Automated Theorem Proving via Interacting with Proof Assistants by Dynamic Strategies

Guangshuai Mo, Yan Xiong, Wenchao Huang and Lu Ma

May 5, 2020

Automated Theorem Proving via Interacting with Proof Assistants by Dynamic Strategies

Guangshuai Mo*, Yan Xiong*, Wenchao Huang*, Lu Ma†
*School of Computer Science and Technology
University of Science and Technology of China, Hefei, Anhui, PR China
†Beijing Institute of Remote Sensing, Beijing 122000, PR China
Email: mgs@mail.ustc.edu.cn, {yxiong, huangwc}@ustc.edu.cn, syml1234@163.com

Abstract—**Proof assistants offer a formal language to write mathematical definitions, executable algorithms, and theorems together with an environment for interactive development of machine-checked proofs. Developers manually construct definitions and lemmas on proof assistants to prove the theorems. However, the time and labor costs of manually proving theorems in proof assistants remain prohibitively high. Therefore, proving the theorem automatically for the sizeable formal project becomes significant. In this paper, we propose SmartCoq, a proof search system that uses a dynamic strategy to solve the problem of automating the interaction with proof assistants. The design of our dynamic strategy is flexible and straightforward: it can automatically optimize itself based on theorems without manual intervention. SmartCoq uses dynamic strategies to learn from the wrong paths in the past and find a correct path to complete the proof. We demonstrate SmartCoq on the proof obligations from a large practical proof project, the CompCert verified C compiler, and the result shows that it can automatically solve 14.5% of proofs in our test dataset.**

Index Terms—**Proof assistants, Interactive theorem proving, Machine learning**

I. Introduction

Humans write theorems into automated theorem proving (ATP), and the ATP output is the result of the proof. ATP converts the premises and goals of the theorem into first-order clauses in conjunctive normal form (CNF) and proves the theorem by proof first-order clauses. While this provides a universal procedure, the CNF representation of simple formulas can be long, complicated and inscrutable, making it difficult to benefit from the higher-level abstraction and manipulation that is common to human mathematical reasoning.

Due to the difficulty of the practical application of ATP, interactive theorem proving (ITP) [6] was proposed, such as Coq [2] [5] or Isabelle HOL [15] [3] [10]. ITP allows people to prove a theorem by a formal language called tactics manually. The tactics combine high-level proof techniques such as the Calculus of Inductive Constructions, put the low-level details of the proof to proof assistants. One can use tactics to prove various theorems and executable algorithms on the proof assistant, including the formalization of mathematics, certification of properties of programming languages, database systems, compilers.

However, certification on ITP in proof assistants takes too much time and labour. For example, the CompCert compiler certification project [13] took six person-years, which include 175 Coq files and 100,000 lines of Coq to write and verify, and Iris Project [9] took five person-years, which include 143 Coq files. For a project with limited resources, this consumption is intolerable. Therefore, proving the theorem automatically is essential and meaningful work.

Proving the theorem automatically, however, is a complicated task, which suffers from two challenges. One is generating some corresponding tactics and arguments for the current context (including hypothesis and goals) at every step of the certification. Generating the corresponding tactic from 221 tactics libraries, and the arguments can be related to intermediate lemma searched from the global, expressions and a sophisticated line of code with functions. Some previous work was to use machine learning to train and evaluate datasets to find the right tactics and arguments, such as ML4PG [7], Gamepad [8], TacticToe [4]. The other is to find correct command for corresponding tactics and arguments. After generating some relevant tactics, we need to find the correct command, which is composed of tactics and arguments.

In this paper, we propose SmartCoq, a proof search system that proves the theorem automatically in Coq proof assistants. SmartCoq uses an algorithm to search the relevant lemma globally to generate tactics and arguments. Afterwards, with tactics and parameters as nodes, a proof search tree is established. Then finally, we use a dynamic strategy inside SmartCoq, which can be used to smartly search proof paths.

Our dynamic strategy introduces Deep Q Network (DQN) [14], a reinforcement learning agent, into the proving method. Update the strategy according to incorrect historical paths in DQN, and the successful paths will get a higher score. The DQN supports the training of neural networks with stochastic gradient descent in a stable manner. As a result, an optimized strategy tends to select a node getting a higher score, which leads to a higher probability of successful proving. Linking these nodes together is the proof path we need.

The innovation of our work is to use DQN to find a correct path in the search tree dynamically. Experimental results show that SmartCoq can automatically prove the theorem and solve 14.5% of proofs in our test dataset, without any human intervention.

We make the following contributions in this paper:

- We take a new approach for generating tactics and arguments more comprehensively and searching the relevant lemma globally.

- We used a dynamic strategy to optimize the search proof paths, which leads to a higher probability of success in proving.
- We demonstrate that SmartCoq can solve 14.5% of the proofs in our test dataset of CompCert, which is over the previous state of the art system that attempts to the same task.

The rest of this paper is organized as follows. Section II introduces the related work. Section III introduces the structure of SmartCoq and details of each part of SmartCoq. Section IV introduces the details of dynamic strategies. Section V introduces our experiment and section VI concludes this paper.

II. Related Work

Some proof assistants allow a user to use existing Automated theorem proving systems directly. Automated theorem proving Modern theorem provers [12] transform the theorem into first-order logic, and it then proves the theorems using external provers.

Machine learning provides a new way to prove the theorem [16] [11], although it still has some flaws. When proving a theorem via Machine learning, it requires building a large number of available options and searching for those options that are needed. Machine learning has been used for various tasks such as code and patch generation [1], program classification, and learning loop invariants.

Some existing automatic theorem proving methods often use machine learning to build tactics and arguments [17] automatically.

ML4PG [7] introduces benchmark suites and frameworks for exploring machine learning in Coq. ML4PG, while it focuses on the development of methods of interactive interfacing between the ITP and machine learning interfaces and does not attempt to generate proofs.

Gamepad [8] provides a structured Python representation of Coq proofs, including all the proof states encountered in a proof, the steps taken, and expression abstract syntax trees (ASTs). However, Gamepad trains mainly on synthetic data of simple proofs but does not combine their proof command predictor with a high-level search.

TacticToe [4] learns from human proofs which mathematical technique is suitable in each proof situation, which is then used in a Monte Carlo tree search algorithm to explore promising tactic-level proof paths.

CoqGym [17] provides a large-scale dataset and learning environment for theorem proving via interacting with a proof assistant. CoqGym learns to generate tactics as abstract syntax trees and can be used to prove new theorems beyond the reach of previous automatic provers.

Compared with ML4PG, Gamepad, and CoqGym, SmartCoq has some essential conceptual differences. First, when using machine learning to build tactics and arguments automatically, the training dataset may not be in the same project, such as Hoare logic proof and mathematical proof. At the same time, we cannot determine the strong correlation between proof command. Therefore, SmartCoq takes another approach, pattern matching, to build commands. Second, SmartCoq uses Deep Q Network to find the correct path in the search tree, instead of using depth-first or breadth-first, which is determined by the method used by SmartCoq to generate commands.

III. The Proposed Method

In this section, we will present SmartCoq's internal details. We can see the top-level structure of SmartCoq in Fig. 1.

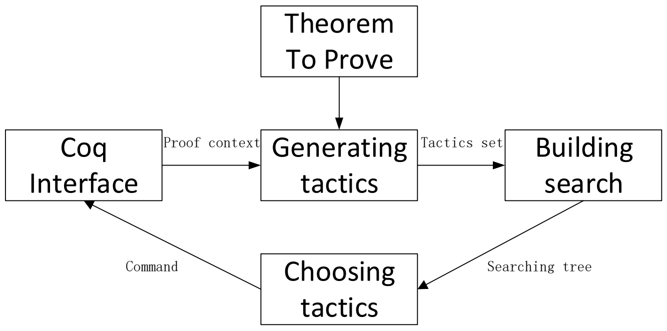

Fig. 1. The overall architecture of SmartCoq

In the Coq Interface section, Coq proof assistants will provide SmartCoq with the context (hypothesis and goals) of the proof, when we give a theorem and give a command to prove the theorem. The Generating tactics section will search the global lemma based on the context of the proof to generate some tactics and tactic arguments. Afterward, the Choosing tactics section will choose a command from tactics and push it to Coq Interface. After learning and optimizing from incorrect historical paths in DQN, SmartCoq will choose a correct proof path.

Given a theorem, the workflow of SmartCoq consists of the following steps:

- Step 1: The Deep Q Network (DQN) is initialized with a purely random strategy, which takes multiple candidates (The Generating tactics and tactic arguments in section A) as input, and randomly chooses a candidate with the uniform probability as output.
- Step 2: SmartCoq conducts proof searching by using the current proof command, which was generated by Step 1. Simultaneously, Coq proof assistants generate the context (hypothesis and goals) of the current command.
- Step 3: If the path generated in Step 2 is correct and complete, SmartCoq will terminates and outputs the path as the result.
- Step 4: Otherwise, if it is estimated that all the paths generated in Step 2 are incorrect, SmartCoq will start a new round, in which DQN will be trained according to the incorrect proof path, and the strategy will be updated. Here, we use the term episode to denote the time step in which the DQN is optimized with a new reward. Afterward, go to Step 2.

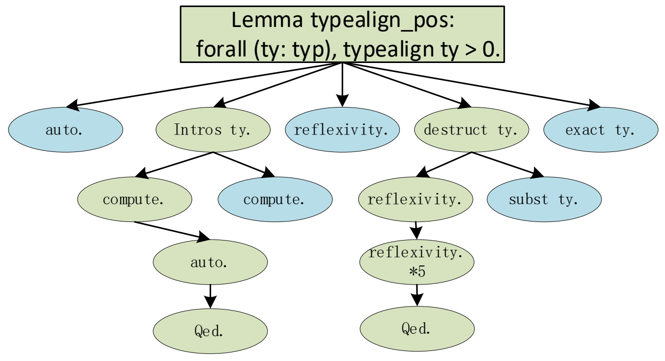

Fig. 2. A graph of a SmartCoq search

A. Generating tactics and tactic arguments

To generate tactics and tactic arguments, we collect the Coq standard library and some projects as a Coq dataset. The Coq dataset contains 10755952 lines of Coq command and Coq theorem. In the dataset, we found that manual proofs use very few proof tactics in all Coq tactics. Some tactics are used frequently, and others are rarely used or not used in manual proofs, which gives us the possibility to model the Coq proof command.

Through the analysis and observation of the coq dataset, SmartCoq will build three kinds of Coq command models as the nodes for the next step: tactics without arguments, tactics with local arguments, and tactics with global arguments.

*1) **tactics without arguments**:* SmartCoq contains 33 tactics without arguments, which are used as the nodes of the next step. For instance, *intros*, *reflexivity*, and *decide equality*. Based on the observation of the Coq dataset, we consider these tactics to be effective and sufficient.

*2) **tactics with local arguments**:* SmartCoq collects 12 tactics with local arguments in the Coq dataset. For instance, *destruct*, *apply* and *unfold*. Tactics with local arguments contain two kinds of arguments, Goal-token arguments, and Hypothesis-identifier arguments. For instance, if the goal is: *negb (negb b) = b*, SmartCoq will take negb and b as Goal-token arguments. In the case of tactics like *destruct* and *unfold*, the argument is often a token in the goal. If the hypothesis is:

n, m : nat
H : n = m

SmartCoq will take H as Hypothesis-identifier arguments. In this case, the argument is often a token in the hypothesis.

*3) **tactics with global arguments**:* Prior some works were trained and evaluated on data set using machine learning. However, Machine learning does not solve the problem of searching for global arguments. On the one hand, the data set is too small, and on the other hand, there is no effective machine learning algorithm.

Our strategy builds upon the insight that there must be some common tokens in context and lemma. For instance, we use the command of *rewrite plus_comm* and *plus_comm* is *n + m = m + n* when the goal is *n + m + (p + q) = m + n + (p + q)*. SmartCoq searches n, m, +, p and q as the tokens in all theorems. SmartCoq counts the number of common tokens in context and lemma as a basis for using global arguments.

B. Building search

This module generates a validation tree as input to the Coq proof assistants. The search tree uses the commands generated in the first part as the nodes of the tree. The command is stored in the node, and the children of a node are the commands after the command in proof. The paths in the tree correspond to possible proof paths in verification. SmartCoq uses the method of building a dynamic Search Tree to establish proofs. A path from the root node to the leaf node is the proof command generated by SmartCoq.

In Fig. 2, this is an example of SmartCoq's search tree. It is a Lemma *typealign_pos* in CompCert project. In green are the tactics that formed part of the discovered solution, as well as the lemma name and QED(terminator). In blue are nodes that resulted in a context that is at least as hard as one previously found. In Fig. 2, the proof of this theorem is *Intors ty. compute. auto. Qed.*. It can be seen from the figure that there are two solutions that can solve the proof.

C. Choosing tactics

After building a search tree, SmartCoq uses Deep Q Network to find the proof path for searching. The advantage of our algorithm is that it can automatically optimize the path chosen by the algorithm. We will show the details of our algorithm in section IV. The depth-first search is also an excellent way to handle search paths. However, the use of a depth-first search leads to problems such as multiple sub-goals. Here, we discuss some issues with a depth-first search in experiments on Session V.

IV. Dynamic Strategies

We achieve our dynamic strategy by using the DQN and designing the rewards based on the insight. In this part, we introduce two algorithms, one is the framework of DQN, and the other is the selection and optimization algorithm.

In Algorithm 1, we introduce the Implementation of Deep Q Network, which is the framework of our dynamic strategy. The N represents the total memory of the system. The DQN runs iteratively with multiple-episodes. In each episode, SmartCoq executes the path selection algorithm. If a path is estimated correctly and complete, SmartCoq will terminate the program, and the generated path is a proof of correctness (line 7). If all the selected paths are estimated incorrectly, the policy is optimized (line 9).

Algorithm 2 contains a path selection algorithm and policy optimization algorithm. In path selection algorithm, the DQN in SmartCoq maintains an action-selection policy which takes a state s_t as input, and outputs an action number a_t. Here,

Algorithm 1 Implementation of Deep Q Network

```
1: Initialize a replay memory D to capacity N
2: Initialize an action-value function Q
3: success = False
4: for $i$ to $episode$ do
5:     Execute path_selection
6:     if success = True then
7:         Program ends
8:     end if
9:     Execute policy_optimization
10: end for
```

s_t represents the proof state, which is a node in the Search Tree, and a_t is the number of the selected child of the node. The state s_t is a six-dimensional vector transformed from the context information in the node. Note that because the network is optimized through the historical selection of rewards, it is important that the states of different nodes in the tree are identical. It is possible to have the same state on different paths. For instance, when we use *induction* tactic to prove, the proof path (command) of the first sub-goal may be different, but in the second sub-goal, the different paths have the same context.

Algorithm 2 Selection and Optimization

```
1: function path_selection:
2: initialize a proof state $s_1$
3: for $t = 1$ to $ROUND$ do
4:     With probability $\varepsilon$ select a random action $a_t$
5:     otherwise select $a_t$=$max_a Q(s_t, a, \theta_e)$
6:     Generate next state $s_{t+1}$ according to $a_t$
7:     Store a transition $(s_t, a_t, \omega, s_{t+1})$ in D
8:     Update state $s_t$
9:     if the path is estimated incorrect then
10:        break
11:    end if
12:    if the path is estimated correct and complete then
13:        success=True
14:        return
15:    end if
16: end for
17:
18: function policy_optimization:
19: Sample n random transitions $(s_j, a_j, r_j, s_{j+1})$
20: Set $y_j = r_j + \gamma max_{a'} Q(s_{j+1}, a'; \theta_e)$
21: Perform a gradient descent step on $(y_j - Q(s_j, a_j, \theta_{e+1}))^2$
```

SmartCoq uses two strategies in the policy to choose action. Combining the two strategies, we use a ε-greedy strategy to select actions. The first exploration strategy (line 4): 1) is to choose random actions, which is to explore the values of unchosen actions. Another exploration strategy (line 5): 2) is a greedy strategy to choose a which may have the largest Q value currently. Here, given the node s_t and its ath child, $Q(s_t, a, \theta_e)$ outputs comparable value. The Q function also takes θ_e as input, where θ_e is the set of the DQN's parameters at episode e, and θ_e is updated into θ_{e+1} in policy optimization.

We set the reward to the same negative number for all the edges on each estimated incorrect proof path. In line 7, a transition, *i.e.*, tuple $(s_t, a_t, \omega, s_{t+1})$, is generated and added to D which is a replay memory, where ω is the negative reward for the action a_t at the state s_t. In line 8, s_t will be updated according to s_{t+1}. Finally, the program end condition is determined.

The parameters of the DQN is optimized by minimizing a loss function. In the policy optimization algorithm, θ_e in Q function is updated as mentioned. Here, n tuples are randomly selected from D. For each selected tuple (s_j, a_j, r_j, s_{j+1}), we compute y_i according to θ_e (line 20). Then θ_{e+1} is estimated by using the loss function $(y_j - Q(s_j, a_j, \theta_{e+1}))^2$ (line 21). Minimizing a loss function optimizes parameters in the DQN, through the optimization of these historical paths, we find our correct path.

The advantage of applying DQN is that DQN can update our dynamic strategy efficiently if the reward in DQN is designed effectively. Finally, by learning the wrong path, DQN can find a correct proof path.

V. Experiments

This section shows the experiments of SmartCoq. On the search tree, we first experimented with a depth-first search in our dataset. However, in the process of proving search, we found some flaws in the depth-first search.

First, multiple sub-goal: the use of depth-first search results in an increase in sub-goals, which proves to be incomplete. For instance, when we prove the theorem in the Coq standard library, Each time an *inversion x* tactic is used to prove *negation_fn_applied_twice* theorem, a sub-goal is added. Second, ranking algorithm: the premise of using the depth-first search is a good ranking algorithm. However, due to the complexity of goals, hypothesis and tactics, there is currently no good algorithm.

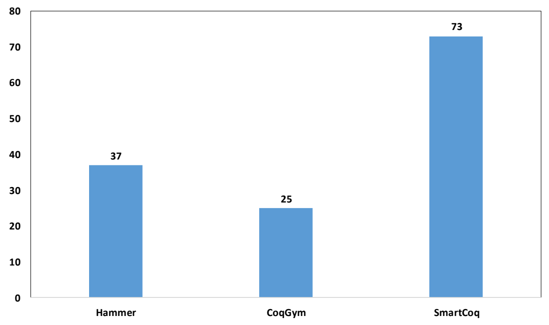

Fig. 3. A comparison of SmartCoq and CoqGym's abilities to complete proofs

A. Proof production

TABLE I is the distribution of SmartCoq's tactics. SmartCoq counts different uses of a tactics as different tactics. In our work, we collect 10755952 lines of Coq command and Coq theorem to build a tactic set.

TABLE I
DISTRIBUTION OF TACTICS

Tactics	Number
Without arguments	33
With local arguments	12
With global arguments	12

Running our experiments on CompCert verified C compiler, we show SmartCoq's ability to produce correct proofs. We tested CoqGym end-to-end by training on the proofs from 162 files from CompCert, and testing on the proofs from 13 files. The test set contains 501 proof scripts, all of which are manually proven by humans. We do this for two reasons: First, some existing work uses machine learning to generate tactics and arguments. We need to build a training dataset in the project. Second, when different items are used as training sets at the same time, it does not prove that the problems are applicable at the same time.

B. Comparison to others

In Fig. 3., Hammer solves 37/501, as a single invocation; CoqGym solves 25/501 and ours solves 73/501 of the proofs in CompCert verified C compiler. From these data, we can see that we have improved the efficiency of automated proofs. Nevertheless, in our experiments, these 73 proofs are still relatively short, and most of them do not exceed 20 in length. These need us to improve in future work.

TABLE II
PERCENTAGE OF THEOREMS SUCCESSFULLY PROVED

Method	Success rate (%)
trivial	2.4
auto	2.9
intuistion	4.4
easy	4.9
hammer (default time limit)	17.8
CoqGym	12.2
CoqGym + auto	12.8
ours + auto	**14.5**

In TABLE II, we compare experimentally to previous work. It can be seen that the existing certification work automatically is at a relatively low level of efficiency. Hammer proved to be more efficient, reaching 17.8%, with no other work exceeding 15%. However, in the experiment, the Hammer did not reach 17.8%. We think this is why we are running SmartCoq in actual projects. Compared to some Coq standard libraries, the actual project is more complicated, with a longer length and more complex structure.

VI. CONCLUSION

This paper solves the problem of automating the interaction with proof assistants. We generate three tactics and arguments in new ways to automate the proof. We have also optimized the search proof paths and adopt a dynamic strategy to solve the theorem automatic proof, which leads to more efficient proof. In future work, we will continue to optimize the method of generating tactics and arguments. The efficiency of searching using a search tree is also a problem that needs to be improved.

ACKNOWLEDGEMENT

The research is supported by the National Key R&D Program of China 2018YFB0803400, 2018YFB2100300, National Natural Science Foundation of China under Grant No.61972369, No.61572453, No.61520106007, No.61572454, and the Fundamental Research Funds for the Central Universities, No. WK2150110009.

REFERENCES

[1] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018.

[2] B. Barras, S. Boutin, C. Cornes, J. Courant, J.-C. Filliatre, E. Gimenez, H. Herbelin, G. Huet, C. Munoz, C. Murthy *et al.*, "The coq proof assistant reference manual: Version 6.1," 1997.

[3] S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, *Theorem Proving in Higher Order Logics: 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009, Proceedings*. Springer, 2009, vol. 5674.

[4] T. Gauthier, C. Kaliszyk, and J. Urban, "Tactictoe: Learning to reason with hol4 tactics," *arXiv preprint arXiv:1804.00595*, 2018.

[5] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. O. Biha *et al.*, "A machine-checked proof of the odd order theorem," in *International Conference on Interactive Theorem Proving*. Springer, 2013, pp. 163–179.

[6] J. Harrison, J. Urban, and F. Wiedijk, "History of interactive theorem proving." in *Computational Logic*, vol. 9, 2014, pp. 135–214.

[7] J. Heras and E. Komendantskaya, "Ml4pg in computer algebra verification," in *International Conference on Intelligent Computer Mathematics*. Springer, 2013, pp. 354–358.

[8] D. Huang, P. Dhariwal, D. Song, and I. Sutskever, "Gamepad: A learning environment for theorem proving," *arXiv preprint arXiv:1806.00608*, 2018.

[9] R. Jung, R. Krebbers, J.-H. Jourdan, A. Bizjak, L. Birkedal, and D. Dreyer, "Iris from the ground up: A modular foundation for higher-order concurrent separation logic," *Journal of Functional Programming*, vol. 28, 2018.

[10] C. Kaliszyk and J. Urban, "Learning-assisted automated reasoning with flyspeck," *Journal of Automated Reasoning*, vol. 53, no. 2, pp. 173–213, 2014.

[11] C. Kaliszyk, J. Urban, H. Michalewski, and M. Olšák, "Reinforcement learning of theorem proving," in *Advances in Neural Information Processing Systems*, 2018, pp. 8822–8833.

[12] L. Kovács and A. Voronkov, "First-order theorem proving and vampire," in *International Conference on Computer Aided Verification*. Springer, 2013, pp. 1–35.

[13] X. Leroy, "Formal verification of a realistic compiler," *Communications of the ACM*, vol. 52, no. 7, pp. 107–115, 2009.

[14] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[15] L. C. Paulson, "Natural deduction as higher-order resolution," *The Journal of Logic Programming*, vol. 3, no. 3, pp. 237–258, 1986.

[16] M. Wang, Y. Tang, J. Wang, and J. Deng, "Premise selection for theorem proving by deep graph embedding," in *Advances in Neural Information Processing Systems*, 2017, pp. 2786–2796.

[17] K. Yang and J. Deng, "Learning to prove theorems via interacting with proof assistants," *arXiv preprint arXiv:1905.09381*, 2019.