Verifying Value Iteration and Policy Iteration in Coq

A thesis presented to

the faculty of

the Russ College of Engineering and Technology of Ohio University

In partial fulfillment

of the requirements for the degree

Master of Science

David M. Masters

April 2021

This thesis titled

Verifying Value Iteration and Policy Iteration in Coq

by

DAVID M. MASTERS

has been approved for

the School of Electrical Engineering and Computer Science

and the Russ College of Engineering and Technology by

Gordon Stewart

Assistant Professor of Electrical Engineering and Computer Science

Mei Wei

Dean, Russ College of Engineering and Technology

Abstract

MASTERS, DAVID M., M.S., April 2021, Computer Science

Verifying Value Iteration and Policy Iteration in Coq (62 pp.)

Director of Thesis: Gordon Stewart

Reinforcement learning is a growing field of research, but little work is being done to verify the correctness of reinforcement learning algorithms. Researchers are exploring the use of reinforcement learning in safety critical systems such as self-driving cars and autonomous aircraft, so mathematical proofs of correctness of the underlying reinforcement learning algorithms would greatly improve our confidence in the systems that utilize reinforcement learning. This project verifies convergence and optimality of two fundamental reinforcement learning algorithms: value iteration and policy iteration. These algorithms converge and are optimal if they eventually produce an optimal policy. It also is designed to be extensible to future research into verified reinforcement learning.

Acknowledgments

I would like to first thank my advisor, Dr. Gordon Stewart for his support of my research and his guidance during while pursuing my degree. Dr. Stewart was the professor that introduced me to formal verification and he has supported me in all of my research in the subject. I am so grateful for his patience with me as I researched and wrote my thesis.

I also want to thank my committee, Dr. Razvan Bunescu, Dr. David Juedes, and Dr. Charlotte Elster. This has been a complicated year for everyone, and so I even more grateful for the time each of you have set aside for me. Dr. Bunescu and Dr. Juedes's courses provided me with much of the knowledge I needed to complete this thesis.

Robin Kelby, Nathan St Amour, and Tim Steinberger have been incredible lab partners. I am so grateful for all of the encouragement and assistance they have given me over the past few years as we have completed our degrees together.

Finally, I want to express my gratitude to my family for their support. The care and support of my mother and Brad set me up to succeed and their wisdom helped prepare me for all of the challenges I have faced.

Table of Contents

List of Tables

List of Figures

1 Introduction

Although programmers spend much of their time reducing the bugs in their software, bugs in software have become accepted as inevitable, but when safety critical systems have bugs, the results can be catastrophic. For example, in 1996, the Ariane 5 rocket crashed because of overflow errors in the flight control system [CDT17]ariane. Even when systems are not safety critical, bugs in commonly used software can affect thousands of people. For example, in 1990 AT&T's entire long distance network went down because of a misplaced break statement [Bur95]. Even the most rigorous bug testing methods do little more than checking as many edge cases as possible. One cannot, however, be completely sure a program tested this way will not fail because it is impossible to enumerate all potential cases. To fully trust a system, we need mathematically rigorous proofs of correctness.

Many proofs about algorithms are written in natural language, but any programmer who has written code on paper and tried to directly type the code on a computer can immediately see a problem with this method: handwritten code is prone to mistakes in syntax and logic. Even if programmers have their code reviewed by their peers, they cannot be confident in the correctness of their code until they run and test it. Yet computer scientists are comfortable writing proofs about algorithms by hand. Although published proofs are reviewed to a high standard, we should not leave proofs of correctness in safety critical systems up to human error.

Even if an algorithm is proven correct in natural language and has been reviewed thoroughly enough that we are completely confident there are no flaws in the proof, we cannot guarantee that any given implementation of that algorithm is correct and shares the same assumptions as the proof. For example, many algorithms are proven correct over integers of infinite bounds, but many algorithm implementations use fixed length integers which can cause unexpected overflow errors. In the 1990 AT&T bug, the correctness of

the algorithms they were implementing were irrelevant because the bug was due to a misunderstanding of a language feature [Bur95].

Computer verified proofs solve these problems. Assuming the proof verifier is correct, any proof it accepts can be guaranteed to be valid. In the Coq proof assistant, for example, a programmer can write a program, prove it fits some specification, and extract the program to an executable [CDT17]. This process eliminates human error on the correctness of the proof and guarantees the extracted implementation fits a specification.

For its potential benefits, computer verified proofs get little attention among researchers. Computer scientists are often more interested in pushing the limits of what algorithms can do, rather than developing a verified theoretical framework for their research. Reinforcement learning is a current area of research that receives significant attention from academics and tech giants like Google [HPA+18], but very little development is focused on verified implementations of reinforcement learning algorithms.

Reinforcement learning can be informally defined as the study of policies of agents in some environment [SB18]. Generally, these agents are given a reward depending on their actions and the effects of their actions, and a better policy is one that increases the reward given to an agent. This broad definition of reinforcement learning allows for many environments where it can be applied, such as deciding what stocks to buy to maximize profit or which action to take in chess to maximize an agent's chance of winning.

For any formal verification research on reinforcement learning to be accomplished, researchers must first develop a theoretical framework to formalize reinforcement learning in a formal verification setting. Traditionally, a reinforcement learning scenario is formalized by modeling it as a Markov Decision Process (MDP). MDPs are defined as a set of states, a set of actions, a function representing the probability of transitioning from one state to another, and another function representing the reward for each state transition[SB18].

There are many methods of discovering effective policies in an MDP: Q-learning works in environments where the transition and reward function are not known. Deep learning uses neural networks to extrapolate a more generalized policy from an incomplete search of the state space. Dynamic programming methods can derive the optimal policy in cases where the model of the MDP is completely known and is small enough to be fully explored [SB18].

Dynamic programming methods can guarantee the optimal policy, unlike many other reinforcement learning methods that estimate the optimal policy [SB18], but dynamic programming has two disadvantages: it requires complete knowledge of the dynamic of the MDP and it becomes intractable if the state space grows too large [LDK95]. Rather than making dynamic programming irrelevant, these disadvantages make it foundational: other methods approximate the results of dynamic programming methods, and it is much easier to verify an approximation of a model if the model being approximated is already verified.

Although some research projects have been done on machine learning and MDPs, their scope is limited. A verified perceptron algorithm has been implemented in Coq, but it does not use reinforcement learning [MGS17]. In Isabelle, another proof assistant tool, an implementation of Markov Chains has been developed and verified, but Markov Chains do not allow for variable actions [Höl17], which limits their scope in reinforcement learning.

In my thesis, I have formulated a flexible library for MDPs that is defined as a type class to support many variations and implementations of MDPs. I believe my definitions of MDPs could be extended to support infinite, continuous, and partially observable MDPs without breaking the algorithms I implemented and their proofs of correctness.

To further generalize MDPs, I define the reward and transition probability functions over a type class I developed called Numeric that supports addition, multiplication, negation, and ordering. I prove that integers, reduced rationals, reduced dyadic rationals,

and real numbers are all implementations of Numeric. The variety of supported types allows the computational speed of dyadic rationals, the flexibility of rationals, and the limit analysis of real numbers without changing the structure of the proofs or definitions.

Most of my thesis is focused on proving convergence of value iteration and the optimality of the value function it produces. Much of that analysis of value iteration is done on the limit of value iteration as it is evaluated to infinity, but limits are not computable, as an algorithm cannot be run to infinity. My thesis therefore also focuses on finite runs of value iteration and proves bounds on the error and sub-optimality of value iteration as it is run for a finite number of iterations and a computable method for evaluating these bounds.

The contributions of my research are as follows:

- A generalized "Numeric" definition that supports the basic mathematical operations needed for MDPs, as well as many proofs over the Numeric class.
- Functions over Numerics such as max, argmax, and a generalization of a computable summation library already developed, as well as proofs over these functions.
- A generalized formalization of MDPs, as well as less general extensions of this definition, such as enumerable MDPs.
- An implementation of value iteration on this definition of MDPs, proofs of convergence of the algorithm, and proofs that the policy generated by value iteration converge to the value of the optimal policy.
- Implementation of Banach's fixed point theorem, used for limit analysis of value iteration defined over $\mathbb{R}$.

My MDP library is written using type classes, so it is flexible to being extended. For example, I defined a type class for partially observable Markov decision processes (POMDPs) that extends the base MDP type. Given a model of randomness, my definitions could be extended to define exploratory algorithms such as SARSA or Q-learning. Combining my definitions with verified deep learning could produce a model for deep reinforcement learning.

2 Background

This chapter describes the background information required for the work of in my thesis. This chapter explains Coq, Reinforcement Learning, the OUVerT library, and some important theorems and explores some relevant related research.

2.1 Verified Software

Software verification is the process of mathematically proving correctness of a program[Pie07]. This is done by defining a formal specification of how a program should behave, an implementation of that program, and a proof that the program implements that behavior. The formal specification is a model of what the program must do to be correct. This may include properties such as a termination, restrictions on side effects, and a specification of the output. The implementation of this program is a process that fulfills these specifications.

For a simple example of formal software verification, I will describe the process of verifying a function that returns the maximum of a list. The formal specification of this function could be

$\forall$ list $L, 0 \neq \text{length}(L) \rightarrow (\max(L) \in L) \wedge (\forall x, x \in L \rightarrow x \leq \max(L))$

An implementation of this max algorithm could be

---

procedure MAX(l)

  if length(l) == 0 **then**

    return -infinity

  max′ ← max(l[1:])

  if l[0] < max′ **then**

    return max′

  return l[0]

The verification step would be a proof this implementation matches the specification above, such as a proof by induction showing this specification is correct for the empty list, and for any list this specification is true for, it is also true for any list created by adding one element to the original list.

2.2 Coq

Coq is a proof assistant frequently used in software verification[CDT17]. Coq is a formal language for writing both programs and proofs. While many mathematical proofs are written in natural language, Coq proofs are written in a formal language that can be automatically checked for correctness. This has two major benefits over traditional styles of proofs. The first benefit is greater confidence in the correctness of your proof. Assuming the Coq proof checker is valid, and proof written in and checked by Coq must also be valid. The second benefit of Coq is that it allows proof automation. Coq has useful methods for generating proof steps such as: trying some tactic on each case generated in case analysis, repeating some tactic until it fails or proves the goal, and chaining automation methods together.

Another useful feature of Coq is program extraction. Programs written in Coq can be extracted into other programming languages, such as Haskell and Ocaml, that can be efficiently computed. Program extraction is very useful because it allows the programmer to be confident that their proofs about their program are still valid on the code that they are actually running, assuming the extraction procedure, compiler, and runtime are correct.

2.3 Reinforcement Learning

Reinforcement learning is a branch of machine learning for finding optimal policies for agents in a probabilistic environments with potentially delayed rewards. [SB18] Some reinforcement learning methods find exact solutions, such as value iteration and policy iteration, which require a complete model of the environment. Approximate methods seek

the maximize the optimality of the policy they produce, but they do not need a full model of the environment as they can learn through exploration. Approximate methods also are useful when the state space is large so that enumerating the state space is intractable.

Most current reinforcement learning research is on approximate methods, mostly with deep learning. But the theoretical basis of the approximate methods are the methods that produce an exact solution. To prove properties of approximate methods, it is generally very useful to model and prove properties of what the methods are approximating.

2.3.1 MDPs

MDPs are the formalization of reinforcement learning environments. A MDP is defined as

- A set of states, S
- A set of actions, A
- A transition function T: $S \to A \to S \to \mathbb{R}$, which represents the probability of ending in st_2 given an initial state and action.
- A reward function R: $S \to A \to S \to \mathbb{R}$, which represents the reward of taking an action in state st_1 and ending in state st_2.

The transition function must also fulfill the following properties,

- $\forall s, a, s', 0 \leq T(s, a, s')$
- $\forall s, a, \sum_{s' \in S} T(s, a, s') = 1$

That is, the transition function must be stochastic, or always be non-negative and the must sum to one for any initial state and action. This properties is necessary for T to be a function from state and action to a probability distribution of states.

The state and action sets are not required to be finite, or even countable, but many algorithms, especially those that provide exact solutions, require these sets to be finite.

I will represent examples of MDPs as directed graphs, where each node represents a state and each edge represents a possible state transition. Each edge may be labeled with the reward received, the action taken and the probability of that transition occurring with the labeled action. If the probability label is omitted then it can be assumed to be one. If the action label is omitted it can be assumed that all actions from the initial state have the same transition probability. If there is no edge between two states then the probability of transitioning between the two states is zero.

A deterministic MDP is an MDP where the transition function is a function from state and action to next state, and the reward function is a function from state and action to reward. If T_{det} and R_{det} are the transition and reward functions for a deterministic MDP, this MDP can be represented by a probabilistic MDP with the transition function T and and reward function R:

$$T(s, a, s') := \begin{cases} 1 & \text{if } T_{det}(s, a) = s' \\ 0 & \text{otherwise} \end{cases} \tag{2.1}$$

$$R(s, a, s') := R_{det}(s, a) \tag{2.2}$$

2.3.2 Policies

A policy represents the agent in a reinforcement learning environment. Specifically, a policy is defined as a function from states to actions. The purpose of reinforcement learning algorithms is to optimize policies.

For a policy to be considered optimal, there has to be some metric for evaluating it. An MDP can be simulated with a policy (π) and initial state (s) by sampling the transition probability function using s and $\pi(s)$, and then recording the reward for the transition to the sampled next state (s'). This process is then repeated for s' until a termination

condition is reached, usually either a fixed number of steps or reaching a terminal state. The total reward is the sum of the rewards at each step in the simulation. The sequence of states from this simulation is called a walk.

The expected total reward of a policy and initial state is the average total reward of each walk, weighted by the probability of that walk occurring. This can be estimated by simulating the MDP a large number of times and averaging the total reward. An exact solution of the expected value of policy π can be computed from the following function:

$$E^{\pi}_{n+1}(s) = \sum_{s' \in S} \left[P(s, \pi(s), s') \cdot (R(s, \pi(s), s') + E^{\pi}_{n}(s') \right] \tag{2.3}$$

where $E^{\pi}_{n}(s)$ is the expected reward of simulating policy π for n steps starting in state s and E^{π}_{0} is initialized to some arbitrary value.

The discounted reward is the expected reward if the rewards given in later states have a discount factor γ, so the first reward would be multiplied by γ^0, the second by γ^1, etc. The expect reward of a policy may not converge if the MDP does not always terminate, but the discounted reward always will, assuming an upper bound on the reward function. The expected discounted reward can by computed by:

$$E^{\pi}_{n+1}(s) = \sum_{s' \in S} [P(s, \pi(s), s') \cdot (R(s, \pi(s), s') + \gamma E^{\pi}_{n}(s')] \tag{2.4}$$

This is similar to the Bellman equation:

$$V^{\pi}(s) = \sum_{s' \in S} [P(s, \pi(s), s') \cdot (R(s, \pi(s), s') + \gamma V^{\pi}(s')] \tag{2.5}$$

The Bellman equation defines the function, V, that does not change with one iteration of the expected discounted reward calculation [SB18].

2.3.3 Value Functions and Q Functions

A value function is a function from state to a number. A value function usually represents the expected reward from starting in a specific state. For example, E^{π}_{n} is a value function representing the expected value of evaluating π for n steps.

A Q function is a function from state and action to a number. A Q function usually represents the expected reward from starting in a specific state and taking an initial action. Q functions can be converted to policies with the formula :

$$\pi(s) = argmax_a(Q(s, a)) \tag{2.6}$$

2.3.4 Policy Iteration

Policy iteration is a process that starts with an initial policy and repeats the following two steps until the policy does not improve

- Evaluate the current policy
- Update the current policy to take the best action derived from the policy evaluation

At each iteration, the policy will improve until it is optimal if each policy is evaluated to a sufficient number of steps.

Algorithm 1 Policy iteration

Let θ be some small positive number

function EVALUATE_POLICY(π)

Initialize V as any value function

loop

$\forall s, V'(s) \leftarrow \sum_{s' \in S} (T(s, \pi(s), s')(R(s, \pi(s), s') + \gamma V(s'))$

if $max_s|V - V'| < \theta$ **then return** V'

$V \leftarrow V'$

function POLICY_ITERATION

Initialize π to any policy

$V \leftarrow$ EVALUATE_POLICY(π)

loop

$\pi'(s) \leftarrow argmax_a \left(\sum_{s' \in S} T(s, a, s') \left(R(s, a, s') + \gamma V\left(s'\right)\right)\right)$

$V' \leftarrow$ EVALUATE_POLICY(π)

if $max_s|V - V'| < \theta$ **then return** π'

$\pi \leftarrow \pi'$

$V \leftarrow V'$

2.3.5 Value Iteration

Value iteration is another algorithm for finding the optimal policy for an MDP. Unlike policy iteration, value iteration returns the value function of evaluating the optimal policy rather than the optimal policy itself, but the optimal policy can be derived from the optimal value function with equation 2.7.

$$\pi_{opt}(s) := argmax_a \left(\sum_{s'} [P(s, a, s') \cdot (R(s, a, s') + \gamma V(s')] \right) \tag{2.7}$$

Algorithm 2 Value iteration

Let θ be some small positive number

Initialize V to any value function

loop

$V'(s) \leftarrow max_a \left(\sum_{s' \in S} T(s, a, s') \left(R(s, a, s') + \gamma V\left(s'\right)\right)\right)$

if $max_s|V(s) - V'(s)| < \theta$ **then return** V

$V \leftarrow V'$

2.3.6 Value/Policy Iteration Constraints

For policy iteration and value iteration to work, a few constraints must be fulfilled. First, the transition function must produce a valid distribution (it cannot output negative numbers and must sum to one for all initial states and actions). An additional constraint must be added: the state and action sets must be finite. This property is not explicitly part of the definition of MDPs, but it is necessary for policy and value iteration to be able to enumerate the states and actions.

Two other properties are needed that are often implicitly assumed but are not directly derivable from the previous properties in Coq. The first property is that there is decidable equality on both the state type and the action type (equality of states can be evaluated to a bool and the same for actions). The second property is that the set of all states and actions can be enumerated, meaning that a program can iterate through all states.

2.4 Banach's Fixed Point Theorem

Banach's fixed point theorem states that any function that is a contraction operation has a unique fixed point.[C+07] A function f is a contraction operation in some metric space (X, d) if f is a function $X \to X$ and there exists some $q \in [0, 1)$, such that applying f to any two points reduces their distance by at least a factor of q.

Banach's fixed point theorem can be formalized as: for any metric space (X, d) and $f : X \to X$, if there exists some q, such that $0 \leq q < 1$ and $\forall x, y \in X, d(f(x), f(y)) \leq q \cdot d(x, y)$, then there exists a unique $p \in X$ such that $f(p) = p$.

Another useful property of contraction functions that Banach's fixed point theorem gives us is that when iteratively applied to itself, a contraction function converges to the fixed point. If f is a contraction function and p is the fixed point of f, then $\forall x, d(f(x), p) \leq q \cdot d(x, p)$ because $f(p) = p$.

2.5 OUVerT

OUVerT is a library that provides proofs needed for my work that are not specific to machine learning. This includes implementations and proofs of operations I need such as summations, list maximums, and arithmetic over dyadic numbers.

2.5.1 Dyadics

Dyadic rationals are the subset of rational numbers that can be represented by an integer over two to the power of some integer. Operations on dyadic numbers can be much faster computationally than operations over rational numbers because the space required to store the denominator of rational numbers grows exponentially faster than the space required for the denominator of dyadic numbers.

2.5.2 Enumerables

OUVerT provides a definition of enumerations and enumerable type. An enumeration of a type is defined in OUVerT as a list that has no duplicates and contains all values of a type, and a enumerable type is defined as any type with an enumeration. This allows operations defined over lists to be used on enumerations of a type.

2.6 Related Research

2.6.1 Formal Verification of MDPs and Value Iteration

Proofs of correctness of value iteration and policy iteration are not new. In 1996, Bertsekas and Tsitsiklis proved, not only that policy iteration and value iteration provide the optimal policy when converged, but also that they produce a near optimal policy when run to a sufficient number of iterations [BT95]. Bertsekas and Tsitsiklis proofs were not computer verified.

2.6.2 CertRL

CertRL is another library, released near the end of the development of this project, for verified value iteration and policy iteration [VSP+20]. CertRL has significant overlap with this project, as it also proves convergence and optimality of value iteration and policy iteration in Coq. This thesis and CertRL both focus on proving correctness when the algorithms have converged and proving approximation of correctness for a finite number of iterations.

There are a few differences between this thesis and CertRL. For example, CertRL does not provide a method for extracting value iteration and policy iteration to another programming language. Another difference is that all of the proofs in CertRL are specific to $\mathbb{R}$, but many of the proofs in this thesis apply to, $\mathbb{R}$, $\mathbb{Q}$, $\mathbb{N}$, and dyadic rational numbers.

CertRL proves that value iteration approximates the value of the optimal policy (V^*) to any positive margin of error within a finite number of steps, but this thesis goes a step further and proves that converting the result of value iteration to a policy creates a policy that is approximately optimal. This proof is important because if it was not true, a near optimal value function may not produce a near optimal policy.

3 IMPLEMENTATION

This chapter describes the algorithms implemented and the types modeled in this project. This contains some of the contributions I made to the OUVerT repository, the properties of MDPs I defined, and reinforcement learning algorithms I implemented. The results chapter contains the theorems I proved about the implementations described in this chapter.

3.1 Additions to OUVerT

3.1.1 Numerics and Numeric_Props

To generalize my results, I created a class called Numeric. A type is Numeric if it implements addition, negation, multiplication, injection from naturals, less than, decidable less than, and decidable equality as well as having an additive and multiplicative identity. The class implemented in Coq is shown in listing 3.1

```
Class Numeric (T:Type) ≜
    mkNumeric {
        plus: T → T → T where "n + m" ≜ (plus n m);
        neg : T→ T where "– n" ≜ (neg n);
        mult: T → T → T where "n * m" ≜ (mult n m);
        pow_nat: T → nat → T;
        of_nat: nat → T;
        plus_id: T;
        mult_id: T;
        lt: T→ T→ Prop where "n < m" ≜ (lt n m);
        ltb: T→ T→ bool;
        eqb: T→ T→ bool;
```

}.

Listing 3.1: Numerics Implementation

Another class, called Numeric_Props, contains all of the properties of a Numeric type needed for my proofs. The properties of Numeric_Props is derived from the minimal axioms used in the Coq real numbers axiomatized implementation [CDT17] without the axioms for the multiplicative inverse and completeness.

```
Class Numeric_Props (T:Type) `{numeric_t : Numeric T} ≜
    mkNumericProps {
        plus_id_lt_mult_id: plus_id < mult_id;
        mult_plus_id_l: ∀t : T, plus_id * t = plus_id;
        of_nat_plus_id: of_nat O = plus_id;
        of_nat_succ_l: ∀n : nat, of_nat (S n) = mult_id + of_nat n;
        plus_comm : ∀r1 r2, r1 + r2 = r2 + r1;
        plus_assoc : ∀r1 r2 r3, r1 + (r2 + r3) = r1 + r2 + r3;
        plus_neg_r : ∀r, r + − r = plus_id;
        plus_id_l : ∀r, plus_id + r = r;
        mult_comm : ∀r1 r2, r1 * r2 = r2 * r1;
        mult_assoc : ∀r1 r2 r3, r1 * (r2 * r3) = r1 * r2 * r3;
        mult_id_l : ∀r, mult_id * r = r;
        mult_plus_distr_l : ∀r1 r2 r3, r1 * (r2 + r3) = r1 * r2 + r1 * r3;
        lt_asym : ∀r1 r2, r1 < r2 → ~ r2 < r1;
        lt_trans : ∀r1 r2 r3, r1 < r2 → r2 < r3 → r1 < r3;
        plus_lt_compat_l : ∀r r1 r2, r1 < r2 → r + r1 < r + r2;
        mult_lt_compat_l : ∀r r1 r2, plus_id < r → (r1 < r2 ↔ r * r1 < r * r2);
        pow_natO: ∀t, pow_nat t O = mult_id;
        pow_nat_rec: ∀t n, pow_nat t (S n) = t * pow_nat t n;
        total_order_T : ∀r1 r2, {r1 < r2} + {r1 = r2} + {r2 < r1};
        eqb_true_iff: ∀n m, eqb n m ↔ n = m;
        ltb_true_iff: ∀n m, ltb n m ↔ n < m;
      }.
```

Listing 3.2: Numeric Props

The third class defined for Numeric is Numeric_to_R. This defines Numeric types that have an injection into $\mathbb{R}$ and requires proofs that the arithmetic operators over Numeric are consistent with the the injection into $\mathbb{R}$. For example, adding two Numerics and then injecting the result into $\mathbb{R}$ is equivalent to injecting each Numeric into $\mathbb{R}$ before adding them.

```
Class Numeric_R_inj (T : Type) `{numeric_t : Numeric T} ≜
  mkNumericRInj {
    to_R : T → R;
    to_R_plus: ∀t1 t2 : T, Rplus (to_R t1) (to_R t2) = to_R (t1 + t2);
    to_R_mult: ∀t1 t2 : T, Rmult (to_R t1) (to_R t2) = to_R (t1 * t2);
    to_R_lt: ∀t1 t2 : T, t1 < t2 ↔ Rlt (to_R t1) (to_R t2);
    to_R_neg: ∀t : T, Ropp (to_R t) = to_R (- t);
    to_R_inj: ∀n m : T, to_R n = to_R m → n = m;
  }.
```

Listing 3.3: Numerics To R implementation

Any type that can implement Numeric and Numeric_Props corresponds to a totally ordered ring, which is a totally ordered set equipped with addition and multiplication, but Numeric types also have decidable comparison, which rings may not have.

3.1.2 Max, Argmax, and Mapmax

Argmax and mapmax are functions commonly needed in value iteration, policy iteration, and policy evaluation, and so I created a file with definitions of these functions and many lemmas needed for my proofs over Markov Decision Processes. Max returns

the maximum value in a Numeric list. Mapmax maps a list to a Numeric list and returns the max of the resulting list. Argmax returns an argument that produces the maximal value in mapmax. Max is a function from list Numeric to option Numeric, argmax is a function from list T and T → Numeric to option T, and mapmax is a function from list T and T → Numeric to option Numeric, where T is some arbitrary type. These functions return an option type, because there is no correct return value if the list is empty.

For simpler definitions in my MDPs I created the functions max_nonempty, argmax_nonempty, and mapmax_nonempty. These three functions require a proof that the list argument is nonempty, and so they do not return an option type. In this document the proof argument is omitted when describing algorithms that use these functions, as the specific proof provided is irrelevant.

3.1.3 Enumerable Tables

I defined Enumerable tables as a type class that maps from one enumerable type to another type. There are two major benefits to using enumerable tables rather than functions. Enumerable tables allow saving results of a function call into a table rather than recalculating values every time a value is needed. Therefore enumerable tables provide a method for implementing dynamic programming. Enumerable tables also have the useful property that the type of a enumerable table is itself enumerable if it is a table from one enumerable type to another enumerable type.

Enumerable tables are implemented with an enumeration of the domain type and a list of the co-domain type of equal length. A lookup is done by zipping the domain enumeration and the co-domain list and then finding the pair in the zipped list that contains the lookup value. Therefore a lookup in an enumerable table is a O(n) operation. The lookup function could be reduced to O(log n) with an implementation that uses a balanced binary search tree.

```
Definition max (l : list Nt) : option Nt ≜
match l with
| [] ⇒ None
| x :: l' ⇒
        match (max l) with
        | None ⇒ x
        | Some x' ⇒ if x' < x then x else x'
        end
end.

Definition mapmax (T : Type) (l : list T) (f : T→ l) : option Nt ≜
max (map f l).

Definition argmax (T : Type) (l : list T) (f : T→ l) : option T ≜
match l with
| [] ⇒ None
| x :: l' ⇒
        match (argmax l') with
        | None ⇒ x
        | Some x' ⇒ if f x' ≤ f x then f x' else f x
        end
end

Definition max_ne (T : Type) (l : list T) (f : T→ l) (H : O ≠ length l) : Nt.
Definition mapmax_ne (T : Type) (l : list T) (f : T→ l) (H : O ≠ length l) : Nt.
Definition argmax_ne (T : Type) (l : list T) (f : T→ l) (H : O ≠ length l) : T.
```

Listing 3.4: max, argmax, and mapmax implementation in Coq

3.2 Helper Definitions

This section explains the definitions that were not necessary to prove the correctness of these algorithms, but abstracted parts of other definitions that were frequently repeated. This allowed for shorter and simpler definitions as well as proofs over these definitions.

3.2.1 Value Functions

$|V - V'| \equiv$ value_dist V V' if V and V' are value functions

Figure 3.1: Value distance function notation

Value functions are functions from the state of an MDP to a Numeric. This type appears in nearly every reinforcement learning algorithm implemented in this project. For example, value iteration step is a function from value function to value function, and policy evaluation is a function from policy to value function.

I also define a distance metric over value functions. value_dist v_1 v_2 is defined to be the maximum distance from any v_1(s) to v_2(s) for any s. Or more formally value_dist(v_1,v_2) = $max_s|v_1(s) - v_2(s)|$.

3.2.2 V_to_Q

A helper function I defined, called V_to_Q, converts a value function to a Q function by determining the expected discounted reward of taking an action in a specific state and taking the expected reward after the initial step as input, rather than iteratively computing it. V_to_Q is defined as:

$$V\_to\_Q(V, s, a) = \sum_{s' \in S} [P(s, a, s') \cdot (R(s, a, s') + \gamma V(s')] \tag{3.1}$$

This is called V_to_Q because when it is partially applied with a single value function, it becomes a function from state and action to number, which is a Q function. Because

V_to_Q does one step of look-ahead, the expected discounted reward can be rewritten in terms of V_to_Q:

$$E^{\pi}_{n+1}(s) = V\_to\_Q(E^{\pi}_{n}(s), s, \pi(s)) \tag{3.2}$$

3.3 MDP Definition and Extending Typeclasses

I define a MDP to be a state type, action type, a transition function, and a reward function. The transition and reward functions are of type
State → *Action* → *State* → *Numeric* The transition function represents the probability of ending in a final state given an initial state and an action performed. The reward function is the Numeric reward given to a state action state transition.

```
Record mdp : Type ≜
  {
    St : Type;
    A : Type;
    Trans : St→ A→ St→ Nt;
    Reward : St→ A→ St→ Nt;
  }.
```

Listing 3.5: Coq MDP definition

To perform value iteration and policy iteration, the states and actions must be enumerable, equality on states and actions must be decidable, and there must exist at least one state and action. I defined a type class, mdp_fin, to be a MDP such that these conditions hold. I defined a separate type class mdp_fin_ok to be a mdp_fin that the state and action types are enumerable, the transition function sums to at most 1, and the transition function is non-negative on all inputs. While most definitions of MDPs require the transition function to sum to exactly 1, I realized that this restriction is not necessary to

prove optimality and convergence of value iteration and policy iteration, and so I generalized my definition of the transition function.

The definition of policy evaluation, value iteration, and policy iteration are only defined on MDPs that are of type mdp_fin, but mdp_fin_ok is only used in the proof of correctness of these algorithms. The separation of mdp_fin_ok allows value iteration and policy iteration to be evaluated without providing a proof that the underlying MDP is valid.

3.4 Implementation of Algorithms

```
Definition policy_evaluation_step (p : policy) (v : value_function) (s : state) ≜
V_to_Q v s (p s)

Fixpoint policy_evaluation (n : nat) (p : policy) (v : value_func) : value_func ≜
match n with
        | O ⇒ v
        | S n' ⇒ policy_evaluation n' p (policy_evaluation_step v)
end
```

Listing 3.6: Policy Evaluation Coq

All of the function definitions and theorems about value iteration and policy iteration are in a section. This section has variables for the MDP and the discount factor (γ). This creates a MDP and γ that can be referenced by any definition in this section.

In this project a policy evaluation step is expressed as a function from policy, value function, and state to reward. Policy evaluation is defined in terms of V_to_Q as described in listing 3.6. Unfolding the definition of V_to_Q, policy_evaluation_step becomes

$\sum_{s' \in S}$ [P s (p s) s' · (R s (p s) s' + γ· (v s')]

This is equivalent to the definition of one step of policy evaluation in equation 2.3.

Partial application of a policy to the policy evaluation step function creates a function from value function and state to reward, or equivalently, a function from value function to value function. This is useful because the iterative policy evaluation function can be defined as iteratively applying the output of one policy evaluation step to the next step.

```
Definition value_iteration_step (v : value_func) (s : state) ≜
mapmax_nonempty actions (λ a ⇒ V_to_Q v s a)

Fixpoint value_iteration (n : nat) (v : value_func) : value_func ≜
match n with
| O ⇒ v
| S n' ⇒ value_iteration n' (value_iteration_step v)
end
```

Listing 3.7: Value Iteration Coq

Value iteration step is defined in terms of V_to_Q, similarly to how policy iteration is defined. The major difference between the two functions is that policy evaluation passes the action from the policy to the Q function returned from V_to_Q but value iteration passes the argument that maximizes V_to_Q. The value iteration recursive function iteratively calls value iteration step a fixed number of times, applying the result of the previous value iteration step call to the next step.

```
Definition policy_iteration_step (n : nat) (p : policy) (v : value_func) : (value_func * policy) ≜
let v' ≜ policy_evaluation n p v in
        (v', (λ s ⇒ argmax_nonempty actions (λ a ⇒ V_to_Q v' s a) ) ).

Fixpoint policy_iteration (m n : nat) (p : policy) (v : value_func) : (value_func * policy) ≜
match m with
| O ⇒ v
| S m' ⇒ let (v',p') ≜ policy_iteration_step n p v in
        policy_iteration m' n p' v'
end.
```

Listing 3.8: Policy Iteration Coq

Policy iteration step evaluates a policy to a fixed number of steps and returns an improved policy and the evaluation of the original policy. The evaluation of the original policy is returned because using the evaluation of the previous policy is a common optimization of policy iteration. The policy iteration recursive function iteratively calls policy iteration step a fixed number of times using a fixed number of policy evaluation steps and passes the result of each policy iteration step into the next step.

3.5 Table-based Implementation

Policy evaluation, value iteration, and policy iteration are dynamic programming algorithms, but the implementations described above do not use either tables or memoization. These implementations recalculate the same value many times over, and so they quickly become intractable. To fix this issue, I developed a table-based implementation of these algorithms using the enumerable tables I added to OUVerT. This

allows the result of each step to be saved in a table, significantly reducing the time complexity of these algorithms.

To create a table-based implementation, I first defined value tables, which is a table from state to number. Value tables are used in table-based implementations in a similar way to how value functions are used. One useful property of value tables is that partially applying a value table to the table lookup function produces a value function, as the lookup is a function from table to state to number. Using value tables I created a tabular implementation of V_to_Q, value iteration, policy evaluation, and policy iteration.

3.6 MDP to R

One property of Numeric_to_R is that they must have an injection into $\mathbb{R}$. This can be used to define an injection from MDPs over any Numeric_to_R type to MDPs of $\mathbb{R}$. Because the operations over Numerics (addition,negation,multiplication, etc.) must be consistent with the corresponding operation over $\mathbb{R}$, algorithms defined over MDPs of any numeric type must be consistent with those algorithms defined over the injection into MDPs of $\mathbb{R}$.

This can be useful because the limits of evaluating a policy for a MDP of some Numeric type may not exist in that Numeric type. For example, the limit of evaluating policy for a dyadic MDP may not be dyadic. In the results section I will show how this was useful in proving convergence and optimality of value iteration and policy iteration.

4 Results

In this chapter I review the results that I was able to accomplish and some of the problems I attempted but was unable to solve. This chapter also reviews the performance of my implementations extracted to Haskell.

4.1 OUVerT Proofs

4.1.1 Banach's Fixpoint Theorem

Banach's fixed point theorem gives us a method to prove that a function converges to a fixed point. For my work, I use value functions as a metric space and the value_dist function defined above as a distance function. Then I prove that value iteration step and policy evaluation step are contraction functions. This proves that value iteration and policy iteration converge to a fixed point.

4.1.2 Generalizing Big_sum

Before I started working on this project, OUVerT had functions, called big_sum for summing lists of real numbers, rational numbers, and integers. I created a big_sum function that is generalized over Numeric, and so this big_sum function was able to unify all of the big_sum functions. I also proved all of the properties in the existing big_sum library for my generalized big_sum function.

4.2 RL theorems

4.2.1 Generalized Proofs Over Numeric_Props

I attempted to prove as many results as I could using only properties of Numeric without relying on properties of real numbers. This was for two reasons, the first being that using real numbers in Coq relies on the introduction of axioms, which is best to avoid if possible. The more important reason I preferred to prove results over Numeric rather

$E_n(\pi, v)$:= policy_evaluation_rec p v n (evaluate policy π for n steps starting with value function v)

$E^*(\pi)$:= banach.converge_func (R_eval_pol_banach p) (evaluate policy π to convergence using the banach's fixed point theorem library)

$V_n(v)$:= value_iteration_rec v n (run value iteration for n steps starting with value function v)

V^* := banach.converge_func R_vi_banach (the converged result from value iteration using the banach's fixed point theorem library)

$|V_1 - V_2|$:= value_dist V_1 V_2 (the maximum difference between the two value functions when evaluated with the same state)

V^*_π := value_function_policy V^* (the policy derived from the converged result of value iteration)

Figure 4.1: Notations

than $\mathbb{R}$ is because Numeric proofs are more generalizable. Because $\mathbb{R}$ is an instance of Numeric, any proof over Numeric is automatically a proof for real numbers, integers, dyadics, and rationals, but the converse is not true. Because computers cannot represent all real numbers, real numbers must be extracted to another type, such as floats, but since floats do not have the same properties as real numbers, the proof is not valid for the code it actually extracts to.

The theorems evaluate_policy_contraction and value_iteration_contraction prove that policy evaluation step and value iteration step are contraction functions. All of the

properties about contraction functions I described in the contraction functions section can be directly derived for policy evaluation and value iteration using these two theorems.

Theorem evaluate_policy_contraction: ∀(π : policy) (v1 v2 : value_func),

$|(E_1(\pi, v1) - E_1(\pi, v2)| <= \gamma|v1 - v2|$

Theorem value_iteration_contraction: ∀(v1 v2 : value_func),

$|V_1(v1) - V_1(v2)| \leq \gamma|v1 - v2|$

Another useful property about value iteration that can be proven without using real numbers is that value iteration is an upper bound to policy evaluation. More precisely, for all states, the value function produced by value iteration is an upper bound to the value function produced by an equal number of steps of policy evaluation. This does not prove that value iteration produces the optimal policy, as this does not prove any connection between the value function returned from value iteration and the expected reward from the policy derived from value iteration.

Lemma value_func_eval_ub: ∀(s : St)(p : policy) (n : nat) (v : value_func),

$|E_n(p, v)| \leq |V_n(v)|$

4.2.2 Proofs Over Real Numbers

To prove properties of the limits of value iteration and policy iteration, these properties had to be proven over real numbers. Because my proofs about MDPs so far

have been over generalized Numeric, and real numbers are an instance of Numeric, all of the previously proven properties are automatically proven for real numbers.

The first result I was able to prove exclusively over real numbered MDPs is that, for any policy π, policy evaluation converges to a value function, $E^*(\pi)$. I was also able to prove the converged value function of value iteration (V^*) is a fixed point of value iteration and is equivalent for every initial value function. I proved both of these properties using Banach's fixed point theorem, as policy evaluation and value iteration are contraction functions.

I was also able to prove that evaluating the policy derived from the limit of value iteration is equivalent to the limit of value iteration. Because I have already proven that the value function produced from n steps of value iteration is an upper bound to of the value function of evaluating any policy to n steps, the converged value function from value iteration must be an upper bound to the converged value function from evaluating any policy ($\forall \pi, E^*(\pi) \leq V^*$).

Because the policy produced from running value iteration to convergence is equal to the value function it produced, which is an upper bound to the expected value evaluating any policy, the policy produced from V^* must be optimal.

Another useful optimality property of V^* is that the difference between V^* and the expected value from the policy produced from a value function has an upper bound related to the difference between V^* and that value function. More specifically,

$$\forall v, |V^* - E^*(P_v)| \leq \frac{1+\gamma}{1-\gamma}|v - V^*|$$

This tells us that any value function that closely approximates V^* will produce a near-optimal policy.

Below are the major theorems I proved that were specialized to real numbers.

Lemma value_iteration_R_cauchy_crit: ∀(v : value_func) (s : St),

Cauchy_crit (λ n ⇒ ($V_n(v)$ s))).

value_iteration_R_cauchy_crit proves that the result of value iteration is a Cauchy sequence, and therefore it converges to a real number. This is necessary in many of the other proofs because proving the existence of the number value iteration converges to allows us to reason about that number.

Theorem value_iteration_fixpoint_unique: ∀(v : value_func),

v = $V_1(v)$ → v = V^*.

value_iteration_fixpoint_unique is a property derived from Banach's fixed point theorem that provides a method of proving a given value function is optimal.

Lemma value_iteration_eval_step_fixpoint: ∀s, $E_1(V^*_\pi, V^* s) = V^*(s)$

value_iteration_eval_step_fixpoint proves that if evaluating the result of value iteration does not change the value function; the result of value iteration is a fixed point for the evaluation of its own policy. This allows us to unfold a step of policy evaluation.

Lemma step_value_diff_converge: ∀(v : value_func),

$|V_1(v) - V^*| \leq \gamma |v - V^*|$

step_value_diff_converge proves that the result of value iteration converges towards V^* for any value function this is a direct result of the contraction property of value iteration and the fact that V^* is a fixed point of value iteration.

Lemma value_dist_opt_ub_vi: ∀(v : value_func),

$$(1 - \gamma)|v - V^*| \leq |v - V_1(v)|$$

value_dist_opt_ub_vi provides an upper bound for the difference between any value function and V^* in terms of how much a value iteration step changes a value function.

Lemma value_dist_opt_ub_eval: ∀(v : value_func),

$$(1 - \gamma)|v - V^*| \leq (1 + \gamma)|v - E^*(\text{ value\_func\_policy v })|$$

value_dist_opt_ub_eval provides an upper bound to the distance between a value function and the optimal value function in terms of how close that value function is to expected value of the policy derived from that value function.

Theorem value_iteration_limit_opt: ∀(s : St) π,

$$E^*(\pi, s) \leq E^*(V^*_\pi, s)$$

value_iteration_limit_opt uses the previous proofs to prove that the policy derived from the convergence of value iteration is optimal. However, this does not prove anything about the optimality of value iteration when run to any finite number of steps.

Theorem value_iteration_R_opt_finite_steps: ∀v, ∃n_0, ∀n, $n_0 \leq n \rightarrow$

E^* (value_func_policy (value_iteration_rec v n)) = V^*.

value_iteration_R_opt_finite_steps expands on value_iteration_limit_opt to show that value iteration will eventually produce the optimal policy without needing to completely converge. value_iteration_R_opt_finite_steps first proves that, for any given sub-optimal policy, value iteration will eventually produce a policy that is strictly more optimal. Then it uses the the property that the policy space is finite to prove that value iteration will eventually produce a policy that is strictly more optimal than *all* sub-optimal policies.

The following theorems prove the optimality of policy iteration.

Lemma policy_iteration_R_stable_opt: ∀(p : policy),

E^*(p) = E^*(policy_iteration_R_step p) → E^*(p) = V^*

policy_iteration_R_stable_opt proves that if the expected value of a policy does not change after a policy iteration step, then the expected value of the policy must be optimal.

Theorem policy_iteration_improve: ∀(p : policy) s,

$E^*(p, s) \leq E^*$(policy_iteration_R_step p)s.

policy_iteration_improve proves that policy iteration always produces a policy that is at least as optimal as the original policy assuming that the policy evaluation step was run to convergence.

Theorem policy_iteration_R_converge_opt: ∀(p : policy), ∃n, E^*(policy_iteration_R p n) = V^*.

policy_iteration_R_converge_opt uses policy_iteration_improve and the fact that policies are finite to prove that policy iteration eventually produces the optimal policy if policy evaluation fully converges.

Theorem policy_iteration_not_monotonic : ∀(steps : nat),

∃mdp (p : policy) (v : value_func) (s : St) (γ: R),

$0 \leq \gamma < 1 \wedge E^*$(policy_iteration_step steps p v) < E^*(p)

policy_iteration_not_monotic proves that if the policy evaluation step of policy iteration has not been run to convergence, then policy iteration is not guaranteed to improve the policy. This shows that the assumption policy_iteration_R_converge_opt makes that policy evaluation has converged is a necessary assumption to prove optimality of policy iteration.

4.2.3 Relating Proofs Over Real Numbers to Proofs Over Numerics

Another use of relating Numeric MDPs to real MDPs is that the distance between value functions, including the optimal value function remains consistent.

$(\forall v_1 v_2, to\_R(|v_1 - v_2|) = |to\_R(v_1) - to\_R(v_2)|)$

Because Banach's fixed point theorem gives us a rate that a sequence converges to the optimal value, we have a rate that value iteration over MDP_to_R converges, and also a rate that value iteration over any Numeric type converges. This was useful to generalize the proofs that were originally specialized to $\mathbb{R}$.

4.3 Enumerable Policies

To prove the existence of an optimal policy, I needed to prove that there are a finite number of policies. To do this, I used policy tables rather than policy functions. As part of the definition of MDP_enum, the state and action types must be enumerable and finite, and as I showed above , if a table is from an enumerable type to another another enumerable type, then the type of the table itself is enumerable.

The reason I needed to prove that policies are finite an enumerable was because, although I proved that value iteration would eventually produce a better policy than any specific sub-optimal policy, I needed to prove that value iteration would eventually produce a policy that is at least as optimal any other policy, since if there was an infinite number of policies, there could be an infinite number of sub-optimal policies that get arbitrarily close to optimal. Or more formally,

I had already proved $\forall \pi, \exists i, \forall j \geq i, E^*[\pi] \leq E[value\_func\_policy(V_i)]$, where V_i is the value function produced from i steps of value iteration,

But I needed to prove that $\exists i, \forall j \geq i, \forall \pi, E^*[\pi] \leq E^*[value\_func\_policy(V_i)]$

This proof becomes more straightforward once you can enumerate the policies and have a function that will produce the i such that $E^*[\pi] \leq E^*[value\_func\_policy(V_i)]$, as you can take the mapmax of this function applied to all of the policies to get an i that this property will hold true for all policies.

I then proved that there is a mapping from policy functions to policy tables that will produce a policy with an equivalent expected value. I then used this to prove that value iteration would eventually produce a more optimal policy than all sub-optimal policy functions.

4.4 Performance Tests

Although performance was not the main point of my work, I did some performance tests to test to see if my solution was still tractable. I set up an MDP with 15 states and 4 actions. The states were pairs of integers in the range (0,0) to (2,4) and the actions were integers from 0 to 3. The rewards of reaching each state is shown in table 4.1.

Table 4.1: Test MDP rewards

0	-1	0
-1	10	-1
-2	-5	-1
0	0	1
-1	-1	-1

Table 4.2: Test MDP optimal value function

1.21	4.99	1.21
5.74	1.60	5.87
0.590	5.48	0.758
0.430	1.78	0.907
-0.632	-0.390	0.317

Generated by rounding the results of value iteration

The actions can be labeled as ”up”, ”down”, ”left”, and ”right”. Each action has a 50% probability of moving one tile in intended direction, a 25% probability of moving two tiles in that direction, and a 12.5% probability of moving one tile in each direction perpendicular to the indented direction.

This is an example of a grid world MDP. A grid world is an MDP where the states can be aligned into a grid and the actions represent moving in some direction. Grid world MDPs also often have a probability of moving in the intended direction as well as a probability of moving in an unintended direction. This environment is similar to the frozen lake MDP on openAI gym, which a probabilistic MDP designed to model an agent moving on a slippery surface where they may not go in the intended direction [Dut18].

For my tests I wrote an implementation of value iteration and this MDP in python, using numpy to optimize performance. I also tested value iteration with the Coq implementation extracted to Haskell. The numeric types tested in the extraction were dyadics and floats, and the implementations of the tabular value iteration were also tested. I repeated each test with doubling number of iterations until a test took longer than several minutes or it had run for many more iterations than the other tests. Other than the python test, all of the tests were from Coq code extracted to Haskell. The timing results are shown in figure 4.2.

The Haskell and Python implementation both converged to the same value table. This value table is rounded and visualized in table 4.2. The Haskell implementation took 56 iterations to reach a stable value function and the Python implementation took 60. This difference could be due to my Haskell implementation and numpy rounding differently because of a difference in order of operations.

The fastest method was the numpy implementation of value iteration. I expected the Coq implementation performance to be comparable to the Python implementation, but the python implementation was around 50x as fast. There are two major factors that I believe contribute to this speed difference. The first is that most of the computation in the python implementation is vectorized with numpy, which is much faster than a purely python implementation. The second major factor is the lack of optimization of the Coq

implementation. There are several points in the code that could be optimized that I will describe in the further work section.

The other notable result is how the time scales with the number of iterations for the dyadic implementation. The time it takes to run the dyadic implementation seems to quadruple every time the number of iterations doubles, and this implies the dyadic implementation has a roughly quadratic time complexity. The reason for this is because the dyadic numbers do not round and therefore dyadics must increase in size to accommodate higher precision.

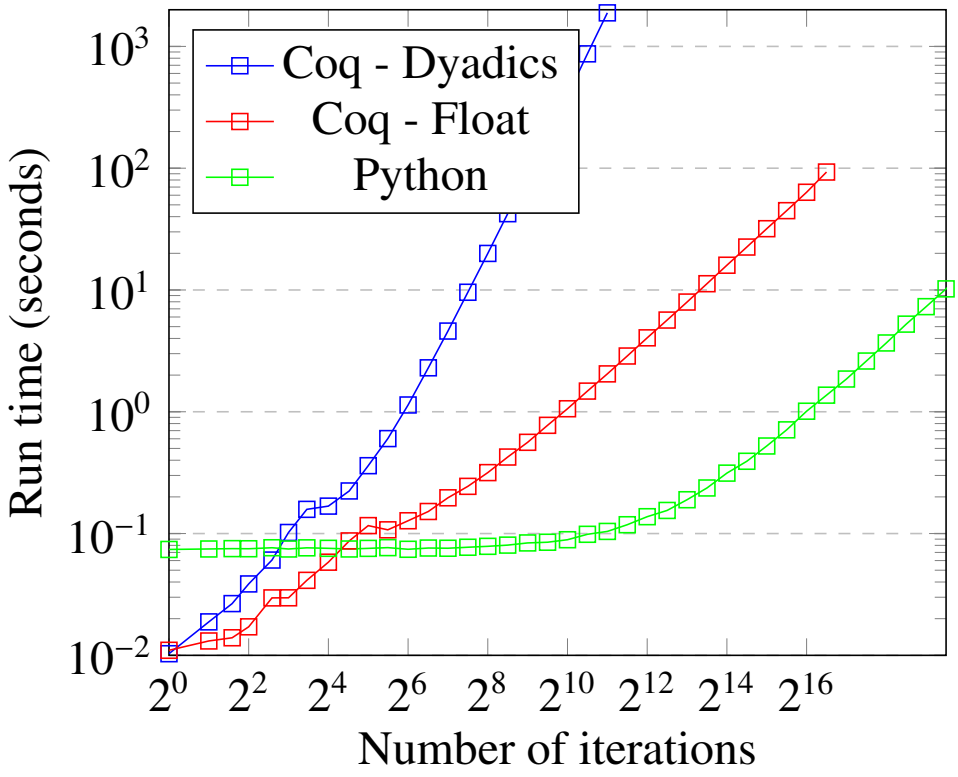

Figure 4.2: Run time of each value iteration implementation

```
def value_iteration(mdp,steps):
    V = np.zeros((S,))
    S,A,T,R,discount = mdp
    for step in range(steps):
        Q = np.sum(T * R + discount * T * V,axis=2)
        V = np.max(Q,axis=1)
    return V
```

Listing 4.1: Python Value Iteration Implementation

5 Obstacles

This chapter describes the obstacles encountered in this project. This chapter is separated into three parts. The first part is about properties of these algorithms that are used in proofs but are only true when these algorithms are run to convergence. The second part is about properties I initially thought would be true and would have been helpful in my proofs, but turned out to be false. The third part is about issues unique to Coq that would not be present in a less formal proof even if convergence was not assumed.

One constraint that is often assumed is that value iteration and policy evaluation are run until convergence. This means that the value function produced from policy evaluation is equivalent to the expected discounted reward of that policy. A few issues arise when attempting to prove properties of these algorithms when run to a finite number of steps by using reasoning that assumes convergence.

The first issue is that proofs that two values are equal become proofs that the difference between the two values are within some bounds. These bounds need to be as tight as possible because they are propagated through multiple proofs that may also grow the error bound, and eventually the bound can grow too large to be useful in a proof.

Another issue is that properties that are true at convergence may not be true for approximations. For example, the converged result of value iteration is a unique fixed point for the value iteration step, but an approximation of the converged result is neither a fixed point nor unique.

Without the assumptions of convergence, proving correctness of these algorithms is no longer simply translating proofs to Coq. In many cases steps in proofs that would otherwise be simple substitution require careful analysis on the error bounds when convergence is not assumed.

5.0.1 Policy Iteration Does Not Always Improve The Policy

It is often stated that policy iteration strictly improves the policy for each policy update step. [SB18] This property is important because it is used in the proof that policy iteration produces the optimal policy. The proof of this property relies on the following property:

$$\forall \pi, E^*(\pi, s) = \sum_{s'} (P(s, \pi(s), s') * (R(s, \pi(s), s') + \gamma * E^*(\pi, s')) \tag{5.1}$$

This property can be proven if policy evaluation has converged because the expected discounted reward of a policy is the fixed point of the policy evaluation function, so the property

$$\forall n, \pi, E^*(\pi) = E_n(\pi, E^*(\pi)) \tag{5.2}$$

also can also be proven from the assumption of convergence. Therefore if policy evaluation has converged,

$$\forall \pi, E^*(\pi, s) = E_1(\pi, E^*(\pi) = \sum_{s}^{'} (P(s, \pi(s), s') * (R(s, \pi(s), s') + \gamma * E^*(\pi, s')) \tag{5.3}$$

The issue is that this property can only be proven if policy evaluation has converged, but this may not happen within a finite number of steps. What can be proven is that

$$v(s) = \sum_{s}^{'} (P(s, \pi(s), s') * (R(s, \pi(s), s') + \gamma * v(s')) \ \pm \text{some bound.} \tag{5.4}$$

Therefore the proof that policy iteration is optimal is not valid if the policy evaluation function has not completely converged. Not only is the proof that the each policy iteration step strictly improves the policy invalid, it is actually false. Bertsekas and Tsitsikl [BT95] provide counterexamples that prove that policy iteration does not always provide the optimal policy if the policy evaluation step is not exact.

This thesis provides a method for generating an MDP with only two states and two actions from a fixed number of steps for policy evaluation that can produce a sub-optimal policy when initialized with an optimal policy. This project also proves the property that,

for any given number of evaluation steps, this function will produce an MDP that policy iteration does not provide a strictly better policy.

Policy iteration not producing an optimal policy is not usually a problem in practice because, as Bertsekas and Tsitsikl [BT95] prove, if the policy evaluation error is small enough, then the produced policy will be very close to optimal.

5.0.2 There May Not Exist an Optimal Policy When Evaluation is Finite

There is guaranteed to exist a policy with an optimal expected reward, which is a property I prove in my project, but there is not guaranteed to exist an policy that has an optimal expected reward when evaluated to a finite number of steps. Consider the following MDP.

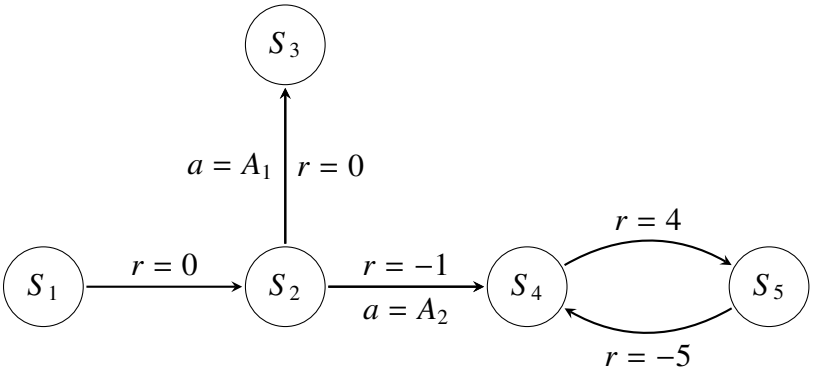

Figure 5.1: Counterexample MDP

Let π_1 be the policy that chooses action A_1 and π_2 be the policy that chooses A_2. All other policies will evaluate to the same value as one of these policies because there is only one state where the action taken matters. Below is a table of policy evaluation with a discount of .5.

Table 5.1: Result of evaluating MDP 5.1

n	$E_n(\pi_1, S_1)$	$E_n(\pi_2, S_1)$	$E_n(\pi_1, S_2)$	$E_n(\pi_2, S_2)$
0	0	0	0	0
1	0	-1	0	0
2	0	1	0	-1/2
3	0	-1/4	0	1/2
4	0	1/4	0	-1/8
5	0	-1/16	0	1/8
6	0	1/16	0	-1/32

This table shows that for any even $n > 1, \pi_2$ is the optimal policy for S_1 but π_1 is the optimal policy for S_2 and the opposite is true for any odd $n > 1$. Although this shows that there may not be an optimal policy for a finite number of evaluation steps, this is not a disproof that an optimal policy always exists if policy evaluation has converged because in each of these policies, the expected value of S_1 and S_2 converge to zero, so both of these policies are optimal.

5.0.3 There is no Ordering of the Expected Value of Policies

It would be helpful if it could be shown that policies could be listed in a way that each policy's expected value is strictly greater than or equal to the expected value of the previous policy for all states. Unfortunately this is not the case whether the policy evaluation is run for a finite number of steps or has converged. Attempting to prove an ordering on policies was originally how I tried to prove that the existence of an optimal policy. Eventually I was able to prove the existence of an optimal policy constructively by proving that value iteration produced the optimal policy.

5.0.4 Value Iteration Does Not Always Improve Optimality

A property that I initially believed to be true is that the expected value of a policy derived from a value iteration step is always at least as optimal as the expected value of the original value function converted to a policy. Although it is true that each step of value iteration lowers some bounds on the sub-optimality of the policy it produces, the exact value of the policy within those bounds can decrease between steps.

5.1 Issues Unique to Coq

5.1.1 Lack of Linear Algebra

Although most of the algorithms about MDPs do not rely on linear algebra, some of the proofs make use of relating policy evaluation to operations on matrices. For example, the policy evaluation step function can be written as

$$v_{n+1} = P_\pi(R_\pi + \gamma v_n) \tag{5.5}$$

Where R_π and P_π are the matrices obtained from applying π to the reward function and transition probability functions respectively. Using linear algebra and the constraints of the transition probability function, it is straightforward to prove

$$[I - \gamma * P_\pi]v \leq [I - \gamma * P_\pi]v' \rightarrow v \leq v' \tag{5.6}$$

Instead, I had to prove the property that is much less straightforward:

$$\forall s, v(s) - \sum_{s'}(P(s, \pi(s), s')(R(s, \pi(s), s') + v(s')) \leq v'(s) - \sum_{s'}(P(s, \pi(s), s')(R(s, \pi(s), s') + v'(s')) \rightarrow \forall s, v(s) \leq v'(s) \tag{5.7}$$

5.1.2 Properties That Are Trivially True But Must Be Proven In Coq

There are many statements that seem trivial but must be rigorously proven in Coq. One example of this is the statement: ”if the policy improvement step always improves the

policy, unless it is already optimal, and there are a finite number of policies, then it will eventually produce the optimal policy." Issues like this often make a theorem take significantly longer to prove in Coq than they would in other contexts.

5.1.3 Lack of Automation Over Numerics

There is a tactic in Coq called lra that can automatically solve many goals over real numbers. For example, a goal of $a + b < 2 * b + 2 * a$, where a and b are positive real numbers, can be trivially solved by lra, but this tactic does not work over generalized numeric types, so a one line proof over real numbers is several lines over generalized numeric types.

Fortunately, the tactic ring does exist. Proving that multiplication and addition of numerics is commutative, associative and distributive as well as showing that 0 the additive identity and 1 is the multiplicative identity allows us to use the ring tactic. The ring tactic can automatically solve some goals that two numeric expressions are equivalent, but unfortunately it does not solve goals about inequalities.

Because many of my proofs are about estimations and limits, most of the difficult goals are over inequalities, making the ring tactic of limited use. Despite these limitations, the ring tactic is still useful. In the previous example, $a + b < 2 * b + 2 * a$, ring cannot directly prove this goal unlike lra, but it can assist in some of the steps for proving this goal. For example, ring will automatically solve $2 * b + 2 * a = 2 * (a + b)$.

6 Conclusions

This thesis successfully provides a verified implementation of value iteration and policy iteration. The main proofs of this thesis can be split into two categories: proofs where policy iteration and value iteration have converged and proofs where they approach convergence. The proofs that do not assume convergence are useful because implementations of these algorithms may not run to convergence, but convergence must be assumed to prove the result is exactly correct. In this work, I prove these algorithms provide the optimal policy and correct value of the optimal policy when convergence is assumed, and I also prove the algorithms' approximate correctness and optimality without convergence.

Along with proofs about policy iteration and value iteration, this thesis also provides a Haskell implementation of value iteration which was directly extracted from the verified Coq implementation, and, assuming the extraction process does not introduce bugs, this implementation fulfills all of the properties proven about the implementation in Coq. Although the Haskell extraction of value iteration is slow, it is able to run small MDPs fast enough to converge to a small error within a few seconds, as shown in figure 4.2.

A major focus of this thesis is defining MDPs and these algorithms in a way that can be extended to other implementations and data types. Generalizing these algorithms over Numeric types allows using this algorithms with faster or more complex types. The implementation of Banach's fixed point theorem would simplify the process of proving correctness of any variation of these algorithms that also relies on contraction functions. This thesis lays the groundwork for projects with a much larger scope.

6.1 Further work

6.1.1 Convergence of Policy Iteration Without Convergence of Evaluation

If I had more time, I would have proven that policy iteration converges to a near optimal policy within a finite number of steps. Below is an outline of how the proof would work [PP03]

1. Prove $E^*[\pi] \leq E^*[policy\_iteration\_step(\pi)] - \epsilon_1$ where ϵ_1 is some small number with a bound derived from the number of evaluation steps done in each step in policy iteration

2. Prove that there exist some ϵ_2 such that
$|E^*[\pi] - E^*[policy\_iteration\_step(\pi)| < \epsilon_2 \rightarrow \pi$ is optimal.

3. Show that if policy iteration was run with a sufficient number of evaluation steps such that $\epsilon_1 < \epsilon_2$, the resulting policy must be strictly better than the original policy, or the original policy is already optimal.

6.1.2 Optimizations

There are a number of optimizations that can be made in the tabular value iteration. Here is a list of some of the major changes that could create a speedup.

- Currently the transition and reward functions work by finding the nth element of a list. In a functional language like Haskell, list lookup is a $O(n)$ operation, and so these functions could be optimized using a balanced binary search tree to improve this to $O(log(n))$.

- In the tabular value iteration step function my implementation does a lookup in the value table for each state. This could be optimized to use a map operation instead, but this would require major changes to my implementation and proofs.

- Replacing Coq nats with Haskell integers would also likely improve the performance. Coq nats are unary and use $O(n)$ space and $O(n)$ complexity for addition in terms of the magnitude of the nat, but integers use $O(log(n))$ space and time for addition. This may not be as large of an issue as it appears, as the size of the nats I use scales linearly with the number of operations performed, and most operations on nats in my code are incrementing and decrementing, which is a $O(1)$ operation.

6.1.3 Approximate Methods

6.1.3.1 Learning By Exploration

One possible extension of this project is modeling and proving properties of RL methods that learn by exploration without enumerating the entire model of the environment. As the state space grows, exact solution methods such as value iteration eventually become intractable, as value iteration has a time complexity of $O(|S|^2 * |A| * n)$ where n is the number of value iteration steps. Also, value iteration and policy iteration are impossible in situations where the entire dynamics of the environment may not be known.

Exploration algorithms such as Q-learning and SARSA solve these problems by learning according to observations received from the exploring the environment. Although this project does not model these algorithms, it could be extended to provide a bound of optimality of the results of these algorithms according a proven or estimated bound of the expected value of the produced policy.

6.1.3.2 Deep Reinforcement Learning

Most modern reinforcement learning research uses deep reinforcement learning. Deep Reinforcement Learning is an extension of exploration methods that uses deep learning to generalize the learned expected value and/or policy. If my work is combined

with formalized deep learning, estimated optimality bounds could also be proven according to estimated bounds of correctness between the policy produced and the expected value of that policy.

References

[BT95] Dimitri P Bertsekas and John N Tsitsiklis. Neuro-dynamic programming: an overview. In *Proceedings of 1995 34th IEEE Conference on Decision and Control*, volume 1, pages 560–564. IEEE, 1995.

[Bur95] Dennis Burke. All circuits are busy now: The 1990 at&t long distance network collapse. *California Polytechnic State University*, 1995.

[C+07] Krzysztof Ciesielski et al. On stefan banach and some of his results. *Banach Journal of Mathematical Analysis*, 1(1):1–10, 2007.

[CDT17] The Coq Development Team. *The Coq Proof Assistant Reference Manual, version 8.7*, October 2017. URL: http://coq.inria.fr

[Dut18] Sayon Dutta. *Reinforcement Learning with TensorFlow: A beginner's guide to designing self-learning systems with TensorFlow and OpenAI Gym*. Packt Publishing Ltd, 2018.

[Höl17] Johannes Hölzl. Markov chains and markov decision processes in isabelle/hol. *Journal of Automated Reasoning*, 59(3):345–387, Oct 2017. URL: https://doi.org/10.1007/s10817-016-9401-5, doi:10.1007/s10817-016-9401-5

[HPA+18] Sean D. Holcomb, William K. Porter, Shaun V. Ault, Guifen Mao, and Jin Wang. Overview on deepmind and its alphago zero ai. In *Proceedings of the 2018 International Conference on Big Data and Education*, ICBDE '18, pages 67–71, New York, NY, USA, 2018. ACM. URL: http://doi.acm.org/10.1145/3206157.3206174, doi:10.1145/3206157.3206174

[LDK95] Michael L. Littman, Thomas L. Dean, and Leslie Pack Kaelbling. On the complexity of solving markov decision problems. pages 394–402, 1995.

[MGS17] Charlie Murphy, Patrick Gray, and Gordon Stewart. Verified perceptron convergence theorem. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2017, pages 43–50, New York, NY, USA, 2017. ACM.

[Pie07] Benjamin C. Pierce. *Software Foundations: Logical Foundations*. First edition, 2007. URL: https://softwarefoundations.cis.upenn.edu/lf-current/index.html

[PP03] Theodore J Perkins and Doina Precup. A convergent form of approximate policy iteration. In *Advances in neural information processing systems*, pages 1627–1634, 2003.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html

[VSP+20] Koundinya Vajjha, Avraham Shinnar, Vasily Pestun, Barry Trager, and Nathan Fulton. Certrl: Formalizing convergence proofs for value and policy iteration in coq. *arXiv preprint arXiv:2009.11403*, 2020.

APPENDIX: COQ DEFINITIONS

$value\_func\_policy(V, s) := max_a[V\_to\_Q(V, s, a)]$

```
Definition V_to_Q (v : value_func) (s : St) (a : A) : Nt ≜
    big_sum s_enum (λ  s' ⇒ (trans s a s') * (reward s a s' + discount * (v s'))).

Definition value_func_policy (v: value_func) : policy ≜
    (λ s ⇒ argmax_ne (V_to_Q v s) a_enum_nonempty).

Definition value_iteration_step (v : value_func) : value_func ≜
  (λ (s : St) ⇒
    mapmax_ne (λ a ⇒ V_to_Q v s a) a_enum_nonempty
  ).

Fixpoint value_iteration_rec (v : value_func) (n : nat)≜
  match n with
  | O ⇒ v
  | S n' ⇒ value_iteration_step (value_iteration_rec v n')
  end.

Definition evaluate_policy_step (pol : policy) (v : value_func) : value_func ≜
    (λ s ⇒ V_to_Q v s (pol s)).

Fixpoint evaluate_policy_rec (pol : policy) (v : value_func) (n : nat)≜
  match n with
  | O ⇒ v
```

```
  | S n' ⇒ evaluate_policy_step pol (evaluate_policy_rec pol v n')
  end.

Definition policy_iteration_step (n : nat) (pv : policy * value_func) : policy * value_func ≜
  let (p,v) ≜ pv in
    (value_func_policy (evaluate_policy_rec p v n), evaluate_policy_rec p v n).

Definition policy_iteration_rec (p : policy) (v : value_func) (eval_n n: nat) : (policy * value_func) ≜
      iter n (policy_iteration_step eval_n ) (p,v).

Definition value_diff (v1 v2 : value_func) : value_func ≜
      (λ s ⇒ v1 s + − v2 s).

Definition value_dist (v1 v2 : value_func) : Nt ≜
      mapmax_ne (λ s ⇒ Numerics.abs ((value_diff v1 v2) s) ) s_enum_nonempty .
```

Thesis and Dissertation Services