

$$(P_d^L\ G_d(v))_s \quad (11)$$

$$= (P_{d'}^L G_d(v))_s \qquad P^L \text{ only depends on } d \text{ at } s \quad (12)$$

$$\leq (P_{d'}^L G_{d^*}(v))_s \qquad d^* \text{ optimal, } P^L \text{ is triangular} \quad (13)$$

$$\leq (P_{d'}^L G_{d'}(v))_s. \qquad G_{d^*} \text{ independent of } d^* \text{ above } s \quad (14)$$

So $(G_d(v))_s = (r^d + \lambda P_d^U v + \lambda P_d^L G_d(v))_s \leq (G_{d'}(v))_s$. Finally assumption 2 lets us complete the proof. □

In Puterman's book, a proof of the ϵ-optimality of this splitting-based method is not given, which we supplement. However, that requires us to change the last step of the algorithm from how it was presented originally (Puterman 1994, Section 6.3.3). There, the policy is determined using a basic value iteration step. To prove ϵ-optimality, we need to, instead, use a Gauss-Seidel step to determine the policy.

Algorithm 1: Modified Policy Iteration

Input : $v \in V_B$ where $v \leq \mathcal{L}(v)$, $m : \mathbb{N} \to \mathbb{N}$
for $i \in 0 \ldots$ **do**
  for $s \in S$ **do** $d(s) \leftarrow \arg\max_{a \in A_s}(L_a(v))_s$
  if $d_\infty(v, \mathcal{L}(v)) < \frac{\epsilon(1-\lambda)}{2\lambda}$ **then return** d
  $v \leftarrow L_d^{m_i+1}(v)$

Algorithm 2: Gauss-Seidel Value Iteration

Input : $v \in V_B$
repeat
  $v_{old} \leftarrow v$
  for $s \in S$ **do** $v(s) \leftarrow \max_{a \in A_s}(L_a(v))_s$
until $d_\infty(v, v_{old}) < \frac{\epsilon(1-\lambda)}{2\lambda}$
for $s \in S$ **do**
  $d(s) \leftarrow \arg\max_{a \in A_s}(L_a(v))_s$
  $v(s) \leftarrow \max_{a \in A_s}(L_a(v))_s$
return d

Verifying an Executable Implementation

Our approach to obtaining executable versions of the algorithms is based on step-wise refinement (Wirth 1971). In this approach, an executable specification of an algorithm is derived from an unexecutable one by replacing mathematical structures by data structures. We make heavy use of locales in our refinement. The built-in data-refinement of the executable code generator of Isabelle/HOL proved to be too inflexible, as it does not allow the same mathematical structure to be implemented with different data structures at different places in the algorithm.

Initially, we have abstract definitions of the algorithms as presented above. In the first step, we show that the reward function and the transition system can be represented as finite maps, and that action sets can be represented as finite sets. The interfaces of data structures implementing finite sets and maps are available as locales in the Isabelle/HOL distribution. These locales come with interpretations for concrete data structures based on e.g. balanced trees or hash maps. We prove correctness on the level of the abstract interfaces, which gives us the flexibility to choose between several concrete data structures without modifying proofs. In our final version, we represent an MDP as an array with an entry for each state. The array stores red-black trees that map actions to rewards and transitions.

We use the code generation facilities provided by Isabelle/HOL (Haftmann and Nipkow 2007) to extract executable Standard ML code from the formally verified algorithms. A wrapper parses problems from a file and passes them to the verified algorithms. To obtain numerically accurate results, real numbers are represented as infinite precision rational numbers. This representation comes with an ever greater performance penalty as the number of iterations increases and the fractions become larger. For comparison, we also provide a separate implementation using floating-point arithmetic. However, due to the possibility of floating-point errors accumulating, we lose the formal guarantees.

Notes on the Formalization

Since a main goal of this project is to showcase theorem proving as a methodology to obtain verified algorithms for

probabilistic systems, we note some of the lessons we learnt and the challenges encountered during the formalization. This section should be of particular interest to readers who would be interested in verifying similar algorithms in Isabelle/HOL or in other theorem provers.

Our work builds on formalization efforts in probability theory (Hölzl 2013) and linear algebra from the archive of formal proofs (AFP) and the Isabelle distribution. The construction of MDPs and their trace space extends previous work in Isabelle/HOL on Markov Chains and MDPs (Hölzl 2017). This library of formalized mathematics was crucial to make our verification project viable.

Isabelle/HOL provides us with powerful tools for automated reasoning, such as the proof method *auto*, and access to external automated theorem provers via *sledgehammer* (Paulson and Blanchette 2010). The strong automation combined with the structured proof language Isar (Wenzel 1999) enables the development of maintainable, reusable and human-readable proofs. However, in our formalization we encounter situations where automation breaks. For instance, chains of equations involving mathematical operators that are potentially undefined, like infinite sums or integrals require duplicate work. We first need to show that the operation maintains the desired property, e.g. summability, measurability, boundedness and integrability. Only then can we show that the algebraic manipulation is actually correct. This problem becomes especially apparent with nested sums or integrals. A potential, albeit still preliminary, solution to this problem has been proposed by (Coen and Zoli 2008).

To improve the reusability of our work, we formalize MDPs with arbitrary, uncountable state spaces and probabilistic action choice. Still, the correctness of the verified algorithms only holds for finite-state MDPs. Our initial formalization of the algorithms models the state space of the MDP as a type, i.e. all the states have to be known at compile-time. Initially, our intention was to use the Isabelle tool *types-to-sets* (Kuncar and Popescu 2016) to generalise our formalization to be parameterised by the set of states of the MDP, and get algorithms operating on the corresponding sets of states. Nonetheless, we could not transfer statements involving arbitrary probability spaces automatically using that tool, as its automation is also still preliminary.

Experimental Evaluation

We evaluate our algorithms on a set of standard benchmarks and compare the results to a reference implementation. Furthermore, we show how our implementation can be supported with solutions obtained by unverified solvers.

Setup We benchmark our implementation on explicitly represented MDPs, which are compiled problems from the *International Planning Competition 2018*.[2] Most domains come with multiple instances of different sizes. The problems are parsed by an unverified wrapper and are then handed to the verified solvers. Each problem is given a timeout of four hours and a memory limit of 4 GB. For infinite-horizon problems, we set a discount factor of $\lambda = 0.95$ and require an accuracy of $\epsilon = 0.05$. Finite-horizon MDPs are run with $\lambda = 1$ and a horizon of $N = 50$.

We compare our work to the dynamic programming algorithms available in the probabilistic model checker PRISM (Hinton et al. 2006), that we extended to support discount factors and modified policy iteration. We use PRISM's explicit representation mode and its floating-point arithmetic mode. The point of this setup in PRISM is to compare the efficiency of our verified data structures and algorithms to their unverified counterparts in PRISM. For part of the benchmarks, we use the values generated by another handwritten implementation of Gauss-Seidel value iteration in Rust as initial values for the verified algorithms. That way, we support the verified programs with unverified solutions in the hope of achieving better performance.

Results We give an overview of the results in Table 1. First, we observe that the verified implementations using floating-point arithmetic can often compete with PRISM. We assume that this is in part due to PRISM being a generalist optimized to handle factored systems. An exception here is policy iteration, because PRISM uses an approximate method to compute the value of the policy that is much faster than our exact method based on Gaussian elimination. Comparing our algorithms, we see that policy iteration is only viable for small problems, as the arithmetic complexity of Gaussian elimination grows cubically in the number of states. Using floating-point arithmetic, the optimized algorithms consistently outperform value iteration. When precise arithmetic is used, Gauss-Seidel value iteration becomes comparatively slow. This is because the fractions representing the value estimates become larger more quickly for in-place updates, as they grow already over the course of a single iteration.

Overall, the experiments show a large performance gap between floating-point and precise arithmetic, especially as the number of iterations required to find a solution increases. With each additional iteration, the precise representation of the current value estimates gets more complex. This prompted us to combine the unverified floating-point Rust implementation for infinite-horizon problems with the verified precise arithmetic: the resulting values from the unverified solver are provided as initial values to the verified precise arithmetic implementation. This way we get the best of both worlds: values that are formally guaranteed to be ϵ-optimal, with performance characteristics comparable to unverified implementations.

Conclusion and Discussion

Our work provides a comprehensive formalization of the basics of MDPs with rewards in the interactive theorem prover Isabelle/HOL. We then prove correct optimized algorithms to solve discounted MDPs. We show that it is feasible to solve non-trivial Markov decision processes with verified algorithms. Overall, our formalization is comprised of approximately 13.000 lines of definitions and proofs. One point worth highlighting is that we use the output of an unverified efficient implementation of value iteration as input to a verified implementation that uses precise arithmetic. This

[2] https://ipc2018-probabilistic.bitbucket.io/

	Instances	VI			GS			PI			MPI			Cert.		Fin-Horizon		
		Prism	Verified		Prism	Verified		Prism	Verified		Prism	Verified		VI	GS	Prism	Verified	
			$\mathbb{R}$	$\mathbb{F}$		$\mathbb{R}$	$\mathbb{F}$		$\mathbb{R}$	$\mathbb{F}$		$\mathbb{R}$	$\mathbb{F}$				$\mathbb{R}$	$\mathbb{F}$
academic	2	–	–	1	–	–	1	–	–	–	–	1	1	1	1	1	1	1
traffic	4	4	4	4	4	2	4	4	2	2	4	4	4	4	4	4	4	4
elevators	8	5	2	6	5	1	6	5	2	2	5	2	6	6	6	7	2	6
game-of-life	3	3	–	3	3	–	3	3	–	3	3	–	3	3	–	3	–	3
manufacturer	2	2	–	2	2	–	2	2	–	1	2	1	2	2	2	2	1	2
luck	5	5	5	5	5	5	5	5	2	2	5	5	5	5	5	5	5	5
skill-teaching	8	6	4	7	6	3	7	6	2	4	6	4	8	6	6	6	4	6
tireworld	6	4	4	4	4	4	4	4	2	2	4	4	4	4	4	6	4	4
wildfire	1	–	–	1	–	–	1	–	–	–	–	–	1	1	1	–	–	1
wildlife	8	6	5	8	6	4	8	6	4	6	6	6	8	8	8	8	6	8

Table 1: A table showing the number of instances solved by different algorithms. The first column gives the number of instances per domain. Columns two to five show the performance of the PRISM implementations vs. our implementations of value, policy, Gauss-Seidel, and modified policy iteration, where $\mathbb{R}$ indicates precise arithmetic, while $\mathbb{F}$ denotes floating-point calculations. The sixth column shows the performance of using an unverified implementation of value iteration followed by one last verified iteration. The last column shows the results for the finite-horizon case.

leads to a system with formal guarantees with similar performance characteristics as an unverified system. The results show the feasibility of verifying practical algorithms in the realm of reinforcement learning or probabilistic planning. In addition to an extensive library of existing formal proofs, Isabelle/HOL provides powerful facilities for proof search, automation, and structuring that in combination allow the development of verified software.

Future Work The most immediate extension of our work to handle much larger state spaces is to formalize data structures for factored representations of MDPs. These representations are standard in both model-checking and planning. Furthermore, in many applications, the current state of the environment is only partially known by the agent. Such systems are modeled using partially observable MDPs. Extending our work in this direction is an important step towards the verification of reinforcement learning algorithms. Another interesting direction concerns how to handle arithmetic: instead of relying on floating-point arithmetic which does not account for the accumulation of errors, we might investigate the possibility of using interval arithmetic to provide error-bounds. Lastly, verified Monte Carlo algorithms would allow us to deal with large state spaces.

Related Work There is a number of related verification approaches that have been tried in the general context of artificial intelligence. The first such thread of research is based on using completely automated methods. It has been applied in planning (Eriksson, Röger, and Helmert 2017) and SAT (Wetzler, Heule, and Hunt 2014), and has recently received the most attention in the area of verifying robustness of neural networks (Katz et al. 2017). In this latter application, a specification of a given neural network's robustness is compiled into an SMT formula, which is then automatically proved/disproved by an SMT solver. An advantage of this approach compared to using ITPs is that it is fully automated, i.e. one could prove a neural network safe without fully understanding it. A disadvantage, however, is that SMT solvers are limited in terms of what specifications they can automatically solve. This is most evident when trying to verify properties of algorithms or programs. For instance, they cannot prove a specification stating that a given implementation of value iteration computes an optimal policy.

Another related approach is that of (Selsam, Liang, and Dill 2017) and (Bagnall and Stewart 2019). In that thread of work, the authors develop ITP frameworks to aid in the formal reasoning about machine learning models and neural network architectures. Since they use ITPs, they are able to prove specifications that are more interesting than neural network robustness. For instance, they are able to show that a certain learning algorithm can guarantee a certain generalisation error, which cannot be done using SMT solvers.

MDPs have been formalized in theorem provers before, e.g. to analyze the semantics of pGCL (Hölzl 2017), or verify soundness of off-policy evaluation (Yeager et al. 2022). The most closely related formalization is the project by (Vajjha et al. 2021) who develop the library *CertRL* in the interactive theorem prover Coq. Our work differs in a number of ways: they show correctness of value and policy iteration while we also prove correct more involved algorithms. Furthermore, they do not verify efficient implementations from their formalisations, while we do. They also define the expected total discounted reward as equation (7), while we give a simpler and more natural definition, which needs the construction of the trace space of an MDP. We also show that deterministic decision rules are in fact optimal over all policies, while they only consider deterministic decision rules as candidates for optimal policies. Finally our formalization of MDPs can handle arbitrary state spaces. Furthermore, in Isabelle/HOL there exists a recent formalization of the basics of discrete reinforcement learning (Chevallier and Fleuriot 2021). Their approach uses stochastic matrices instead of the Giry Monad. They also use equation (7) as their definition of the expected reward, only cover finite MDPs, do not discuss executable algorithms, and only prove, at an abstract level, that value and policy iteration solve MDPs optimally.

Acknowledgements

This work was partially funded by the Deutsche Forschungsgemeinschaft Research Training Group CONVEY - 378803395/GRK2428 and the Deutsche Forschungsgemeinschaft Koselleck Grant NI 491/16-1. We thanks Thomas Keller for helpful comments and providing us with benchmark problems.

References

Abdulaziz, M.; Gretton, C.; and Norrish, M. 2019. A Verified Compositional Algorithm for AI Planning. In *The 10th International Conference on Interactive Theorem Proving (ITP)*.

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)*.

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.*

Bellman, R. 1957. A Markovian Decision Process. *Journal of mathematics and mechanics*.

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

Coen, C. S.; and Zoli, E. 2008. A Note on Formalising Undefined Terms in Real Analysis. In *International Workshop on Proof Assistants and Types in Education (PATE)*.

Eriksson, S.; Röger, G.; and Helmert, M. 2017. Unsolvability Certificates for Classical Planning. In *The 27th International Conference on Automated Planning and Scheduling (ICAPS)*.

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)*.

Giry, M. 1982. A Categorical Approach to Probability Theory. In *Categorical Aspects of Topology and Analysis*.

Haftmann, F.; and Nipkow, T. 2007. A Code Generator Framework for Isabelle/HOL. Technical report, Department of Computer Science, University of Kaiserslautern.

Haftmann, F.; and Wenzel, M. 2006. Constructive Type Classes in Isabelle. In *The International Workshop on Types for Proofs and Programs (TYPES)*.

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. 2013. *Construction and Stochastic Applications of Measure Spaces in Higher-Order Logic*. Ph.D. thesis, Technical University Munich.

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

Katz, G.; Barrett, C. W.; Dill, D. L.; Julian, K.; and Kochenderfer, M. J. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *29th International Conference Computer Aided Verification (CAV)*.

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)*.

Kuncar, O.; and Popescu, A. 2016. From Types to Sets by Local Type Definitions in Higher-Order Logic. In *The 7th International Conference on Interactive Theorem Proving (ITP)*.

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

Milner, R. 1972. Logic for Computable Functions Description of a Machine Implementation. Technical report, Stanford University.

Nipkow, T.; Eberl, M.; and Haslbeck, M. P. L. 2020. Verified Textbook Algorithms - A Biased Survey. In *The 18th International Symposium on Automated Technology for Verification and Analysis (ATVA)*.

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

Paulson, L. C.; and Blanchette, J. C. 2010. Three Years of Experience with Sledgehammer, a Practical Link between Automatic and Interactive Theorem Provers. In *PAAR@ IJCAR*.

Puterman, M. L. 1994. *Markov Decision Processes: Discrete Stochastic Dynamic Programming*.

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)*.

Thiemann, R.; and Yamada, A. 2016. Formalizing Jordan Normal Forms in Isabelle/HOL. In *The 5th ACM SIGPLAN Conference on Certified Programs and Proofs (CPP)*.

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)*.

Wenzel, M. 1999. Isar - A Generic Interpretative Approach to Readable Formal Proof Documents. In Bertot, Y.; Dowek, G.; Hirschowitz, A.; Paulin-Mohring, C.; and Théry, L., eds., *Theorem Proving in Higher Order Logics, 12th International Conference, TPHOLs'99, Nice, France, September, 1999, Proceedings*, volume 1690 of *Lecture Notes in Computer Science*, 167–184. Springer.

Wetzler, N.; Heule, M.; and Hunt, W. A., Jr. 2014. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *International Conference on Theory and Applications of Satisfiability Testing (SAT)*.

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

Yeager, J.; Moss, J. E. B.; Norrish, M.; and Thomas, P. S. 2022. Mechanizing Soundness of Off-Policy Evaluation. In Andronick, J.; and de Moura, L., eds., *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, 32:1–32:20. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.