QEDCartographer: Automating Formal Verification Using Reward-Free Reinforcement Learning

Alex Sanchez-Stern
University of Massachusetts
Amherst, MA, USA
sanchezstern@cs.umass.edu

Abhishek Varghese
University of Massachusetts
Amherst, MA, USA
avarghese@cs.umass.edu

Zhanna Kaufman
University of Massachusetts
Amherst, MA, USA
zhannakaufma@umass.edu

Dylan Zhang
University of Illinois Urbana-Champaign
Champaign, IL, USA
shizhuo2@illinois.edu

Talia Ringer
University of Illinois Urbana-Champaign
Champaign, IL, USA
tringer@illinois.edu

Yuriy Brun
University of Massachusetts
Amherst, MA, USA
brun@cs.umass.edu

***Abstract*—Formal verification is a promising method for producing reliable software, but the difficulty of manually writing verification proofs severely limits its utility in practice. Recent methods have automated some proof synthesis by guiding a search through the proof space using a theorem prover. Unfortunately, the theorem prover provides only the crudest estimate of progress, resulting in effectively undirected search. To address this problem, we create QEDCartographer, an automated proof-synthesis tool that combines supervised and reinforcement learning to more effectively explore the proof space. QEDCartographer incorporates the proofs' branching structure, enabling reward-free search and overcoming the sparse reward problem inherent to formal verification. We evaluate QEDCartographer using the CoqGym benchmark of 68.5K theorems from 124 open-source Coq projects. QEDCartographer fully automatically proves 21.4% of the test-set theorems. Previous search-based proof-synthesis tools Tok, Tac, ASTactic, Passport, and Proverbot9001, which rely only on supervised learning, prove 9.6%, 9.8%, 10.9%, 12.5%, and 19.8%, respectively. Diva, which combines 62 tools, proves 19.2%. Comparing to the most effective prior tool, Proverbot9001, QEDCartographer produces 26% shorter proofs 27% faster, on average over the theorems both tools prove. Together, QEDCartographer and non-learning-based CoqHammer prove 31.8% of the theorems, while CoqHammer alone proves 26.6%. Our work demonstrates that reinforcement learning is a fruitful research direction for improving proof-synthesis tools' search mechanisms.**

***Index Terms*—formal verification, proof assistants, proof synthesis, reinforcement learning**

I. Introduction

Operational software failures cost the US economy $1.81 trillion per year [39]. One method for improving software quality is formal verification, in which an engineer mathematically proves that a program will operate as specified. A common method for formal verification is through the use of proof assistants, such as Coq [78] and HOL4 [72]. Such verification is highly effective at reducing bugs. For example, a study [90] of C compilers found that every tested compiler, including LLVM [43] and GCC [75], had bugs, except the portions of CompCert [47] formally verified in Coq. Airbus France uses CompCert to ensure safety and improve performance of its aircraft [74]. And Chrome, Android, and Firefox use formally verified cryptographic code to secure communication [17], [30].

Unfortunately, verifying software can be incredibly time consuming and require significant expertise. As one example, the proofs formally verifying CompCert are 8 times longer than the code for the compiler itself [46].

Machine-learning-based approaches can automatically synthesize proofs [18], [19], [60], [68], [69], [88] using a predictive model, learned from existing proofs, to guide a metaheuristic search through the space of possible proofs. However, these tools can only verify just over one fifth of the properties of a large benchmark [18] because they lack a way to assess proof progress or direct the search towards more promising paths. The central goal of this paper is to improve the search mechanisms used to synthesize formal verification proofs.

Reinforcement learning uses continuously gathered experience, and can, in theory, automatically learn a smarter way to search through the proof space, prioritizing paths more likely to lead to a successful proof. Unfortunately, the proof space is ill-suited for direct reinforcement learning application because it suffers from the sparse rewards problem: only the final proofs can be measured as successes or failures. Prior work [86] used hand-crafted intermediary rewards, known as "reward shaping," but this can prevent learning target behavior, often causing the model to repeatedly trigger the added rewards [57], [64]. Significantly reducing the expressivity of the model can help, but every such restriction decreases the number of properties that can be proven automatically [86].

In this paper, we present QEDCartographer, an automated proof-synthesis tool that improves on the state-of-the-art search strategies using a progress estimation function learned with reinforcement learning. This function allows QEDCartographer to synthesize proofs for more theorems, and to synthesize shorter proofs more efficiently. We modify standard reinforcement learning in two key ways:

First, we generalize a standard reinforcement learning equation to work for hyperstates, which are states composed of multiple sub-states. This more faithfully models the way multiple proof goals interact: each goal can be split into multiple subgoals to be dispatched individually. This approach enables learning from progress on subproofs, addressing the sparse rewards problem while avoiding reward shaping and its pitfalls.

Second, we remove the reliance on a fixed action space, instead training a action predictor using supervised learning to provide a unique set of actions at each state. This combines the benefits of supervised learning (the ease of bootstrapping and training) and reinforcement learning (learning from both ground truth proofs and proofs found via experimental exploration).

QEDCartographer's unique contribution is improving the search mechanism, and its search strategies can potentially improve many existing tools that use undirected search [6], [18], [19], [42], [67], [69], [77], [88].

We evaluate QEDCartographer on 68.5K theorems from the CoqGym benchmark of 124 open-source Coq projects [88]. On its test set of 12K theorems, the best performing prior tool that relies purely on supervised learning, Proverbot9001 [68], proves 19.8%. QEDCartographer proves 21.4%. CoqHammer [14], an SMT-solver-based approach that applies known mathematical facts to attempt to construct a proof, is complementary to QEDCartographer: together, they automatically prove 31.8%. Compared to a version of Proverbot9001 modified with QEDCartographer's improvements, but still using its original search, QEDCartographer proves 226 more theorems, a 9% increase; together, they prove 14.5% more theorems than Proverbot9001 alone. For theorems both QEDCartographer and Proverbot9001 prove, on average, QEDCartographer produces 26% shorter proofs 27% faster.

The main contributions of our work are:

- An algorithm for overcoming the sparse reward problem in the proof synthesis domain by learning proof state values adapted for the branching structure of proofs.
- A method for overcoming the infinite action space problem of proofs by combining models trained using supervised learning and reinforcement learning.
- An implementation of multiple novel proof search strategies to make use of the state evaluation function learned via reinforcement learning.
- A reification of these advances in QEDCartographer, and an evaluation on the CoqGym benchmark of real-world theorems from open-source projects [88].

The rest of the paper is structured as follows. Section II lays out the necessary background and context for our work. Section III describes QEDCartographer and Section IV evaluates it on real-world data. Section V places our work in the context of related research and Section VI summarizes our contributions.

II. Formal Verification and Machine Learning

Before describing QEDCartographer, we first overview the background necessary to understand this paper.

A. *Theorem Proving in Coq*

Our work focuses on the task of synthesizing proofs for the Coq proof assistant. Coq is a formal proof management system that includes a formal specification language able to describe executable algorithms, mathematical definitions, and proofs. QEDCartographer interfaces with Coq through an interface similar to the one Coq users use.

Coq users write program and datatype definitions in an OCaml-like language, and then proceed to write logical statements (specifications) about those programs they would like to prove. At the beginning of the proof, the user is presented with a single goal, also known as a proof *obligation*. The obligation takes the form of a logical statement, which is also a type in dependent type theory. The user then writes a *proof script*, which we call a proof, which invokes a series of commands, called *tactics*. Each tactic takes the current obligation and either proves it or manipulate it to produce one or more new obligations. The set of obligations at any time is called a "proof state." Once there are no more obligations left to prove, the proof runs the `Qed` command, and the kernel machine checks the proof's correctness.

B. *Machine-Learning-Based Proof Synthesis Tools*

QEDCartographer joins the growing body of tools that use machine learning to prove theorems. Like QEDCartographer, these tools employ models, learned using supervised learning, that predict the next tactics likely to guide a proof to completion. Guided by these models, the tools apply various search strategies to explore the proof space. Proverbot9001 [68], for instance, uses a weighted depth-first search (DFS), and employs search-tree pruning. Section V will discuss search-based and other proof synthesis approaches.

There are (infinitely) many ways prove each (true) theorem, and training and evaluation datasets [88] these tools use typically include a single, human-written proof, which is not canonical in any sense. Tools sometimes find shorter or longer proofs than the ones written by humans [69], and may use alternative yet effective proof approaches.

C. *Reinforcement Learning*

QEDCartographer builds on top of prior work on proof synthesis by adding reinforcement learning in improving the search strategy. Reinforcement learning is a form of machine learning that trains an *agent* by interacting with an environment and encountering rewards upon the completion of certain tasks. These rewards are used to shape the parameters of the agent in developing a policy for which actions to take in a given state.

State value functions (also known as V-value functions) can be learned in several ways, including using the Bellman equation, a method of determining the value of a decision problem based on intermittent payoffs (rewards). (Section III-B3 will describe one of QEDCartographer's key contributions, the generalization of the Bellman equation.) For a reinforcement

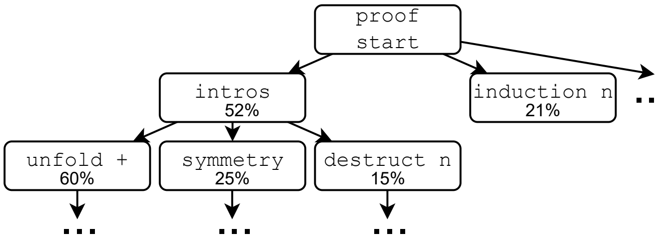

Fig. 1: Proverbot9001 uses a probability-driven depth-first, depth-limited search. A tactic predictor estimates the probability of each tactic (shown as %) and Proverbot9001 explores higher-probability nodes first. Here, a time limit prevents this search from finding a proof for `Theorem add_0_r : forall n:nat, n + 0 = n`.

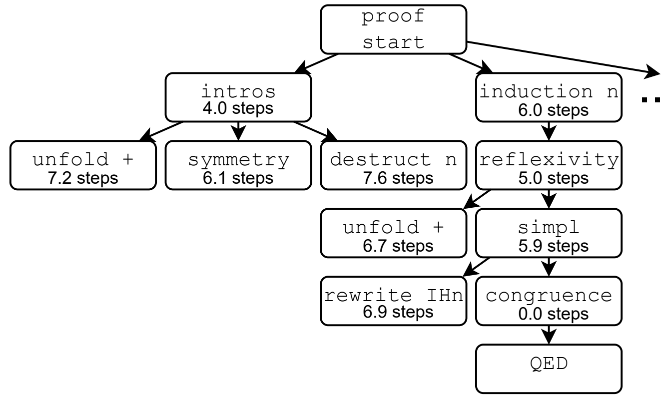

Fig. 2: QEDCartographer guides its search using a state estimator function, which estimates how close each search-tree node is to a complete proof (shown as number of steps remaining). Exploring the nodes with lower, more promising estimates first leads to a successful proof for `Theorem add_0_r : forall n:nat, n + 0 = n`.

learning problem, the Bellman equation for a fixed policy π, describing the value of a state, is:

$$V^{\pi}(s) = R(s, \pi(s)) + \gamma \sum_{s'} P(s'|s, \pi(s)) V^{\pi}(s') \quad (1)$$

where $V^{\pi}(s)$ is the value of state s under policy π, $R(s, \pi(s))$ is the reward of following the policy π in state s, γ is the rate at which future rewards are discounted over current ones, $P(s'|s, \pi(s))$ is the likelihood of ending up in state s' after following the policy π in state s, and $V^{\pi}(s')$ is the recursive value of state s'. The Bellman optimality equation describes the value of a state under the optimal policy, $\pi*$:

$$V^{\pi*}(s) = \max_a \left\{ R(s, a) + \gamma \sum_{s'} P(s'|s, a) V^{\pi*}(s') \right\} \quad (2)$$

Where following the fixed policy π is replaced by taking the action at each step that maximizes the V-value. In deterministic environments, such as theorem proving, the equation can be simplified to:

$$V^{\pi*}(s) = \max_a \{R(s, a) + \gamma V^{\pi*}(s')\} \quad (3)$$

III. QEDCartographer

QEDCartographer, our proof synthesis approach, combines reinforcement learning, supervised learning, and search. Its core functionality is a state-evaluation agent trained using reinforcement learning to guide search and determine which search paths to explore. This agent uses a tactic-prediction model, trained using supervised learning to constrain the agent's action space to the most promising tactics. The tactic-prediction model, used by prior tools [6], [18], [19], [42], [67]–[69], [77], [88], is not a contribution of our work, and thus this section focuses on the state evaluation agent and how it integrates with the supervised tactic-predictor and with proof search.

Section III-A illustrates how QEDCartographer improves on the state of the art.

A. Illustrative example

Suppose a proof search tool is trying to prove the theorem:

```
Theorem add_0_r : forall n:nat, n + 0 = n
```

Prior tools use a tactic predictor to predict the first most likely tactic based on supervised learning on existing proofs, and then perform search through the proof space using these predictions [18], [19], [68], [69], [88]. Figure 1 shows how one such prior tool, Proverbot9001 [68], explores the proof space using a probability-driven depth-first, depth-limited search, with a time limit. Here, the tactic predictor predicts `intros` with probability 52%, `induction n` with probability 21%, and several other lower-probability tactics indicated with "…" in Figure 1. (The predictor first predicts the tactic, e.g., `induction`, then computes all possible in-scope arguments, and then predicts the most likely argument, e.g., `n`.) Proverbot9001 picks the most likely `intros` and asks the tactic predictor for the next most likely tactic. The predictor predicts `unfold +` with probability 60%, `symmetry` with probability 25%, and `destruct n` with probability 15%. Proverbot9001 selects `unfold +` and continues growing this proof, using depth-first search, and checking each partial proof using the theorem prover for whether it proves all obligations or results in an error.

Proverbot9001's search is depth limited. If exploring to the limit's depth does not produce a complete proof, Proverbot9001 backtracks to earlier states and explores the next-highest-probability prediction. The search continues until either a proof proves all obligations, or a time limit is reached. In Figure 1, Proverbot9001 never finds the correct proof because for the proof to succeed, it must do induction on `n`. But the tactic predictor never predicts `induction n` after `intros n` because such a combination is very rare in its training data, and Proverbot9001 runs out of time before backtracking all the way to the start of the proof. (Our example uses a reduced time limit to illustrate the shortcoming of depth-first search.)

If Proverbot9001 were to use breadth-first search instead, it would explore `induction n` early in the search, but would then require fully exploring the search tree to a depth of 4 tactics. This would similarly exhaust the (reduced) time budget as the explored steps grow exponentially in tree depth.

By contrast, QEDCartographer uses a learned state evaluation function to guide its search, as shown in Figure 2. Using the same tactic predictor as Proverbot9001, QEDCartographer starts by predicting `intros` and `induction n` as the highest probability tactics. QEDCartographer uses the theorem prover to check the predictions for errors and to compute the proof states that result from executing each tactic. Next, QEDCartographer uses its state evaluation function to estimate the number of steps required to prove the unproved goals in those proof states. This function predicts that the proof that uses `intros` will require 4.0 more steps, whereas using `induction n` will require 6.0, and thus selects (just as Proverbot9001 did) `intros`. However, after predicting the next tactics, computing the proof states, and estimating the steps remaining for each, QEDCartographer's state evaluation function estimates that `unfold +` will require 7.2 more steps, `symmetry` 6.1, and `destruct n` 7.6. These search paths now appear less promising than the `induction n` path, and so QEDCartographer will expand `induction n` next. While the state evaluation predictions for the next few tactics are not monotonically decreasing — `reflexivity` predicts 5.0 additional steps, `simpl` 5.9, etc. — they are each more promising than the alternatives, and so QEDCartographer explores each of these states, until finding that `congruence` proves all goals, completing the proof.

QEDCartographer finds the proof of the theorem exploring a fraction of the states the prior approaches explored. For simplicity, this example did not differentiate between QEDCartographer's two search-strategies, best-first and A*, but Section III-B4 will detail their differences.

Next, Section III-B will detail how QEDCartographer's state evaluator computes proof state values and is used in search, and Section III-C will describe how QEDCartographer uses theorem statements and data to train the estimator.

B. State Evaluation Model Architecture

QEDCartographer uses reinforcement learning to train an agent that estimates the difficulty of proof states (how hard is it to finish the proof from this state), also known as a V-value model. Figure 3 shows how QEDCartographer synthesizes proofs using its reinforcement-learning-trained state value model. Given a theorem, QEDCartographer uses a search strategy, such as A* or best-first (described in Section III-B4) to navigate the search through the proof space. The search strategy uses the tactic predictor to produce potential actions — next steps of the proof — and then uses the interactive theorem prover to compute the proof contexts that result from executing each of these actions. These proof contexts are hyperstates, consisting of several obligations that need to be proven. Code2Vec (described in Section III-B2) encodes each obligation in these hyperstates and the state value model, trained using reinforcement learning, then estimates each obligation's difficulty. The hyperstate value aggregator combines these estimates into a single estimate of the difficulty of the entire hyperstate (in our case, multiplying the values, as described

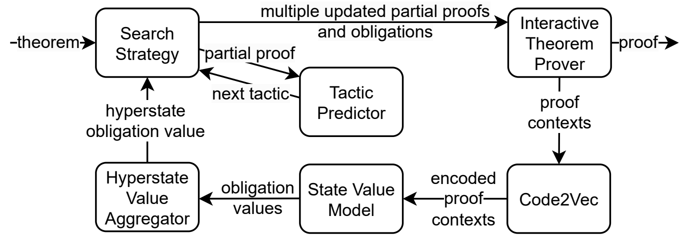

Fig. 3: Given a theorem, QEDCartographer attempts to synthesize its proof. A search strategy uses the tactic predictor to produce potential actions. For each action, the interactive theorem prover computes obligations that need to be proven, and Code2Vec and the state value model estimate the difficulty of each obligation, which the hyperstate value aggregator combines into a single estimate per action (in our case, by multiplying the estimates together). The search uses these estimates to determine which action to take. A partial proof that proves all obligations proves the original theorem.

in Section III-B3). The search strategy then determines which hyperstate should be expanded next. If the interactive theorem prover determines that a proof leads to a context with no remaining obligations, that proof proves the original theorem.

QEDCartographer has four components that overcome key challenges of using reinforcement learning in a proof search context:

- First, to build a tractable action space for exploration, the state agent builds on a supervised tactic predictor that produces a small set of viable tactics at each state.
- Second, to provide a state encoding that captures the complexity of the proof space the state agent uses a text sequence model trained using autoencoding.
- Third, to smooth the sparse rewards inherent to theorem proving, QEDCartographer's state agent models each proof state as a hyperstate, built of multiple obligations.
- Fourth, to use the learned state-evaluation agent to guide proof search, QEDCartographer implements two new search methods for proof synthesis, best-first and A* search.

Next, we describe each of these components in more detail.

1) Building on a supervised predictor: An important part of the design of a reinforcement learning system is defining the action space. In QEDCartographer's theorem proving task, actions correspond to adding a tactic to the proof. Unfortunately, the full space of tactics is infinite. There are over 300 built-in tactics in Coq, and many of them can take as arguments arbitrarily large terms in the underlying logical calculus.

QEDCartographer addresses this issue by combining its reinforcement learning agent with a tactic predictor trained using supervised learning on existing human-written proofs. Instead of asking the agent to pick from the set of all possible actions, or even a fixed finite actions space, QEDCartographer first asks the supervised tactic predictor to predict the most likely N actions over the infinite action space (where N is a hyperparameter). In the initial state of the example from Figure 1, the supervised tactic predictor predicts `intros` and `induction`, among others, and then runs each prediction to get the resulting state for evaluation.

By building its reinforcement agent on top of a tactic predictor trained using supervised learning, QEDCartographer makes the action space tractable and reduces the number of Coq queries run.

2) State encoding using Coq2Vec autoencoder: To allow for deep reinforcement learning, QEDCartographer embeds proof states into a vector space where the model can learn to estimate the value of each state. However, proof contexts consist of lists of sequences of characters, constituting a discrete infinite state space, making the application of reinforcement learning particularly challenging, since a mapping from a state to its values is difficult to learn. Therefore, QEDCartographer needs to find a one-to-one mapping from the discrete infinite space to a continuous real-space, or alternatively speaking, to learn continuous embeddings for each state.

Our approach to overcoming this challenge involves the use of an autoencoder, a deep representation learning approach that is trained in a self-supervised manner with the objective of first compressing the input into a latent space representation and then reconstructing the input from that representation. The strength of autoencoders lies in their ability to retain essential information of the sequence while eliminating statistical redundancies. This is facilitated through their two core components: the encoder and decoder. The encoder is responsible for compressing the input into a latent space, while the decoder is tasked with reconstructing the input from the latent space representation. QEDCartographer adopts a symmetrical LSTM model for both parts.

3) Value estimation using obligation sets: One of the most difficult parts of designing robust reinforcement learning algorithms is deciding how to structure rewards. In theorem proving, the natural reward structure is sparse. Since the goal of the theorem proving task is fully proving a theorem, the most natural reward structure assigns a positive reward to reaching `Qed`, and a zero reward to all other actions. This sort of structure only rewards the agent once it has come upon a long sequence of correct actions, resulting in proof completion. In practice the agent simply does not get rewards and never updates its policy, preventing learning [61].

The challenge we face is how to design a reward structure that allows for intermediary rewards to guide an agent to the overall goal, without creating local minima for the agent to get stuck in. One often-used approach to handling sparse rewards is to smooth the reward space by inserting intermediary rewards [57], [64], known as "reward shaping." For instance, in a video game context, an agent might be rewarded for picking up a coin or defeating an enemy, in addition to its reward for completing a level. In the theorem proving context, since simpler goals are often easier to prove, the agent can be rewarded for making the goal simpler. Alternatively, the agent can be punished via a negative reward for trying a tactic that results in an error.

However, if these intermediary rewards are not carefully constructed, they can lead to the agent getting stuck in a local minimum, which results in pathological behavior [57], [64]. For instance, if the agent is rewarded for making the goal simpler, it might learn to only run `exfalso`, which turns the goal to "False," significantly reducing its complexity but blocking further progress. Or, if the agent is punished for producing errors, it might learn to continually run tactics that cannot fail, such as `simpl`, making no progress but avoiding errors.

Instead, we need to find a different reward structure allows for intermediary rewards to guide an agent, without falling prey pathological behavior. A structure based on the solution of obligations accomplishes both these goals.

To accurately represent the branching structure of proofs, QEDCartographer reformulates the reinforcement learning problem to be over *hyperstates* (sets of states) instead of just states. Each hyperstate is a full proof context, where its component states are the individual obligations. In the example from Figure 1, the initial state only has one obligation, so the hyperstate is composed of a single substate. However, when `induction` is run, one obligation is produced for the base case, and another for the recursive case. The resulting hyperstate then is composed of two substates, one for each obligation. When `reflexivity` is run, it dispatches the first obligation, so the resulting hyperstate is once again composed of only one substate. In general, each tactic operates on a single obligation, but in addition to transforming that obligation into a new one, it can produce multiple new obligations, or it can finish proving the obligation.

To use this kind of transition to update the value of a state, QEDCartographer defines a total value for the set of obligations produced by a tactic. The number of steps needed to fully finish a proof from a particular state is the sum of the number needed to finish each obligation. So since the V-values are logarithmic in the number of steps needed, QEDCartographer combines obligation values by multiplying them.

In our example, if we interpret the scores presented as the estimated steps left,[1] the base case from `induction` is estimated to have a V-value of 0.9 (the value of γ, indicating 1 step remaining), while the recursive case is estimated to have a V-value of 0.59 (γ^5, indicating 5 steps remaining). Therefore, we use the product of these values, 0.531 (γ^6), as the V-value of the hyperstate that results from running `induction`, indicating it has six steps remaining.

Our equation to calculate a target V-value for an obligation, replacing the standard Bellman equation, then becomes:

$$V'(s) = \max_a \left(R(s,a) + \gamma \Pi_{s' \in f(s,a)} V\left(s'\right)\right) \tag{4}$$

This equation has several advantages. First, when state values are bounded to be within the range $[0,1]$, it prevents the agent from getting rewarded for repeatedly creating and proving obligations; the reward from proving an obligation never

[1]These scores also include the number of steps taken so far, as explained in Section III-B4; here, we present them as best-first scores for simplicity.

overcomes the cost of generating it. Second, since tactics that finish proving an obligation (like `reflexivity`) produce an empty set of resulting obligations ($f(s,a) = \emptyset$), there is no need for explicit rewards; a substate just before finishing an obligation will be given a V-value of γ, the same as if a reward of 1 had been explicitly given for obligation completion. This means explicit rewards are no longer needed in QEDCartographer's algorithm, as the equation for calculating values of transition implicitly rewards proving obligations.

Removing the explicit reward term from the above equation results in the final equation:

$$V'(s) = \max_a \left(\gamma \Pi_{s' \in f(s,a)} V(s')\right) \tag{5}$$

This revised equation allows the reward structure to locally reward an agent for proving obligations, without incentivizing the creation of spurious or trivial obligations.

4) Using the state value estimation model for search: In its main mode of operation, Proverbot9001 uses a weighted depth-first search to explore the space of possible proofs using its tactic predictor model. While this strategy is effective, it is not efficient, as the depth-first search exhaustively explores paths for every encountered proof state. QEDCartographer is able to more effectively search a proof state space by utilizing its learned state value estimation agent, as described via an example in Section III-A.

The simplest search that can make use of state value estimates is best-first search, where the best-scoring node is explored at every step. In this search strategy, any method of sorting nodes can be applied, including using the product of probabilities (or sum of log-probabilities) for each tactic in the proof, produced by the tactic predictor. However, the quality of the estimator is extremely important for effectiveness; Section IV-E1 will empirically evaluate how QEDCartographer's search improves upon a version of the search that uses information from the tactic predictor.

A* search builds on best-first search, with the goal of not just finding a solution as quickly as possible, but also finding the shortest solution. In the proof space, this empirically results in a higher success rate (see Section IV-C4); we speculate this is because shorter proofs are more likely to be correct than those that spin far off from the original goal. Figure 2 shows exploration using A* search.

In A*, instead of just sorting the partial proof queue by the state score estimate, partial proofs are sorted by an estimate of the total length of a solution found using that proof prefix, called the *f score*. This estimate is made by adding the steps taken so far (the length of the partial proof candidate) and an estimate of the steps remaining to finish the proof. In the example from Figure 2, the step estimates indicate the *total* estimated length of the completed proof, not just the remaining steps. The state resulting from the `simpl` tactic is estimated to have 2.9 steps remaining until a QED, and has already taken three steps, resulting in a total scorer of 5.9.

For A*, the state value estimate must correspond to an estimate of steps left in the proof, so not all state scoring functions can be used; in particular, the tactic prediction certainties cannot be used to produce a score, since they cannot be converted to a steps-remaining estimate. However, the state evaluation function learned by QEDCartographer *can* be converted to a steps-remaining estimate.

To obtain the estimated number of steps remaining in a proof from the state value, we first notice that for an optimal V-value evaluator of a single obligation, $V(s) = \gamma^{steps(s)}$. Therefore, the estimated steps for each obligation can be computed as $steps(s) = \log_\gamma V(s)$. To complete the proof of a theorem, we must prove all the obligations that exist at any given proof state. Hence, the total number of steps to finish a proof from a proof state P is $steps(P) = \sum_{s \in P} \log_\gamma V(s)$.

QEDCartographer uses A* search for all proof synthesis (except for when evaluating search techniques in our evaluation).

C. Training the Proof State Evaluator

QEDCartographer's agent design overcomes several key challenges of using reinforcement learning in a proof search context. However, there are still three more challenges that cannot be addressed through model architecture alone. These challenges guide how QEDCartographer is trained. First, to address the sample inefficiency of reinforcement learning and the sparsity of theorem proving data, QEDCartographer trains on the subproofs within each proof, described in Section III-C1. Second, because QEDCartographer uses a supervised tactic predictor to constrain its action space, during training, it filters out goals whose human-written proofs cannot be generated because of those constraints, as described in Section III-C2. Third, since the equations for updating the agent are recursive, QEDCartographer is pre-trained in a supervised manner to increase stability, described in Section III-C3.

In addition to addressing reward sparsity in QEDCartographer design through hyperstate modeling, QEDCartographer uses three mechanisms during training to provide rewards to the agent more quickly. These include learning by demonstration (Section III-C4), a "true target" buffer where V-values are computed directly when possible (Section III-C5), and training dead-end states to a zero V-value (Section III-C6).

Finally, to speed up training on a large number of proofs, QEDCartographer makes use of distributed training (Section III-C7).

Figure 4 shows QEDCartographer's distributed training architecture. Multiple actors, deployed on different computing nodes, each perform greedy search trying to prove different theorems from the training data. These observed proof lengths are estimates of how many proof steps each obligation requires to prove. Periodically, these actors send data about their experience to three buffers: the replay buffer, the true target buffer, and the negative state buffer. Training then uses the three buffers to update the state value model weights for estimating how many proof steps each obligation takes to prove, storing the results in the parameter server, periodically sending those updated weights to the actors.

1) Subproof task training: In reinforcement learning, a significant amount of trial-and-error is essential for an agent to

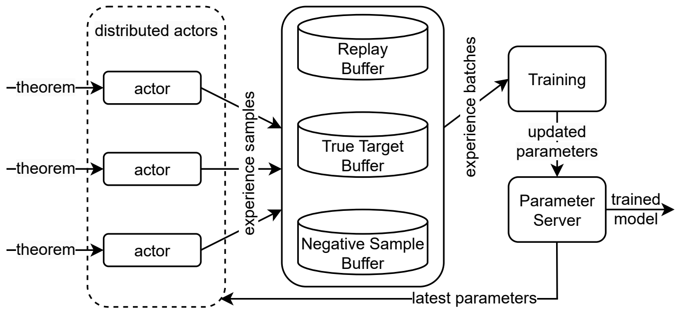

Fig. 4: QEDCartographer uses a distributed training architecture to train its state value model. Multiple actors each perform greedy search on different theorems from the training data, periodically sending data about their experience to three buffers. Training uses these buffers to update the model weights in the parameter server, periodically sending those updated weights to the actors.

learn to achieve the goal successfully, a phenomena known as "sample inefficiency."' Unfortunately, training on a large number of samples is difficult in the theorem proving environment, because of the computational of environment interaction and the limited training data available.

To address these challenges, QEDCartographer trains on not only every proof in our training set, but every obligation that is created by the human-written solution to that proof. Since proving a theorem often involves creating many independently provable obligations, there are many small proofs within each large proof. For instance, if the example from Figure 2 were in the training data, QEDCartographer could train using either of the subgoals resulting from the `induction` tactic; since these subgoals are independently solvable, they need not be trained on together. QEDCartographer takes advantage of the fact that there are human solutions to the proofs in its training data to automatically expand its training set with these subproofs.

2) Task filtering: Building QEDCartographer on top of a tactic predictor model trained using supervised learning allows it to simultaneously consider a large set of tactics, and limit its possible actions at any given point in a proof to a much smaller, most-probable set of tactics. In practice, however, the correct tactic (or one of the correct tactics) is not always in the top predictions of the tactic predictor.

Since QEDCartographer has human-written proofs for the theorems in its training set, it checks if, at every step of the human-written proof, the supervised tactic predictor has a matching tactic in its top predictions. It then filters out during training those proofs it will not be able to generate. (Importantly, we do not filter such proofs from the test set in our evaluations in Section IV; we only filter them from the training set.)

3) Supervised pre-training: To bootstrap our model, QEDCartographer begins by pre-training in a supervised manner. For each obligation in our training set, the length of the ground-truth solution to that obligation can be taken from the human-written proofs. To compute a ground-truth state value, QEDCartographer only needs to raise the value of the γ parameter to the power of the number of tactics in the solution. QEDCartographer uses the Optuna hyperparameter tuning framework [2] to tune the hyperparameters used during pre-training.

4) Learning by demonstrations: To train more quickly and accurately, QEDCartographer uses *learning by demonstration* [70]. In learning by demonstration, the RL agent is not just dropped at the beginning of each task, but instead slowly introduced to the task. The agent is initially guided through every step of the task until the last one, and then tasked with completing the last step and given a reward. Then, the process is repeated but stopping at the second-to-last step, and then again at the third-to-last task. This process is repeated until the agent is solving the full task.

If the example from Figure 2 were in the training data, this would mean that before exploring the full task, the agent would first be guided through `induction n. reflexivity. simpl.`, and then given a reward for predicting `congruence`. It would then restart the proof, and be guided through `induction n. reflexivity.`, and given a reward if it could predict `simpl. congruence`.

This process greatly speeds up convergence in environments with sparse rewards like theorem proving. However, it does require access to labeled training data, which are not necessary when training without demonstrations or subproofs (Section III-C1).

5) True target buffer: To train the state value estimation model, QEDCartographer primarily uses state-transition updates based on the revised Bellman equation from Section III-B3. But this type of training can be unstable because the target estimates can change based on the current weights of the estimator (which itself is in the process of being trained). Stability can be increased by, along with sampling the target from estimates of the next state, also sampling the current ground truth target, if it exists. QEDCartographer does this by using a separate buffer that stores each obligation and the minimum number of steps it took the developer or the reinforcement learning agent to prove it. When the training begins, this buffer is filled with the initial states of every obligation along with the developer's solution length. During training, if our reinforcement learning agent can prove a given obligation in fewer steps than the developer, this buffer is updated with the lower target for that obligation. These new values are included in subsequent training steps by sampling part of every training batch from this buffer. This helps make training more stable and quicker to converge to a solution.

6) Negative examples: During reinforcement learning exploration, QEDCartographer can encounter obligations for which all of the tactics produced by the tactic predictor produce an error. These obligations can never be proven by an estimator built on that tactic predictor. Therefore, we use them as a negative example to our agent and train their associated states to a target value of 0.

7) Distributed training: Section II-C discussed how in theorem proving environments, the proof state after each tactic can always be determined via computation, but that computation can be intensive. This means that rolling out the current policy in different proof obligations quickly becomes the computational bottleneck of training.

To address this challenge, QEDCartographer trains using a distributed reinforcement learning architecture, inspired by the Gorila [56] architecture (see Figure 4). In Gorila, learning and acting are similarly resource intensive, so actors and learners are bundled together, communicating through a shared parameter server. Since acting is much more expensive in the theorem proving environment, QEDCartographer instead bundles the learner and the parameter server, which communicate with each of the distributed actors.

IV. Evaluation

We evaluate QEDCartographer's ability to synthesize proofs, the effectiveness of its components, and the effect of its hyperparameters. Sections IV-A and IV-B describe our methodology. Then, Section IV-C compares QEDCartographer to the state-of-the-art proof synthesis tools, including ASTactic [88], CoqHammer [14], Diva [18], Passport [69], Proverbot9001 [68], and Tok and Tac [19]. Section IV-D directly evaluates the state value agent's effectiveness, and Section IV-E performs an ablation study to understand the impact of search strategies, individual obligation training, and hyperparameters.

We first summarize our main findings:

> QEDCartographer automatically proves 21.4% of the theorems in a large benchmark. Prior supervised learning tools, Tok [19], Tac [19], ASTactic [88], Passport [69], and Proverbot9001 [68], prove 9.6%, 9.8%, 10.9%, 12.5%, and 19.8%, respectively. Comparing to a version of Proverbot9001 modified with QEDCartographer's improvements, but still using its original search, QEDCartographer proves 11% more theorems with the same time budget, and, together, they prove 15% more theorems than modified Proverbot9001 alone. When both QEDCartographer and Proverbot9001 prove a theorem, QEDCartographer produces 26% shorter proofs 27% faster. Meanwhile, combining QEDCartographer and CoqHammer, an SMT-solver-based approach that applies known mathematical facts to attempt to construct a proof, automatically proves 33.7% of the theorems.

A. Research Questions

Our evaluation answers three research questions:

- RQ1 How effective is QEDCartographer at proving theorems?
- RQ2 How effective is QEDCartographer's state-estimation agent at proving proof sub-problems directly in a simple greedy search?
- RQ3 How do learning by demonstration, task filtering, and sub-proof training impact QEDCartographer's effectiveness?

B. Benchmarks

We use the CoqGym benchmark [88] of 124 open-source Coq projects with 68,501 theorems. We exclude one project, coq-library-undecidability; prior evaluations similarly were unable to use this project due to internal Coq errors when processing its proof scripts [18], [19], [69]. Projects in the CoqGym benchmark are a mixture of mathematical formalizations, proven correct programs, and Coq automation libraries. They include several compilers of varying sizes (such as CompCert [47]), distributed systems (such as Verdi [85]), formalizations of set theory, and more. Some of the projects in CoqGym (such as the automation libraries) contain no proofs, but we include them for completeness, as did prior work.

As in prior work, our evaluation uses 97 projects for training, which contain a total of 57,719 theorems, and the remaining 26 projects, which contain a total of 12,161 theorems, to measure the effectiveness of our approach. We further split the 57,719 training theorems into obligations during training (recall Section III-C1). We filter out the obligations whose original solutions would not be reproduced with a perfect state evaluator (recall Section III-C2). We perform some additional filtering by removing obligations whose proofs are only 1 or 2 steps, as we found those did not help training, as well as obligations whose proofs were 6 or more steps to decrease training time. This resulted in a training dataset of 15,005 obligations. (As mentioned earlier, this filtering did not affect the 10,782 theorems on which we measured performance.)

For our fine-grained obligation experiments and ablation studies, we use the CompCert C compiler [47] as a benchmark, as it is also used in the original Proverbot9001 evaluation [68]. We use the training-test split used by the Proverbot9001 evaluation [68]; the training set consists of 162 proof files and the test set consists of 13 proof files of 507 theorems.

C. RQ1: Comparison to the State of the Art

Recall that QEDCartographer's reinforcement-learning-driven search can be applied to all existing search-based proof synthesis tools. To demonstrate QEDCartographer's contribution to the state of the art, we compare QEDCartographer's search strategy with the strategy used by Proverbot9001, the most effective current neural network-based proof synthesis tool. We show that applying QEDCartographer's search strategy to Proverbot9001 proves more theorems with the same time budget. We further show that QEDCartographer finds shorter proofs than Proverbot9001, and that it finds proofs faster. For completeness, we also include Proverbot9001 without a time limit (its default mode of operation) in Figures 5 and 6.

1) Proof synthesis effectiveness: QEDCartographer combines supervised and reinforcement learning to synthesize proofs. On the CoqGym benchmark, QEDCartographer proves 2,445 theorems, or 21.4% (see Figure 5).

Prior tools that relied only on supervised learning and search prove fewer theorems: Tok [19] proves 9.6%, Tac [19] 9.8%, ASTactic [88] 10.9%, and Passport [69] 12.5%. Diva [18], a combination of 62 different tools, each using a different tactic-prediction model, proves 19.2% of the test set, though

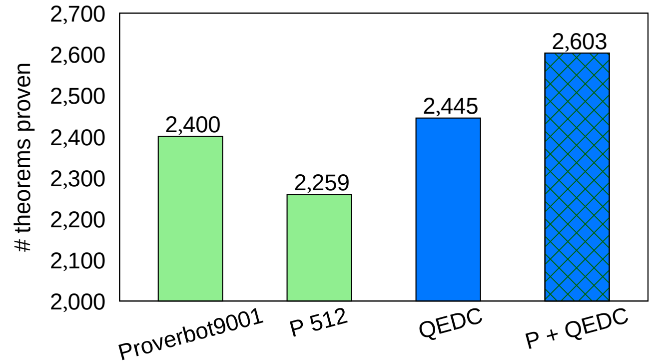

Fig. 5: QEDCartographer (QEDC) proves more theorems than Proverbot9001 and Proverbot9001 limited to 512 steps (P 512). QEDCartographer is limited to 512 steps by default. Together, Proverbot9001 and QEDCartographer (P + QEDC) prove 2,603 theorems.

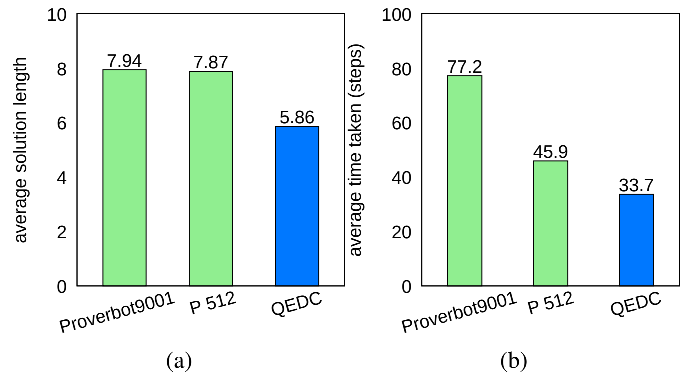

Fig. 6: (a) QEDCartographer (QEDC) synthesizes, on average, shorter proofs than Proverbot9001 and Proverbot9001 limited to 512 steps (P 512). (b) QEDCartographer synthesizes proofs taking fewer search steps, on average, than Proverbot9001 and Proverbot9001 limited to 512 steps. For fairness, we only compare on theorems for which both tools were able to synthesize a proof.

comparing to QEDCartographer running with a single tactic-prediction model is unfair. (Note that the framework all these prior tools used fails to parse 1,379 theorems (about 11%) of the CoqGym benchmark test set. We include these theorems as ones these tools fail to prove; whereas the tools' original evaluations excluded them.) CoqHammer [14] is an SMT-solver-based approach that applies known mathematical facts to attempt to construct a proof. Solver-based tools work well at fully automating simple proofs but are generally limited to first-order problems, while neural tools can potentially be more general. CoqHammer, for example, cannot use induction. CoqHammer proves 23.7% of the benchmark, but is complementary to QEDCartographer: together, they automatically prove 31.8% of the theorems. (This number is likely an underestimate because we used previously published CoqHammer results [88], which also suffers from the inability to parse 1,379 theorems in the test set.)

Finally, Proverbot9001 proves 19.8% of the CoqGym test set, making it the most effective neural network-based proof synthesis tool. For this reason, we used Proverbot9001's architecture for toolname's supervised predictor. Figure 5 shows the number of theorems proven by Proverbot9001, Proverbot9001 limited to 512 steps (P 512), QEDCartographer (QEDC, which is limited to 512 steps by default), and a combination of Proverbot9001 and QEDCartographer. QEDCartographer proves 186 more theorems than Proverbot9001 with the same time budget, an increase of 11%.

The two tools are somewhat complementary: There are 271 theorems QEDCartographer proves that Proverbot9001 cannot with the same time budget. Thus, combining the two tools proves 15% more theorems than Proverbot9001 alone [19]. Meanwhile, Proverbot9001 proves 87 theorems that QEDCartographer does not.

We conclude that QEDCartographer's new search and state value estimation are able to guide search towards proving theorems that would not otherwise be provable.

2) Proof length: Figure 6a shows the average proof length for the in-common proven theorems for Proverbot9001 and QEDCartographer. The average proof length of proofs generated using Proverbot9001's depth-first search is 7.88 tactics, while for QEDCartographer's A* search proofs, it is 5.86 tactics — a 26% reduction in proof length.

Of the 2,174 test theorems for which both QEDCartographer's A* search and Proverbot9001's depth-first search find a proof within the same budget, QEDCartographer finds a shorter proof for 1,477 theorems (68%), and Proverbot9001's finds a shorter solution for 35 theorems (2%). For the remaining 662 theorems (30%), the proofs were of the same length.

3) Search speed: Figure 6b shows the average number of search steps Proverbot9001 and QEDCartographer needed to find a proof of a theorem. The average number of steps Proverbot9001's depth-first search required is 45.9, compared to 33.7 for QEDCartographer's A* search — a 27% decrease.

Of the 2,174 in-common test theorems, QEDCartographer search takes fewer search steps for 1,532 theorems (71%), while Proverbot9001 takes fewer search steps for 501 theorems (23%). For the remaining 141 theorems (6%), the searches were of the equal numbers of steps.

We note that steps for A* and best-first search can be slightly slower than those for depth-first search, because those search strategies can backtrack more, but improvements in proof assistant technology are rapidly closing that gap.

4) A is better than best-first:* To compare QEDCartographer's two new search strategies, we ran QEDCartographer with A* and best-first search on the CoqGym benchmark. A* proved 148 more theorems than best-first (a 6% increase), and found 13% shorter proofs 5% faster. This demonstrates that QEDCartographer's approach to converting a state value score to an estimate of steps remaining in a proof is an effective method for improving search.

However, best-first does prove several theorems that A* search does not, indicating that the two can be combined for additional proving power. Combining the results of both searches in QEDCartographer proves 10% more theorems than Proverbot9001 with the same time budget.

D. RQ2: QEDCartographer's State Value Agent Effectiveness

We next isolate QEDCartographer's state value agent from its A* and best-first search strategies to evaluate the agent's contribution on a more fine-grained level. We replace A* and best-first with a greedy exploration strategy to directly synthesize proofs. Instead of sampling from the set of actions, greedy exploration always picks the highest-scoring action. We compare this greedy search to using just the pre-trained tactic predictor, on all proof obligations within the proofs from the CompCert test set.

Since engineers use proof assistants to manually prove theorems, synthesizing proofs for some proof obligations, in theory, directly reduces the engineers' required effort. This allows the engineers to focus on a smaller set of unproven obligations. Thus, the number and fraction of obligations a tool can automatically synthesize proofs for is an appropriate measure of the tool's effectiveness.

Since this experiment is evaluating the state value estimator directly, we filter the 5,858 obligations in the CompCert test set down to the 3,743 that can be proven by a perfect state value estimator oracle.

We compare QEDCartographer with this greedy strategy to a greedy search that uses Proverbot9001's predictions directly. We find that QEDCartographer with greedy search proves 97.6% of the obligations in our test set, compared to 92.8% that Proverbot9001 with greedy search proves. We conclude that even without QEDCartographer's backtracking search strategies, its state evaluation model can be used to effectively pick from a set of predictions better than the supervised predictor can alone, proving 4.8% more proof obligations. In this setup, QEDCartographer's state evaluator is unaware of the prediction certainties that the tactic predictor produced, reconstructing a better ordering from scratch; future work may explore ways to combine the two sources of information.

E. RQ3: Effects of Design Decisions and Hyperparameters

This section evaluates QEDCartographer's remaining components, design decisions, and hyperparameters. Section IV-E1 evaluates the impact of the search strategy alone by using best-first search without the learned state value estimates, instead using information available in the pre-trained tactic predictor. Next, Section IV-E2 evaluates the choice to train on subproof obligations by comparing to a version that trains only on entire theorems. Finally, Section IV-E3 evaluates the choices of width (the number of actions to consider at each step during training) and γ (the time-discount factor in our state value update equation). We perform these evaluations on the CompCert benchmark.

1) Guided search with and without using QEDCartographer state values: To determine the impact that the state value estimator has on the effectiveness of search, we ran best-first search using only information available to the tactic predictor. Since Proverbot9001's tactic predictor produces probabilities for each suggested action, we use the product of those probabilities along each proof path as a state-value score.

Using QEDCartographer state values as scores in best-first search increases the number of theorems proven by 22%. Searches with the two different state scores are complementary to some extent; combining the theorems proven by both produces 34% more proofs than using certainties alone, and 10% more proofs than using state scores alone.

Because A* search requires that state values be convertible to an estimate of number of steps remaining in the task, it does not make sense to perform a similar comparison using A* search.

2) Individual obligation training: QEDCartographer trains not only on proof scripts for full theorems, but also the proofs of individual proof obligations within proofs (recall Section III-C1). To measure the impact of this proof obligation training, we also train QEDCartographer without access to the proof obligations' proofs.

Without training on these obligations, greedy search using QEDCartographer can prove only 94.3% of obligations, 11 fewer than with subproof training. This shows that training on subproofs improves QEDCartographer's proving power, though the improvement is relatively small. When QEDCartographer uses A*, training without obligations' proofs proves 99 theorems, while training with obligations' proofs proves 103 theorems (an improvement of 4%).

3) Hyperparameter Analysis: During both exploration and training, QEDCartographer uses a fixed number of predictions from the tactic predictor. We call this number the search width. Increasing the width allows QEDCartographer to explore more options at each step, which can lead to more proofs, particularly shorter proofs. However, utilization of the time budget for wider exploration can reduce the chances of completing longer proofs, which require higher search depth.

We tested five width values (3, 5, 7, 9, and 11). A linear regression model found no significant relationship between width and the number of theorems QEDCartographer proves ($p = 0.46$). However, there was a negative (Pearson correlation coefficient of -0.81), weakly significant ($p < 0.1$) correlation between width and the average synthesized proof length and a high positive (Pearson correlation coefficient of 0.91), significant ($p < 0.05$) correlation between width and the average proof synthesis search steps. This suggests that higher width resulted in shorter proofs but required more search exploration, consistent with higher widths exploring more, shorter proofs first.

While not statistically significant, anecdotally, the largest width we evaluated (11) resulted in more proofs, on average (108.7) than all other widths. Further research should look into understanding how varying the width can be leveraged to improve proving power.

When computing state values, QEDCartographer discounts all future rewards by a fixed constant, γ, exponentiated by the time step difference between the current state and the reward state. Varying γ (we evaluated four γ values: 0.5, 0.7, 0.9, and 0.99), a linear regression model found no significant relationship between γ and the number of theorems QEDCartographer proves ($p = 0.31$), the average synthesized proof length ($p = 0.27$), and the average proof synthesis search steps ($p = 0.91$).

While not statistically significant, anecdotally, we did find that for $\gamma = 0.5$, QEDCartographer found shorter proofs (7.2 tactics, on average) than for $\gamma = 0.99$ (8.5 tactics, on average). For $\gamma = 0.99$, QEDCartographer synthesizes slightly more proofs (99) than for $\gamma = 0.5$ (94). For $\gamma = 0.99$, QEDCartographer synthesizes proofs more quickly (13.3 steps, on average) than for $\gamma = 0.5$ (15.0 steps, on average). More research is necessary to understand more fully γ's effect on proof synthesis and the resulting trade-offs.

V. Related Work

Our work applies modified reinforcement learning algorithms to improve performance on proof synthesis tasks. The most similar related work to ours is TacticZero [86], which uses reinforcement learning to prove theorems in HOL4. Unlike QEDCartographer, TacticZero limits the action space to nine tactics and their arguments, which loses expressivity and efficiency, and relies on hand-tuned reward shaping, which can lead to getting stuck in locally optimal policies (recall Sections I and III-B3). These hand-crafted rewards are necessary because the policy models full proof state transitions instead of obligation transitions, so that the obligation reward structure is not automatically derived from the transition equations. Further, TacticZero's evaluation is limited to only theorems with known proofs consisting of those nine tactics, while we evaluate QEDCartographer on all theorems in the benchmark.

DeepHOL [6] uses deep reinforcement learning to synthesize proofs for HOL Light. A later extension to use graph neural networks without reinforcement learning outperforms DeepHOL [60]. In a much simpler, first order tableau-calculus-based theorem prover, a sequence of results apply reinforcement learning, with emphasis on avoiding domain heuristics [38], synthesizing long proofs from little training data [95], and building a comprehensive toolkit [96]. HyperTree uses reinforcement learning for proof search but operates only in a simplified, custom-made theorem prover [40]; by contrast, QEDCartographer runs in an existing, widely-used environment. Bansal et al. [7] use reinforcement learning to learn a premise selection task for theorem proving.

By contrast, our work uses reinforcement learning alongside tactic prediction trained through supervised learning to train a neural theorem prover in a higher-order, dependently typed setting. In pursuit of this goal, our work contributes novel reinforcement learning algorithms not seen in prior work.

Reinforcement learning can synthesize code, using tests to ensure quality [91], but such approaches do not prove correctness.

Many proof-synthesis approaches do not rely on reinforcement learning. For example, hammers use solvers to iteratively apply known mathematical facts to attempt to construct proofs. CoqHammer [14], for example, uses SMT-solvers, and Sledgehammer combines multiple solvers to attempt to prove individual subgoals of a theorem [62]. Draft, Sketch, and Prove [34] uses language models to generate sketches of formal proofs from informal proofs in Isabelle/HOL, and then fill in those sketches using Sledgehammer. Thor [32] combines a language model with Sledgehammer to synthesize proofs for Isabelle/HOL. Combining a learned tactic-prediction model with a search procedure can similarly synthesize proofs, e.g., with Proverbot9001 [68], ASTactic [88], TacTok [19], Passport [69], Tactician [42], and Diva [18] for Coq, and TacticToe [22] and HOLStep [37] for other proof assistants.

Recent tools have used both small and large, transformer-based language models for tactic prediction or proof synthesis. GPT-f [63] combines a specialized pre-trained language model with proof search to synthesize proofs in Metamath. LISA [33] evaluates large language models in combination with proof search on an Isabelle/HOL dataset they introduce. Baldur [20] uses a fine-tuned mathematical language model in combination with a novel self-repairing approach to synthesize proofs for Isabelle/HOL, without using hammers or proof search. Our work can, in theory, augment these approaches as well, to guide their predictions. This paper demonstrated the benefits of QEDCartographer's advances augmenting one such tool, Proverbot9001, and future work should explore QEDCartographer's effects on other such tools.

Recent results have shown that retrieval and memory can augment or guide models for proof synthesis to great effect. Memorizing transformers [87] introduce a memory-augmented transformer model and show how they can improve performance on proof-related benchmarks for Isabelle/HOL. LeanDojo [89] uses a retrieval-augmented large language model to improve performance for proof synthesis for the Lean proof assistant. Our work provides an additional and complementary way to augment or guide models for proof synthesis.

QEDCartographer combines two different learned models to effectively explore the proof space and direct search: a tactic predictor trained using supervised learning and a proof state evaluator trained using reinforcement learning. Combining different learned models to enhance performance on a task is known as ensemble learning. Ensemble methods have become common in machine learning, as combined models can outperform individual ones. Ensemble methods improve accuracy by integrating the predictions of several models, taking advantage of diversity in training data, model architectures, or even hyperparameters.

There are many different kinds of ensemble methods that take advantage of this diversity, including *bagging*, *stacking*, and *boosting*. Ensemble methods also extend to non-traditional areas of machine learning applications, such proof synthesis. In fact, in proof synthesis, the presence of an oracle (the proof checker) opens up new kinds of ensemble methods. Diva [18] takes advantage of this, using Coq's proof checker as an oracle to combine the predictions from multiple models. Thanks to Coq's proof checker, Diva can tell with certainty when a proof that it has suggested actually proves the theorem it is trying to prove. This makes it easier to combine models and provides strong guarantees. The result is an ensemble of diverse models with strong performance on the CoqGym benchmark.

Instead of combining multiple models on the same task, QEDCartographer gives each model a different task: the state value estimator picks which state to expand and the tactic predictor chooses how to expand it. This approach is complementary to other methods of combining models, and QEDCartographer could be used with multiple tactic predictors, as is the case with Diva, to further improve proof synthesis effectiveness.

Improving software quality is an important aspect engineering software system, which takes up 50–75% of the total software development budgets [59]. Automated program repair can improve program quality [1], [35], [44], [48], [52], [94], as well as the quality of other software artifacts [84], and can also help developers debug manually [15], but does not guarantee correctness, and, in fact, often introduces new bugs [53], [73]. The most common manual methods for improving quality are code reviews, testing [4], and debugging (typically with tool support [9], [36], [92]) but only formal verification can guarantee code correctness. Verification requires specifying properties as well as proving them, and our work has focuses on the latter step, but important research remains in supporting manually specifying properties, automatically generating formal specifications from natural language [16], [24], [51], [93], and extending the types of properties formal languages can capture, including privacy properties [82], data-based properties [54], [55], fairness properties [10], [21], among others. Probabilistic verification of certain properties, such as fairness, in certain types of software systems can be automated [3], [23], [28], [49], [79].

Proof assistants, such as Coq [78], Agda [83], Dafny [45], F* [76], Liquid Haskel [81], Mizar [80], Isabelle [58], HOL4 [72], and HOL Light [27] are semi-automated systems for theorem proving. Our work focuses on Coq, which has been used extensively [25], [26], [29], [31], [47], [50], [65], [71], [85], but the approach is applicable to other provers.

Heuristic-based search can partially automate proof synthesis [5], [8], [11], [12]. Hammers use external ATPs to find proofs [14]. Meanwhile Pumpkin Patch can learn from human-made repair patterns for software evolution [66]. Tracking fine-grained dependencies between Coq definitions, propositions, and proof scripts can prioritize failing proof scripts in evolving projects [13] and counterexamples can help increase confidence in theorem correctness [41]. Such tools are complementary to proof-synthesis tools, such as QEDCartographer.

VI. Contributions

We have presented QEDCartographer, a new tool for synthesizing proofs of theorems in the Coq proof assistant. QEDCartographer guides search more effectively than prior work by using reward-free reinforcement learning to learn to estimate the difficulty of proof states. Our evaluation showed empirically that QEDCartographer synthesizes more and shorter proofs, and does so more quickly than the prior state of the art. Our work provides an important framework for breaking down proof synthesis into tactic prediction, state value estimation, and intelligent search, identifying challenges research needs to tackle to help automate proof synthesis, lowering the barrier to verifying software and increasing software quality.

Acknowledgments

We thank Bruno Castro da Silva for help with reinforcement learning and Claire Le Goues for feedback on this paper. This work is supported by the National Science Foundation under grant no. CCF-2210243 and by the Air Force Research Laboratory (AFRL) and Defense Advanced Research Projects Agencies (DARPA) under Contract No. FA8750-24-C-B044.

References

[1] A. Afzal, M. Motwani, K. T. Stolee, Y. Brun, and C. Le Goues. SOSRepair: Expressive semantic search for real-world program repair. *IEEE Transactions on Software Engineering (TSE)*, 47(10):2162–2181, October 2021. doi: 10.1109/TSE.2019.2944914

[2] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama. Optuna: A next-generation hyperparameter optimization framework. In *ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*, pp. 2623–2631. Anchorage, AK, USA, 2019. doi: 10.1145/3292500.3330701

[3] A. Albarghouthi, L. D'Antoni, S. Drews, and A. Nori. FairSquare: Probabilistic verification for program fairness. In *ACM International Conference on Object Oriented Programming Systems Languages and Applications (OOPSLA)*, pp. 80:1–80:30. Vancouver, BC, Canada, 2017. doi: 10.1145/3133904

[4] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, New York, NY, USA, 1 ed., 2008.

[5] P. B. Andrews and C. E. Brown. TPS: A hybrid automatic-interactive system for developing proofs. *Journal of Applied Logic*, 4(4):367–395, 2006. doi: 10.1016/j.jal.2005.10.002

[6] K. Bansal, S. M. Loos, M. N. Rabe, C. Szegedy, and S. Wilcox. HOList: An environment for machine learning of higher order logic theorem proving. In *International Conference on Machine Learning (ICML)*, vol. 97, pp. 454–463. PMLR, Long Beach, CA, USA, 2019.

[7] K. Bansal, C. Szegedy, M. N. Rabe, S. M. Loos, and V. Toman. Learning to reason in large theories without imitation. *CoRR*, abs/1905.10501, 2019.

[8] J. C. Blanchette, L. Bulwahn, and T. Nipkow. Automatic proof and disproof in Isabelle/HOL. In *International Symposium on Frontiers of Combining Systems*, pp. 12–27. Springer, 2011. doi: 10.1007/978-3-642-24364-6_2

[9] Y. Brun, R. Holmes, M. D. Ernst, and D. Notkin. Speculative analysis: Exploring future states of software. In *Foundations of Software Engineering Working Conference on the Future of Software Engineering Research (FoSER)*, pp. 59–63. Santa Fe, NM, USA, November 2010. doi: 10.1145/1882362.1882375

[10] Y. Brun and A. Meliou. Software fairness. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE) New Ideas and Emerging Results Track*, pp. 754–759. Lake Buena Vista, FL, USA, November 2018. doi: 10.1145/3236024.3264838

[11] A. Bundy. A science of reasoning. In *International Conference on Automated Reasoning with Analytic Tableaux and Related Methods*, pp. 10–17. Springer, 1998. doi: 10.1007/3-540-69778-0_2

[12] A. Bundy, F. V. Harmelen, C. Horn, and A. Smaill. The $\mathrm{o}^{\mathrm{Y}}\mathrm{s}^{\mathrm{T}}\mathrm{er\text{-}cl}^{\mathrm{A}}\mathrm{m}$ system. In *International Conference on Automated Deduction (CADE)*, pp. 647–648. Springer, 1990. doi: 10.1007/3-540-52885-7_123

[13] A. Celik, K. Palmskog, and M. Gligoric. ICoq: Regression proof selection for large-scale verification projects. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 171–182. Urbana-Champaign, IL, USA, 2017. doi: 10.1109/ASE.2017.8115630

[14] Ł. Czajka and C. Kaliszyk. Hammer for Coq: Automation for dependent type theory. *Journal of Automated Reasoning*, 61(1-4):423–453, 2018. doi: 10.1007/s10817-018-9458-4

[15] H. Eladawy, C. Le Goues, and Y. Brun. Automated program repair, what is it good for? Not absolutely nothing! In *International Conference on Software Engineering (ICSE)*, pp. 1017–1029. Lisbon, Portugal, April 2024. doi: 10.1145/3597503.3639095

[16] M. Endres, S. Fakhoury, S. Chakraborty, and S. K. Lahiri. Can large language models transform natural language intent into formal method postconditions? *Proceedings of the ACM Software Engineering (PACMSE)*, 1(FSE):84:1–84:24, July 2024. doi: 10.1145/3660791

[17] A. Erbsen, J. Philipoom, J. Gross, R. Sloan, and A. Chlipala. Simple high-level code for cryptographic arithmetic — with proofs, without compromises. In *IEEE Symposium on Security and Privacy (S&P)*, pp. 1202–1219, 2019. doi: 10.1109/SP.2019.00005

[18] E. First and Y. Brun. Diversity-driven automated formal verification. In *International Conference on Software Engineering (ICSE)*, pp. 749–761. Pittsburgh, PA, USA, May 2022. doi: 10.1145/3510003.3510138

[19] E. First, Y. Brun, and A. Guha. TacTok: Semantics-aware proof synthesis. *Proceedings of the ACM on Programming Languages (PACMPL) Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA) issue*, 4:231:1–231:31, November 2020. doi: 10.1145/3428299

[20] E. First, M. Rabe, T. Ringer, and Y. Brun. Baldur: Whole-proof generation and repair with large language models. In *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 1229–1241. San Fransisco, CA, USA, December 2023. doi: 10.1145/3611643.3616243

[21] S. Galhotra, Y. Brun, and A. Meliou. Fairness testing: Testing software for discrimination. In *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 498–510. Paderborn, Germany, September 2017. doi: 10.1145/3106237.3106277

[22] T. Gauthier, C. Kaliszyk, and J. Urban. TacticToe: Learning to reason with HOL4 tactics. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*, vol. 46, pp. 125–143, 2017.

[23] S. Giguere, B. Metevier, Y. Brun, B. C. da Silva, P. S. Thomas, and S. Niekum. Fairness guarantees under demographic shift. In *International Conference on Learning Representations (ICLR)*, April 2022.

[24] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè. Automatic generation of oracles for exceptional behaviors. In *International Symposium on Software Testing and Analysis (ISSTA)*, pp. 213–224. Saarbrücken, Genmany, July 2016. doi: 10.1145/2931037.2931061

[25] R. Gu, Z. Shao, H. Chen, X. Wu, J. Kim, V. Sjöberg, and D. Costanzo. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2016.

[26] A. Guha, M. Reitblatt, and N. Foster. Machine verified network controllers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 483–494. Seattle, WA, USA, 2013. doi: 10.1145/2491956.2462178

[27] J. Harrison. HOL Light: A tutorial introduction. In *International Conference on Formal Methods in Computer-Aided Design (FMCAD)*, pp. 265–269. Palo Alto, CA, USA, 1996. doi: 10.1007/BFb0031814

[28] A. Hoag, J. E. Kostas, B. C. da Silva, P. S. Thomas, and Y. Brun. Seldonian toolkit: Building software with safe and fair machine learning. In *International Conference on Software Engineering (ICSE) Demo Track*, pp. 107–111. Melbourne, Australia, May 2023. doi: 10.1109/ICSE-Companion58688.2023.00035

[29] A. İleri, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Proving confidentiality in a file system using DISKSEC. In *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pp. 323–338. Carlsbad, CA, USA, 2018.

[30] K. Jacobs and B. Beurdouche. Performance improvements via formally-verified cryptography in Firefox. Mozilla Security Blog, 2020.

[31] D. Jang, Z. Tatlock, and S. Lerner. Establishing browser security guarantees through formal shim verification. In *USENIX Security Symposium (USENIX Security)*, pp. 113–128. Bellevue, WA, USA, Aug. 2012.

[32] A. Jiang, K. Czechowski, M. Jamnik, P. Milos, S. Tworkowski, W. Li, and Y. T. Wu. Thor: Wielding hammers to integrate language models and automated theorem provers. In *Neural Information Processing Systems (NeurIPS)*. New Orleans, LA, USA, 2022.

[33] A. Q. Jiang, W. Li, J. M. Han, and Y. Wu. LISA: Language models of ISAbelle proofs. In *Conference on Artificial Intelligence and Theorem Proving (AITP)*, pp. 17.1–17.3. Aussois, France, September 2021.

[34] A. Q. Jiang, S. Welleck, J. P. Zhou, W. Li, J. Liu, M. Jamnik, T. Lacroix, Y. Wu, and G. Lample. Draft, sketch, and prove: Guiding formal theorem provers with informal proofs. In *International Conference on Learning Representations (ICLR)*, 2023.

[35] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen. Shaping program repair space with existing patches and similar code. In *ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 298–309. Amsterdam, The Netherlands, July 2018. doi: 10.1145/3213846.3213871

[36] B. Johnson, Y. Brun, and A. Meliou. Causal testing: Understanding defects' root causes. In *International Conference on Software Engineering (ICSE)*, pp. 87–99. Seoul, Republic of Korea, June 2020. doi: 10.1145/3377811.3380377

[37] C. Kaliszyk, F. Chollet, and C. Szegedy. HolStep: A machine learning dataset for higher-order logic theorem proving. In *International Conference on Learning Representations (ICLR)*. Toulon, France, Apr. 2017.

[38] C. Kaliszyk, J. Urban, H. Michalewski, and M. Olšák. Reinforcement learning of theorem proving. In *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 31, 2018.

[39] H. Krasner. The cost of poor software quality in the US: A 2022 report. https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-Report-Nov-22-2.pdf, 2022.

[40] G. Lample, T. Lacroix, M.-A. Lachaux, A. Rodriguez, A. Hayat, T. Lavril, G. Ebner, and X. Martinet. Hypertree proof search for neural theorem proving. In *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, pp. 26337–26349, 2022.

[41] L. Lampropoulos, Z. Paraskevopoulou, and B. C. Pierce. Generating good generators for inductive relations. *Proceedings of the ACM on Programming Languages (PACMPL)*, 2(POPL):45:1–45:30, Dec. 2017. doi: 10.1145/3158133

[42] B. Lasse, J. Urban, and H. Geuvers. The Tactician: A seamless, interactive tactic learner and prover for Coq. In *International Conference on Intelligent Computer Mathematics (CICM)*, pp. 271–277, 2020. doi: 10.1007/978-3-030-53518-6_17

[43] C. Lattner and N. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization (CGO)*, pp. 75–86. Palo Alto, CA, USA, Mar. 2004. doi: 10.1109/CGO.2004.1281665

[44] C. Le Goues, M. Pradel, and A. Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, Nov. 2019. doi: 10.1145/3318162

[45] K. R. M. Leino. Dafny: An automatic program verifier for functional correctness. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning (LPAR)*. Dakar, Senegal, 2010. doi: 10.1007/978-3-642-17511-4_20

[46] X. Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 42–54. Charleston, SC, USA, 2006. doi: 10.1145/1111037.1111042

[47] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM (CACM)*, 52(7):107–115, 2009. doi: 10.1145/1538788.1538814

[48] K. Liu, A. Koyuncu, D. Kim, and T. F. Bissyandé. TBar: Revisiting template-based automated program repair. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 31–42. Beijing, China, 2019. doi: 10.1145/3293882.3330577

[49] B. Metevier, S. Giguere, S. Brockman, A. Kobren, Y. Brun, E. Brunskill, and P. S. Thomas. Offline contextual bandits with high probability fairness guarantees. In *Annual Conference on Neural Information Processing Systems (NeurIPS), Advances in Neural Information Processing Systems 32*, pp. 14893–14904. Vancouver, BC, Canada, December 2019.

[50] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan. RockSalt: Better, faster, stronger SFI for the x86. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. Beijing, China, 2012. doi: 10.1145/2345156.2254111

[51] M. Motwani and Y. Brun. Automatically generating precise oracles from structured natural language specifications. In *International Conference on Software Engineering (ICSE)*, pp. 188–199. Montreal, QC, Canada, May 2019. doi: 10.1109/ICSE.2019.00035

[52] M. Motwani and Y. Brun. Better automatic program repair by using bug reports and tests together. In *International Conference on Software Engineering (ICSE)*, pp. 1229–1241. Melbourne, Australia, May 2023. doi: 10.1109/ICSE48619.2023.00109

[53] M. Motwani, M. Soto, Y. Brun, R. Just, and C. Le Goues. Quality of automated program repair on real-world defects. *IEEE Transactions on Software Engineering (TSE)*, 48(2):637–661, February 2022. doi: 10.1109/TSE.2020.2998785

[54] K. Muşlu, Y. Brun, and A. Meliou. Data debugging with continuous testing. In *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE) New Ideas Track*, pp. 631–634. Saint Petersburg, Russia, August 2013. doi: 10.1145/2491411.2494580

[55] K. Muşlu, Y. Brun, and A. Meliou. Preventing data errors with continuous testing. In *ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 373–384. Baltimore, MD, USA, July 2015. doi: 10.1145/2771783.2771792

[56] A. Nair, P. Srinivasan, S. Blackwell, C. Alcicek, R. Fearon, A. De Maria, V. Panneershelvam, M. Suleyman, C. Beattie, S. Petersen, S. Legg, V. Mnih, K. Kavukcuoglu, and D. Silver. Massivelyparallel methods for deep reinforcement learning. In *Deep Learning Workshop*. Lille, France, 2015.

[57] A. Y. Ng, D. Harada, and S. J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *International Conference on Machine Learning (ICML)*, pp. 278–287. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1999.

[58] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A proof assistant for higher-order logic*, vol. 2283. Springer Science & Business Media, 2002.

[59] D. H. O'Dell. The debugging mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills. *Queue*, 15(1):71–90, Feb. 2017. doi: 10.1145/3055301.3068754

[60] A. Paliwal, S. M. Loos, M. N. Rabe, K. Bansal, and C. Szegedy. Graph representations for higher-order logic and theorem proving. In *Conference on Artificial Intelligence (AAAI)*, pp. 2967–2974. AAAI Press, New York, NY, USA, 2020.

[61] D. Pathak, P. Agrawal, A. A. Efros, and T. Darrell. Curiosity-driven exploration by self-supervised prediction. In *International Conference on Machine Learning (ICML)*. Sydney, Australia, Aug. 2017.

[62] L. Paulson and T. Nipkow. The Sledgehammer: Let automatic theorem provers write your Isabelle scripts! https://isabelle.in.tum.de/website-Isabelle2009-1/sledgehammer.html, 2023.

[63] S. Polu and I. Sutskever. Generative language modeling for automated theorem proving. *CoRR*, 2020.

[64] J. Randløv and P. Alstrøm. Learning to drive a bicycle using reinforcement learning and shaping. In *International Conference on Machine Learning (ICML)*, pp. 463–471. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[65] T. Ringer, K. Palmskog, I. Sergey, M. Gligoric, and Z. Tatlock. QED at large: A survey of engineering of formally verified software. *Foundations and Trends®in Programming Languages*, 5(2–3):102–281, 2019. doi: 10.1561/2500000045

[66] T. Ringer, N. Yazdani, J. Leo, and D. Grossman. Adapting proof automation to adapt proofs. In *ACM SIGPLAN International Conference on Certified Programs and Proofs (CPP)*, pp. 115–129. Los Angeles, CA, USA, 2018. doi: 10.1145/3167094

[67] J. Rute, M. Olšák, L. Blaauwbroek, F. I. S. Massolo, J. Piepenbrock, and V. Pestun. Graph2Tac: Learning hierarchical representations of math concepts in theorem proving. *CoRR*, abs/2401.02949, 2024.

[68] A. Sanchez-Stern, Y. Alhessi, L. Saul, and S. Lerner. Generating correctness proofs with neural networks. In *ACM SIGPLAN International Workshop on Machine Learning and Programming Languages (MAPL)*, pp. 1–10, 2020.

[69] A. Sanchez-Stern, E. First, T. Zhou, Z. Kaufman, Y. Brun, and T. Ringer. Passport: Improving automated formal verification using identifiers. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 45(2):12:1–12:30, June 2023. doi: 10.1145/3593374

[70] S. Schaal. Learning from demonstration. In *International Conference on Neural Information Processing Systems (NeurIPS)*, p. 1040–1046. MIT Press, Denver, Colorado, 1996.

[71] I. Sergey, J. R. Wilcox, and Z. Tatlock. Programming and proving with distributed protocols. *Proceedings of the ACM on Programming Languages (PACMPL)*, 2(POPL):28:1–28:30, Dec. 2017. doi: 10.1145/3158116

[72] K. Slind and M. Norrish. A brief overview of HOL4. In *International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, pp. 28–32. Montreal, QC, Canada, 2008. doi: 10.1007/978-3-540-71067-7_6

[73] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? Overfitting in automated program repair. In *Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pp. 532–543. Bergamo, Italy, September 2015. doi: 10.1145/2786805.2786825

[74] J. Souyris. Industrial use of compcert on a safety-critical software product. http://projects.laas.fr/IFSE/FMF/J3/slides/P05_Jean_Souyiris.pdf, 2014.

[75] R. M. Stallman. *Using the GNU Compiler Collection*. Free Software Foundation, Boston, MA, USA, 2012.

[76] N. Swamy, C. Hriţcu, C. Keller, A. Rastogi, A. Delignat-Lavaud, S. Forest, K. Bhargavan, C. Fournet, P.-Y. Strub, M. Kohlweiss, J.-K. Zinzindohoue, and S. Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 256–270. St. Petersburg, FL, USA, 2016. doi: 10.1145/2914770.2837655

[77] A. Thakur, G. Tsoukalas, Y. Wen, J. Xin, and S. Chaudhuri. An in-context learning agent for formal theorem-proving. In *Conference on Language Modeling (COLM)*. Philadelphia, PA, USA, 2024.

[78] The Coq Development Team. Coq, v.8.7. https://coq.inria.fr, 2017.

[79] P. S. Thomas, B. C. da Silva, A. G. Barto, S. Giguere, Y. Brun, and E. Brunskill. Preventing undesirable behavior of intelligent machines. *Science*, 366(6468):999–1004, 22 November 2019. doi: 10.1126/science.aag3311

[80] A. Trybulec and H. A. Blair. Computer assisted reasoning with MIZAR. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, vol. 85, pp. 26–28. Los Angeles, CA, USA, 1985.

[81] N. Vazou. *Liquid Haskell: Haskell as a theorem prover*. PhD thesis, University of California, San Diego, 2016.

[82] Véronique Cortier, N. Grimm, J. Lallemand, and M. Maffei. A type system for privacy properties. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pp. 409–423. Association for Computing Machinery, Dallas, Texas, USA, 2017. doi: 10.1145/3133956.3133998

[83] P. Wadler, W. Kokke, and J. G. Siek. *Programming Language Foundations in Agda*. July 2020.

[84] A. Weiss, A. Guha, and Y. Brun. Tortoise: Interactive system configuration repair. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 625–636. Urbana-Champaign, IL, USA, October/November 2017. doi: 10.1109/ASE.2017.8115673

[85] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 357–368. Portland, OR, USA, June 2015.

[86] M. Wu, M. Norrish, C. Walder, and A. Dezfouli. TacticZero: Learning to prove theorems from scratch with deep reinforcement learning. *CoRR*, abs/2102.09756, 2021.

[87] Y. Wu, M. N. Rabe, D. Hutchins, and C. Szegedy. Memorizing transformers. In *International Conference on Learning Representations (ICLR)*, 2021.

[88] K. Yang and J. Deng. Learning to prove theorems via interacting with proof assistants. In *International Conference on Machine Learning (ICML)*. Long Beach, CA, USA, 2019.

[89] K. Yang, A. M. Swope, A. Gu, R. Chalamala, P. Song, S. Yu, S. Godil, R. Prenger, and A. Anandkumar. LeanDojo: Theorem proving with retrieval-augmented language models. In *Conference on Neural Information Processing Systems (NeurIPS) Track on Datasets and Benchmarks*, 2023.

[90] X. Yang, Y. Chen, E. Eide, and J. Regehr. Finding and understanding bugs in C compilers. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp. 283–294. San Jose, CA, USA, 2011. doi: 10.1145/1993498.1993532

[91] Y. D. Yang, J. P. Inala, O. Bastani, Y. Pu, A. Solar-Lezama, and M. Rinard. Program synthesis guided reinforcement learning for partially observed environments. In *International Conference on Neural Information Processing Systems (NeurIPS)*, pp. 29669–29683. Red Hook, NY, USA, 2021.

[92] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, February 2002. doi: 10.1109/32.988498

[93] J. Zhai, Y. Shi, M. Pan, G. Zhou, Y. Liu, C. Fang, S. Ma, L. Tan, and X. Zhang. C2S: Translating natural language comments to formal program specifications. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pp. 25–37, 2020. doi: 10.1145/3368089.3409716

[94] Q. Zhu, Z. Sun, Y. an Xiao, W. Zhang, K. Yuan, Y. Xiong, and L. Zhang. A syntax-guided edit decoder for neural program repair. In *European Software Engineering Conference and ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE)*, pp. 341–353. Athens, Greece, 2021. doi: 10.1145/3468264.3468544

[95] Z. Zombori, A. Csiszárik, H. Michalewski, C. Kaliszyk, and J. Urban. Towards finding longer proofs. In *International Conference on Theorem Proving with Analytic Tableaux and Related Methods*, pp. 167–186, 2021. doi: 10.1007/978-3-030-86059-2_10

[96] Z. Zombori, J. Urban, and C. E. Brown. Prolog technology reinforcement learning prover. In *International Joint Conference on Automated Reasoning (IJCAR)*, pp. 489–507, 2020. doi: 10.1007/978-3-030-51054-1_33