Applied Intelligence
https://doi.org/10.1007/s10489-020-01837-7

Proof searching and prediction in HOL4 with evolutionary/heuristic and deep learning techniques

M. Saqib Nawaz[1] · M. Zohaib Nawaz[2,3] · Osman Hasan[2] · Philippe Fournier-Viger[1] · Meng Sun[4]

Abstract

Interactive theorem provers (ITPs), also known as proof assistants, allow human users to write and verify formal proofs. The proof development process in ITPs can be a cumbersome and time-consuming activity due to manual user interactions. This makes proof guidance and proof automation the two most desired features for ITPs. In this paper, we first provide two evolutionary and heuristic-based proof searching approaches for the HOL4 proof assistant, where a genetic algorithm (GA) and simulated annealing (SA) is used to search and optimize the proofs in different HOL theories. In both approaches, random proof sequences are first generated from a population of frequently occurring HOL4 proof steps that are discovered with sequential pattern mining. Generated proof sequences are then evolved using GA operators (crossover and mutation) and by applying the annealing process of SA till their fitness match the fitness of the target proof sequences. Experiments were done to compare the performance of SA with that of GA. Results have shown that the two proof searching approaches can be used to efficiently evolve the random sequences to obtain the target sequences. However, these approaches lack the ability to learn the proof process, that is important for the prediction of new proof sequences. For this purpose, we propose to use a deep learning technique known as long short-term memory (LSTM). LSTM is trained on various HOL4 theories for proof learning and prediction. Experimental results suggest that combining evolutionary/heuristic and deep learning techniques with proof assistants can greatly facilitate proof finding/optimization and proof prediction.

Keywords HOL4 · Genetic algorithm · Simulated Annealing · LSTM · Proof searching · Proof learning · Fitness

1 Introduction

Theorem proving is a popular formal verification method that is used for the analysis of both hardware and software systems. In theorem proving, the system that needs to be analyzed is first modeled and specified in an appropriate mathematical logic. Important/critical properties of the system are then verified using *theorem provers* [1] based on deductive reasoning. The initial objective of developing theorem provers was to enable mathematicians to prove theorems using computer programs within a sound environment. However, these mechanical tools have evolved in last two decades and now play a critical part in the modeling and reasoning about complex and large-scale systems, particularly safety-critical systems. Today, theorem provers are used in verification projects that range from compilers, operating systems and hardware components to prove the correctness of large mathematical proofs such as the Kepler conjecture and the Feit-Thomson Theorem [2].

There are two main categories of theorem provers: Automated theorem provers (ATPs) and interactive theorem provers (ITPs). ATPs are generally based on propositional and first-order logic (FOL) and involve the development of computer programs that has the ability to automatically perform logical reasoning. However, FOL is less expressive in nature and cannot be used to define complex problems. Moreover, no ATPs can be scaled to the large mathematical libraries due to the problem of search space (combinatorial) explosion. On the other hand, ITPs are based on higher-order logic (HOL), which allows quantification over predicates and functions and thus offers rich logical formalisms such as (co)inductive and dependent types as well as recursive functions. This expressive power leads to the undecidability problem, i.e., the reasoning process cannot be

M. Saqib Nawaz and M. Zohaib Nawaz equally contributed to the paper.

✉ Philippe Fournier-Viger
philfv8@yahoo.com

Extended author information available on the last page of the article.

Published online: 30 September 2020

automated in HOL and requires some sort of human guidance during the process of proof searching and development. That is why ITPs are also known as *proof assistants*. Some well-known proof assistants are HOL4 [3], Coq [4] and PVS [5].

Most studies and efforts on designing proof assistants aim at making the proof checking process easier to users while ensuring proof correctness, and at providing an efficient proof development environment for users. The proof development process in ATPs is generally automatic. Whereas, ITPs follow the user driven proof development process. The user guides the proof process by providing the proof goal and by applying proof commands and tactics to prove the goal. Generally, an ITP user is involved in lots of repetitive work while verifying a nontrivial theorem (proof goal), and thus the overall process is quite laborious and tedious as well as time consuming. For example, a list of 100 mechanically verified mathematical theorems is available at [6]. The writing and verification process for many of these theorems required several months or even years (approximately 20 years for the Kepler conjecture proof in HOL Light [7] and twice as much for the Feit-Thompson theorem in Coq [8]) and the complete proofs contain thousands of low-level inference steps. Another example is the CompCert [9] compiler, that took 6 person-years and approximately 100,000 lines of Coq to write and verify.

The formal proof of a goal in ITPs, such as HOL4, mainly depends on the specifications available in a theory or a set of theories along with different combinations of proof commands, inference rules, intermediate states and tactics. Because a theory in ITPs often contains many definitions and theorems [10–12], it is quite inefficient to apply a brute force or pure random search based approach for proof searching. This makes proof guidance and proof automation along with proof searching an extremely desirable feature for ITPs.

However, proof scripts for theories in ITPs can be combined together to develop a more complex computer-understandable corpora. With the evolution in information and communication technologies (ICT) in the last decade, it is now possible to use deep mining and learning techniques on such corpora for guiding the proof search process, for proof automation and for the development of proof tactics/strategies, as indicated in the works done in [2, 13–17].

In previous work, we proposed an evolutionary approach [18] for proof searching and optimization in HOL4. The basic idea is to use a Genetic Algorithm (GA) for proof searching where an initial population (a set of potential solutions) is first created from frequent HOL proof steps that are discovered using a sequential pattern mining (SPM)-based proof guidance approach [17]. Random proof sequences from the population are then generated by applying two GA operators (crossover and mutation). Both operators randomly evolve the random proof sequences by shuffling and changing the proof steps at particular points. This process of crossover and mutation continues till the fitness of random proof sequences matches the fitness of original (target) proofs for the considered theorems/lemmas. Various crossover and mutation operators were used to compare their effect on the performance of GAs in proof searching. The approach was successfully used on six theories taken from the HOL4 library. This proof searching approach [18] is quite efficient in evolving random proofs efficiently. However, alternative proof searching approaches could be developed. Moreover, the approach is unable to learn the previously proved facts and to predict the proofs for new unproved theorems/lemmas (conjectures).

This paper addresses these issues by comparing evolutionary and heuristic-based proof searching approaches for HOL4 and proposing a deep neural networks based approach to learn the proof process and to predict proofs. The tasks of proof learning and prediction in theorem provers are mainly related to the task of *premise selection*, where relevant statements (formulas) that are important and useful to prove a conjecture are selected. This means that for a given dataset of already proved facts and a user provided conjecture, the problem of premise selection is to determine those facts that will lead to a successful proof of the conjecture. The proof searching approaches cannot be used to find a sequence of deductions that will lead from presumed facts to the given conjecture. The reason for this is that the state-space of the proof search is combinatorial in nature, where the search can quickly explode, despite the fact that some facts are often relevant for proving a given conjecture. On the other hand, the applicability of deep neural networks in embedding the semantic meaning and logical relationships into geometric spaces suggest their possible usage to guide the combinatorial search for the proofs in theorem provers. This paper extends the work published in [18] with the following contributions:

1. A heuristic-based approach is provided, where simulated annealing (SA) is used for proof searching and optimization. The performance of SA is compared with that of GA [18] for various parameter values. It is found that SA outperforms GA for proof finding and optimization. Moreover, we elaborate on how pure random searching and a brute force approach are not suitable for this task.
2. LSTM is used on HOL4 theories for learning and predicting proof sequences. The basic idea is to first convert the proofs in HOL4 libraries into a suitable format (vectors (tensors)) so that LSTM can process them. LSTM then learns from the proofs and predicts new proof sequences.

3. For experimental evaluation, we have selected 14 theories from the HOL4 library. All three approaches are used on these theories and the detailed results are presented in this paper.

The rest of this paper is organized as follows: Section 2 discusses the related work. Section 3 briefly discusses the HOL4 theorem prover, GAs, SA and recurrent neural networks (RNNs). Section 4 presents the proposed proof searching approaches, where GA and SA are used to find and optimize random HOL4 proofs. Evaluation of the proposed approaches on different theories of the HOL4 library is presented in Section 5. Section 6 presents the proof learning approach that is based on LSTM along with obtained results. Finally, Section 7 concludes the paper while highlighting some future directions.

2 Related work

In this section, we present the related work on the integration of evolutionary and heuristic algorithms in ITPs, followed by the use of deep learning techniques for the task of proof guidance and premise selection in theorem provers.

2.1 Evolutionary algorithms in ITPs

A GA was used in [19, 20] with the Coq proof assistant to automatically find formal proofs of theorems. However, the approach can only be used to successfully find the proofs of very small theorems that contain less number of proof steps. Whereas, for large and complex theorems that require induction and depend on the proofs of other lemmas, interaction between the proof assistant and the user is still required. Similarly, genetic programming [21] and a pairwise combination (that focuses only on crossover based approach) were used in [22] on patterns (simple tactics) discovered in Isabelle proofs to evolve them into compound tactics. However, Isabelle's proofs have to be represented using a linearized tree structure where the proofs are divided into separate sequences and weights are assigned to these sequences. Linearization of proofs tree leads to the loss of important connections (information) among various branches. Because of this, interesting patterns and tactics may be lost in the evolution process. In this work, the dataset for the proof sequences contains all the necessary information that is required for the discovery of frequent proof steps, through which initial population for the GA and SA is generated. Moreover, in both approaches, no human guidance is required during the evolution process for random proof sequences.

2.2 Deep learning in theorem provers

The task of premise selection with machine learning was first investigated in [23] in the ATP Vampire, where a corpus for Mizar proofs was constructed for training two classifiers with bag-of-word features that represent the terms occurrences in a vocabulary. Deep learning techniques (recurrent and convolutional neural networks) were first used in [15] for premise selection on the Mizar proofs (represented as FOL formulas) corpus. Experiments were done in the ATP E. Similarly, deep networks have been used in [24] for internal guidance in E. A hybrid (two-phase) approach was used on the Mizar proofs for the selection of clauses during proof search. However, the tree representation of proofs in first-order formulas was not invariant to variable renaming and was unable to capture the binding of variables by quantifiers. For large datasets, the HolStep dataset was introduced in [2], which consists of 2M statements and 10K conjectures. A deep learning approach was provided in [25], where formulas were represented as graphs that were then embedded into a vector space. Finally, formulas were classified by relevance. Experiments were performed on the HolStep dataset and obtained results showed better performance than sequence-based models.

GRU networks were used in [26] for guiding the proof search of a tableau style proof process in MetaMath. However, both approaches [25, 26] are tree-based approaches. Premise selection techniques were developed in [16] for the Coq system, where various machine learning methods were used and compared for learning the dependencies in proofs taken from the CoRN repository. Premise selection based on machine learning and automated reasoning for HOL4 is provided in [27] by adapting HOL(y)Hammer [28]. Tactictoe, a learning approach based on Monte Carlo tree search algorithm, was developed in [14] for HOL4 to automate theorems proofs.

Recently, RNNs and LSTMs were used in [29] to predict and suggest the tactics for the (semi)-automation of the proof process in one Coq theory. Compared with the mentioned approaches, our deep learning method does not rely on carefully engineered features and has the ability to learn from the textual representation of HOL4 proofs, without the need of converting the proofs to FOL formulas or representing them with graphs.

3 Preliminaries

A brief introduction to the HOL4 proof assistant, GA, SA and RNN is provided in this section to facilitate the understanding of the rest of the paper.

3.1 HOL4

HOL4 [3] is an ITP that utilizes the simple type theory along with Hindley-Milner polymorphism for the implementation of HOL. The logic in HOL4 is represented with the strongly-typed functional programming language meta language (ML). An ML abstract data type is used to define theorems and the only way to interact with the theorem prover is by executing ML procedures that operate on values of these data types. A theory in HOL4 is a collection of definitions, axioms, types, constants theorems and proofs, as well as tactics. The theories in HOL4 are stored as ML files. Users can reload a theory into the system and can utilize the corresponding definitions and theorems right away. Here, we provide a simple example of the factorial function as a case study. Some more details on HOL4 are discussed with the case study to facilitate the readers understanding.

The factorial, denoted as !, of a positive integer n returns the product of all positive integers that are less than or equal to n. Mathematically, factorial for n can be defined as:

$$n! = \begin{cases} 1 & n = 0 \\ n \times (n-1)! & n > 0 \end{cases}$$

For example, factorial of $6! = 6 \times 5 \times 4 \times 3 \times 2 \times 1 = 720$. Whereas 0! equal 1. This definition is recursive as a factorial is defined in terms of another factorial. In HOL4, factorial function can be specified recursively as:

```
val factorial_def = Define
    `(factorial 0 = 1) /\
      (factorial(SUC n) = (SUC n)
      *(factorial n))`;
```

The term *SUC n* in the HOL4 specification for factorial function represents successor of n, i.e., $n + 1$. Recursive definitions in HOL4 generally consist of two parts: (1) a base case, and (2) a recursive case. For the factorial function, the base case is *factorial 0 = 1*, which directly tells us the answer. The recursive case is *factorial(n+1) = (n+1)*(factorial n)*. The recursive case does not provide us the answer, but defines how to construct the answer for the factorial of $(n + 1)$ on the basis of the answer of factorial of n.

The base and the recursive cases are connected with a conjunction ($\wedge$). To make the example more understandable, the factorial function calculates the factorial of 3 as follows: *factorial 3 = 3 * factorial 2 = 3 * (2 * factorial 1) = 3 * (2 * (1 * factorial 0)) = 3 * (2 * (1 * 1))) = 6*. In HOL4, a new theorem is given to the HOL proof manager via *g*, which starts a fresh goalstack. For example: *g'!(n:num). (0 < factorial n)'*. This proof goal represents the property that for any natural number n, the factorial function will always be greater than 0, i.e., $\forall n.\ (0 <$ factorial $n)$.

Two types of interactive proof methods are supported in HOL4: forward and backward. In forward proof, the user starts with previously proved theorems and applies inference rules to get the proof of new theorem. A backward (also called goal directed proof) method is the reverse of the forward proof method. It is based on the concept of a tactic; which is an ML function that divides the main goal into simple sub-goals. In this method, the user starts with the desired theorem (a main goal) and specifies tactics to reduce it to simpler intermediate sub-goals. The above steps are repeated for the remaining intermediate goals until the user is left with no more sub-goals and this concludes the proof for the desired theorem. The theorem for factorial is proved using the backward method. The first step, in most of the cases, is to remove the quantifications using *GEN_TAC*. Induction is then applied which divides the proof goal (theorem) into two subgoals corresponding to the base case and recursive case. The base case is proved simply by using the *RW_TAC* proof command with the definition of $factorial$ function. The *RW_TAC* command rewrites and simplifies the goal with respect to the $factorial$ definition and then uses first order reasoning to automatically prove the base case. Whereas, the recursive case is proved with commands: *RW_TAC*, and *MATCH_MP_TAC "LESS_MULT2"*. Note that the *MATCH_MP_TAC* proof command matches the proof goal with the conclusion of the theorem given as its argument and based on the given theorem, generates the assumptions under which the proof goal would be true. The complete proof steps for the theorem are as follows:

```
e (GEN_TAC);
e (Induct_on `n`);
e (RW_TAC std_ss [factorial_def]);
e (RW_TAC std_ss [factorial_def]);
e (MATCH_MP_TAC LESS_MULT2);
e (RW_TAC std_ss []);
```

3.2 Genetic algorithms

GAs [30, 31] are based on Darwin's theory (survival of the fittest) and biological evolution principles. GAs can explore a huge search space (population) to find near optimal solutions to difficult problems that one may not otherwise find in a lifetime. The foremost steps of a GA include: (1) population generation, (2) selection of candidate solutions from a population, (3) crossover and (4) mutation. Candidate solutions in a population are known as chromosomes or individuals, which are typically finite sequences or strings $(x = x_1, x_2 \ldots, x_n)$. Each x_i (genes) refers to a particular characteristics of the chromosome. For a specific problem, GA starts by randomly generating a set of chromosomes to form a population and evaluates these chromosomes

using a fitness function f. The function takes as parameter a chromosome and returns a score indicating how good the solution is. Besides optimization problems, GAs are now used in many other fields and systems, such as bioinformatics, control engineering, scheduling applications, artificial intelligence, robotics and safety-critical systems.

The GA crossover operator is used to guide the search toward the best solutions. It combines two selected chromosomes to yield potentially better chromosomes. The two selected chromosomes are called parent chromosomes and the new chromosomes obtained by crossover are named child chromosomes. If an appropriate crossing point is chosen, then the combination of sub-chromosomes from parent chromosomes may produce better child chromosomes. The mutation operator applies some random changes to one or more genes. This may transform a solution into a better solution. The main purpose of this operator is to introduce and preserve diversity of the population so that a GA can avoid local minima. Both crossover and mutation operators play a critical part in the success of GAs [32].

3.3 Simulated annealing

Simulated annealing (SA) [33, 34] is a simple, probabilistic based well-known metaheuristic method to solve black box global optimization problems. It is based on the analogy of physical annealing, which is the process of heating and then slowly cooling a metal to obtain a strong crystalline.

The SA starts by creating a random initial solution. In each iteration of the SA, the current solution is replaced by a random "neighbor" solution. The neighbor solution is selected with a probability that depends on the difference between the corresponding function values and on a global parameter T (called the temperature). The value of T is decreased gradually in each iteration. Note that the process of finding the neighbor solution in SA and mutation operator of GA are very similar to each other.

The original suggestion for SA was to start the search from a high temperature and reduce it to the end of a process [35]. However, the cooling rate (called α) and the initial value of T are usually different for different problems and are generally selected empirically. The four main components that need to be defined when applying SA to solve any problem are: (1) problem configuration, (2) neighborhood configuration, (3) objective function and (4) cooling/annealing schedule.

For optimization problems, SA is faster than GA because it finds the optimal solution with point-by-point iteration rather than a search over a population of solutions. SA is very similar to the Hill-Climbing algorithm with one main difference: at high temperature, SA switches to a worse neighbor, which avoids SA to get stuck in a local optima.

3.4 Recurrent neural networks

Artificial neural network (ANN) [36] mimics the behavior of the human brain's neural structure to solve tasks such as decision-making, prediction, classification and visualization etc. in various domains, such as natural language processing (NLP) and speech recognition. The traditional neural network (also called feedforward neural network) is a type of ANN that allows information to travel only one way: from input layer(s) through hidden layer(s) to output layer(s). There are no cycles/loops (feedback) in such networks, which means that the output of any layer has no effect on that layer or any other layer. On the other hand, recurrent neural networks (RNN) [37] allow previous output(s) to be used as input(s) while having hidden layers. The output of an RNN depends not only on the present inputs but also on the neurons' previous states. Therefore, RNN has a "memory" that stores all the computed information. The architecture of a traditional RNN and its unrolled version is shown in Fig. 1, where at timestep t, x_t represents the input, hs_t (memory) is the hidden state and y_t is the output. The value of hs_t is calculated based on the input at the current step: $hs_t = f(Ux_t + Whs_{t-1})$, where the activation function f shows a nonlinearity such as *tanh* or *RelU*. With hs_t, the output y_t can be calculated as: $y_t = (Vhs_t)$.

A RNN basically contains multiple copies of the same network, where each network passes the information to the successor. Therefore, RNN can be viewed as many copies of a traditional neural network that are executed in a chain. One major drawback in RNNs is that their "memory" is unable to store longer term dependencies (retaining information from a long time ago) due to the vanishing gradient problem, which is a well-known issue in gradient descent and back-propagation. Moreover, RNN training is a difficult task as they cannot process very long sequences with activation function such as *tanh* or *RelU*. Different RNN variants are now available that aim to solve the vanishing gradient problem, such as the long short-term memory (LSTM)

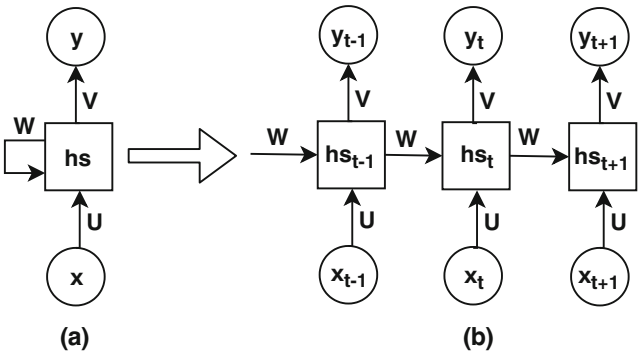


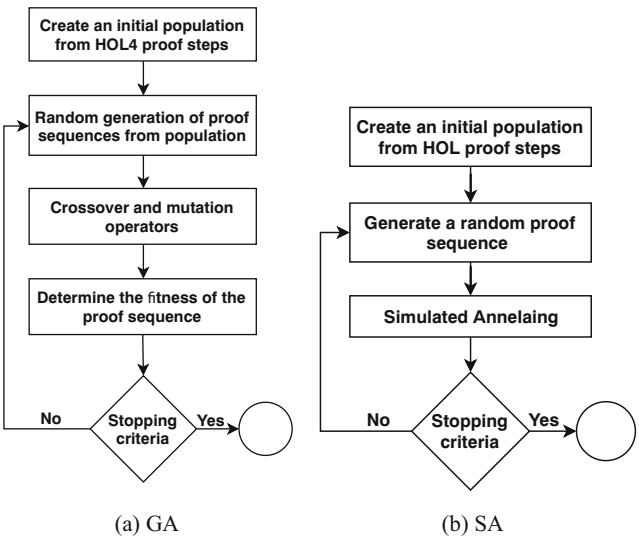
Fig. 1 **a** A simple RNN, **b** Unrolled version of RNN

network [38]. In this paper, we propose to use LSTM for the considered problem and more details about LSTM are provided in Section 6.

4 Proof searching approaches

The proposed structure (flowchart) of the evolutionary and heuristic-based approaches that is used to find and optimize the proofs of theorems/lemmas in HOL4 theories is shown in Fig. 2.

The proof development process in HOL4 is interactive in nature and it follows the *lambda calculus proof representation*. Proofs in HOL4 are constructed with an interactive *g*oal stack and then put together using the ML function *p*rove. A user first provides the property (in the form of a lemma or theorem) that is called a proof goal. A proof goal is a sequent that constitutes a set of assumptions and conclusion(s) as HOL formulas. Then the user applies proof commands and tactics to solve the proof goal. A tactic is basically a function that takes a proof goal and returns a sub-goals together with a validation function. The action resulting from the application of a proof command and tactics is referred to as a HOL4 proof step (*HPS*). A *HPS* may either prove the goal or generate another proof goal or divide the main goal into sub-goals. The proof development process for a theorem or lemma is completed when the main goal or all the sub-goals are discharged from the goal stack. The fact that the proof script of a particular theorem or lemma depends on the application of the *HPS* in a specific order makes automatic proof search for a goal quite challenging. However, evolutionary and heuristic algorithms have the potential to search for the proofs of theorems/lemmas due to their ability to handle black-box search and optimization problems.



(a) GA (b) SA

Fig. 2 Flowchart of proof searching approaches in HOL4

We propose to convert the data available in HOL4 proof files to a proper computational format so that a GA and SA can be used. Moreover, the redundant information (related to HOL4) that plays no part in proof searching and evolution is removed from the proof files. The complete proof for a goal (theorem/lemma) can now be considered as a sequence of *HPS*. Let $PS = \{HPS_1, HPS_2, \ldots HPS_m\}$ represent the set of HPS proof steps. A *proof step set* PSS is a set of *HPS*, that is $PSS \subseteq PS$. Let the notation $|PSS|$ denote the set cardinality. PSS has a length n (called *n-PSS*) if it contains n proof commands, i.e., $|PSS| = n$. For example, consider that PS = {*RW, PROVE_TAC, FULL_SIMP_TAC, REPEAT_GEN_TAC, DISCH_TAC*}. Now, the set {*RW, FULL_SIMP_TAC, REPEAT_GEN_ TAC*} is a proof step set that contains three proof steps. A proof sequence is a list of proof step sets $S = \langle PSS_1, PSS_2, ..., PSS_n \rangle$, such that $PSS_i \subseteq PSS$ $(1 \leq i \leq n)$. For example, $\langle$ {*RW, PROVE_TAC*}, {*FULL_SIMP_TAC*}, {*GEN_TAC, DISCH_TAC*}$\rangle$ is a proof sequence, which has three *PSS* and five *HPS* that are used to prove a theorem/lemma.

A *proof dataset PD* is a list of proof sequences $PD = \langle S_1, S_2, ..., S_p \rangle$, where each sequence has an identifier (ID). For example, Table 1 shows a *PD* that has five proof sequences.

4.1 Proposed genetic algorithm (GA)

Algorithm 1 presents the pseudocode of the proposed GA that is used to find the proofs in the HOL4 theories. It contains the *HPS* used for the verification of theorems and lemmas in the considered theories.

Algorithm 1 Flow of the GA.

Input: *FHPS*: Frequent HOL proof steps, *PD*: proof sequences database
Output: Generated proof sequences

```
1:  Pop ← FHPS
2:  for each P ∈ PD do
3:      P1 ← randomseq(Pop, length(P))
4:      P2 ← randomseq(Pop, length(P))                ▷ P1 ≠ P2
5:      repeat
6:          C ← Crossover(P1, P2, P)
7:          Child ← Mutation(C)
8:          if Fitness(Child, P) < Fitness(P, P) then
9:              repeat
10:         else
11:             bFitness ← Fitness(Child, P)
12:             bChild ← Child
13:         end if
14:     until (Fitness(Child, P) = Fitness(P, P))
15:     return bFitness, bChild
16: end for
```

Table 1 A sample proof dataset

ID	Proof Sequence
1	⟨{*GEN_TAC, CONJ_TAC, MP_TAC*}⟩
2	⟨{*GEN_TAC, X_GEN_TAC, PROVE_TAC*}⟩
3	⟨{*RW, PROVE_TAC, CONJ_TAC, MAP_EVERYTHING_TAC, AP_TERM_TAC*}⟩
4	⟨{*GEN_TAC, SUBGOAL_THEN, DISCH_TAC, CASES_ON, AP_TERM_TAC, BETA_TAC, CASES_TAC*}⟩
5	⟨{*RW_TAC, SUBST1_TAC, Q.SUBGOAL_THEN, SRW_TAC, FULL_SIMP_TAC*}⟩

An initial population (*Pop*) is first created from frequent *HPS* (*FHPS*) that are discovered with various sequential pattern mining (SPM) techniques [39]. Based on this initial population, two random proof sequences (P_1 and P_2) are generated and are passed through the crossover operation where the child proof sequences are generated and their fitness is evaluated. The mutation operation is applied to the child having the better fitness value to generate the mutated child sequence.

If a mutated child's fitness is equal to the fitness of the target proof sequence from *PD*, then the mutated child is returned as the final proof sequence. The process of crossover and mutation continues until randomly generated proof sequences exactly match the proof sequences from the *PD*. The fitness values guide the GA toward the best solution(s) (proof sequences). Here the fitness value represents the total number of *HPS* in the random proof sequence that match the *HPS* in the position of the original (target) proof sequence. Algorithm 2 presents the procedure for calculating the fitness value of a proof sequence.

Algorithm 2 Fitness.

Input: *Pseq*: A proof sequence, *P*: The current target proof sequence
Output: Integer that represents the fitness of a proof sequence (*Pseq*)

```
1:  procedure FITNESS(Pseq, P)
2:      i, f ← 0
3:      while (i ≤ length(Pseq) - 1) do
4:          if (Pseq[i] = P[i]) then
5:              f ← f + 1
6:          end if
7:          i ← i + 1
8:      end while
9:      return f
10: end procedure
```

In each generation, the priority of the randomly generated proof sequence is ranked according to the fitness values calculated based on the above mentioned fitness procedure. This procedure evaluates how close a given solution is to the optimum solution (in our case, the target solution). It compares each gene i of a random proof sequence (*Pseq*) with the genes of the target (*P*). The fitness of *PSeq* is set to 0, and increased by 1 for each matching gene and if the genes in both sequences are equal then the fitness of 1 is assigned. For example, consider the following random proof sequence (*RP*) and the target sequence (*TP*):

$$\begin{aligned} RP = {} & \mathit{MAP\_EVERYTHING\_TAC},\ \mathit{RULE\_ASSUM\_TAC},\\ & \mathbf{X\_GEN\_TAC},\ \mathit{SRW\_TAC},\ \mathbf{AP\_TERM\_TAC},\\ & \mathit{DISCH\_TAC},\ \mathit{DECIDE\_TAC},\ \mathbf{RW\_TAC}\\ TP = {} & \mathit{POP\_ASSUM},\ \mathit{REAL\_ARITH\_TAC},\ \mathbf{X\_GEN\_TAC},\\ & \mathit{COND\_CASES\_TAC},\ \mathbf{AP\_TERM\_TAC},\\ & \mathit{RULE\_ASSUM\_TAC},\ \mathit{X\_GEN\_TAC},\ \mathbf{RW\_TAC} \end{aligned}$$

The *Fitness* procedure returns 3 as three *HPS* are the same in both sequences (at Positions 3, 5 and 8, respectively).

Algorithm 3, 4 and 5 present the pseudocode of the three crossover operators. The symbol **o** in these algorithms represents the concatenation. These three crossover procedures are explained with simple examples. Let P_1 and P_2 be:

$$\begin{aligned} P_1 = {} & \mathit{SRW\_TAC},\ \mathit{MAP\_EVERYTHING\_TAC},\ \mathit{X\_GEN\_TAC},\\ & \mathit{AP\_TERM\_TAC},\ \mathit{RULE\_ASSUM\_TAC},\ \mathit{DISCH\_TAC},\\ & \mathit{DECIDE\_TAC},\ \mathit{RW\_TAC}\\ P_2 = {} & \mathit{REAL\_ARITH\_TAC},\ \mathit{POP\_ASSUM},\ \mathit{X\_GEN\_TAC},\\ & \mathit{COND\_CASES\_TAC},\ \mathit{RW\_TAC},\\ & \mathit{RULE\_ASSUM\_TAC},\ \mathit{X\_GEN\_TAC},\ \mathit{AP\_TERM\_TAC} \end{aligned}$$

Let n represents the length of both proof sequences and let position cp ($1 \leq cp \leq n$) be chosen randomly as crossing point in both proof sequences. Single point crossover (*SPC*) produces the following proof sequences for $cp = 4$:

$$\begin{aligned} P_1' = {} & \mathit{SRW\_TAC},\ \mathit{MAP\_EVERYTHING\_TAC},\ \mathit{X\_GEN\_TAC},\\ & \mathit{COND\_CASES\_TAC},\ \mathit{RW\_TAC},\ \mathit{RULE\_ASSUM\_TAC},\\ & \mathit{X\_GEN\_TAC},\ \mathit{AP\_TERM\_TAC}\\ P_2' = {} & \mathit{REAL\_ARITH\_TAC},\ \mathit{POP\_ASSUM},\ \mathit{X\_GEN\_TAC},\\ & \mathit{AP\_TERM\_TAC},\ \mathit{RULE\_ASSUM\_TAC},\ \mathit{DISCH\_TAC},\\ & \mathit{DECIDE\_TAC}, \mathit{RW\_TAC} \end{aligned}$$

Fitness of newly generated sequences are checked last and SPC returns the proof sequence having the highest fitness.

Algorithm 3 Single point crossover.

Input: P_1, P_2: Two proof sequences, P: The current target proof sequence
Output: Child proof sequence

```
1:  procedure SPC(P1, P2, P)
2:      size ← min(length(P1), length(P2))
3:      cp ← randomint(1, size)                    ▷ (1 ≤ cp ≤ size)
4:      P1 ← P1[1, cp] o P2[cp + 1, length(P2)]
5:      P2 ← P2[1, cp] o P1[cp + 1, length(P1)]
6:      if (Fitness(P1, P) > Fitness(P2, P) then
7:          return P1
8:      else
9:          return P2
10:     end if
11: end procedure
```

Two crossing points are selected by the multi point crossover (MPC) operator. Let cp_1 and cp_2 represent two crossing points ($cp_1 < cp_2 \leq n$). For P_1 and P_2, the new proof sequences generated for $cp_1 = 4$ and $cp_2 = 5$ are:

$$P_1' = \textit{SRW\_TAC},\ \textit{MAP\_EVERYTHING\_TAC},\ \textit{X\_GEN\_TAC},\ \textit{COND\_CASES\_TAC},\ \textit{RW\_TAC},\ \textit{DISCH\_TAC},\ \textit{DECIDE\_TAC},\ \textit{RW\_TAC}$$

$$P_2' = \textit{REAL\_AIRTH\_TAC},\ \textit{POP\_ASSUM},\ \textit{X\_GEN\_TAC},\ \textit{AP\_TERM\_TAC},\ \textit{RULE\_ASSUM\_TAC},\ \textit{RULE\_ASSUM\_TAC},\ \textit{X\_GEN\_TAC},\ \textit{AP\_TERM\_TAC}$$

Newly generated sequences are evaluated last and MPC returns the proof sequence having the highest fitness.

In uniform crossover (UC), each element (gene) of the proof sequences is assigned to the child sequences with a probability value p. UC evaluates each gene in the proof sequences and selects the value from one of the proof sequences with the probability p. If p is 0.5, then the child has approximately half of the genes from the first proof sequence and the other half from the second proof sequence. For P_1 and P_2, some newly generated proof sequences after UC with $p = 0.5$ are:

$$P_1' = \textit{SRW\_TAC},\ \textit{POP\_ASSUM},\ \textit{X\_GEN\_TAC},\ \textit{COND\_CASES\_TAC},\ \textit{RULE\_ASSUM\_TAC},\ \textit{DISCH\_TAC},\ \textit{X\_GEN\_TAC},\ \textit{RW\_TAC}$$

$$P_2' = \textit{REAL\_ARITH\_TAC},\ \textit{MAP\_EVERYTHING\_TAC},\ \textit{X\_GEN\_TAC},\ \textit{AP\_TERM\_TAC},\ \textit{RW\_TAC},\ \textit{RULE\_ASSUM\_TAC},\ \textit{DECIDE\_TAC},\ \textit{AP\_TERM\_TAC}$$

Because UC is a randomized algorithm, depending on the selection probability, the generated child proof sequences can be different. Fitness of newly generated sequences is then checked and the UC returns the sequence having the highest fitness.

Algorithm 4 Multi point crossover.

Input: P_1, P_2: Two proof sequences, P: The current target proof sequence
Output: Child proof sequence

```
1:  procedure MPC(P1, P2, P)
2:      size ← min(length(P1), length(P2))
3:      cp1 ← randomint(1,size)
4:      cp2 ← randomint(1,size)
5:      if cp2 > cp1 then
6:          cp2 ← cp2 + 1
7:      else
8:          cp2 ← cp1
9:          cp1 ← cp2
10:     end if
11:     P1 ← P1[1, cp1] o P2[cp1 + 1, cp2] o P1[cp2+
        1,length(P1)]
12:     P2 ← P2[1, cp1] o P1[cp1 + 1, cp2] o P2[cp2+
        1,length(P2)]
13:     if (Fitness(P1, P) > Fitness(P2, P) then
14:         return P1
15:     else
16:         return P2
17:     end if
18: end procedure
```

Algorithm 5 Uniform crossover.

Input: P_1, P_2: Two proof sequences, P: The current target proof sequence
Output: Child proof sequence

```
1:  procedure UC(P1, P2, P)
2:      size ← min(length(P1), length(P2))
3:      p ← 0.5
4:      for i in range(size) do
5:          if unifromreal[0,1] ≤ p then
6:              P1[i] ← P2[i]
7:              P2[i] ← P1[i]
8:          end if
9:      end for
10:     if (Fitness(P1, P) > Fitness(P2, P) then
11:         return P1
12:     else
13:         return P2
14:     end if
15: end procedure
```

The mutation operation is applied after the crossover operation. The standard mutation (SM) operator of GAs adds random information to the search process, so that it does not get stuck in a local optima. In SM, the selected location value is changed from its original value with some probability, called mutation probability, and is denoted as

p_m. For a proof sequence, a randomly chosen genes value i is replaced by a random *HPS* from the current population *Pop*. For example, a mutation of the proof sequence P_1 is:

$$P_1' = \textit{SRW\_TAC},\ \textit{POP\_ASSUM},\ \textit{X\_GEN\_TAC},\ \textbf{DECIDE\_TAC},\ \textit{RULE\_ASSUM\_TAC},\ \textit{DISCH\_TAC},\ \textit{X\_GEN\_TAC},\ \textit{RW\_TAC}$$

Algorithm 6 Standard mutation.

Input: P_1: A proof sequence
Output: Mutated child proof sequence

```
1: procedure SM(P1)
2:     ind ← randomint(1, length(P1))
3:     alter ← randomsample(Pop, 1)          ▷ (1-length proof sequence form Pop)
4:     P1[ind] ← alter                       ▷ (P1[ind] ≠ alter)
5:     return P1
6: end procedure
```

The pairwise interchange mutation (*PIM*) operator selects and interchanges two arbitrary genes from a proof sequence. But for proof searching, we empirically observed that a GA could not find the target proof sequence with *PIM* as it was only interchanging the values between two genes in the random proof sequence. To address this issue, we revised the *PIM* procedure such that the two selected gene values are replaced by random *HPS* from the population rather than interchanging the values. For instance, by applying modified *PIM* on the proof sequence P_1, the following mutated proof sequence can be obtained:

$$P_1' = \textit{SRW\_TAC},\ \textbf{REWRITE\_TAC},\ \textit{X\_GEN\_TAC},\ \textit{DECIDE\_TAC},\ \textit{RULE\_ASSUM\_TAC},\ \textbf{BETA\_TAC},\ \textit{X\_GEN\_TAC},\ \textit{RW\_TAC}$$

Algorithm 7 Modified pairwise interchange mutation.

Input: P_1: A proof sequence
Output: Mutated child proof sequence

```
1: procedure MPIM(P1)
2:     mp1 ← randomint(1, length(P1))
3:     mp2 ← randomint(1, length(P1))        ▷ mp1 ≠ mp2
4:     ng, alter ← randomsample(Pop, 2)
5:     P1[mp1] ← ng                          ▷ (P1[mp1] ≠ ng)
6:     P1[mp2] ← alter                       ▷ (P1[mp2] ≠ alter)
7:     return P1
8: end procedure
```

The reason to use more than one crossover and mutation operators is to investigate their effect on the overall performance of the GA in proof searching. It is important to point out that in each generation, a random proof sequence goes through crossover and mutation operation with a probability of 1 to reduce the number of iterations performed by the GA.

4.2 Simulated annealing (SA)

Algorithm 8 presents the proposed pseudocode of the SA that is used to find the proofs in HOL4 theories.

Algorithm 8 Flow of the SA.

Input: *FHPS*: Frequent HOL4 proof steps, *PD*: proof sequences database, Temp, Temp_min, α
Output: Generated proof sequences

```
1: Pop ← FHPS
2: for each P ∈ PD do
3:     OF ← Fitness(P, P)
4:     PS ← randomseq(Pop, length(P))
5:     BF ← Fitness(PS, P)
6:     if BF ≥ OF then
7:         return PS
8:     end if
9:     while (Temp > Temp_min) do
10:        NS ← get_neighbor(PS)
11:        NF ← Fitness(NS, P)
12:        if NF == OF then
13:            return NS
14:        end if
15:        if NF > BF then
16:            PS ← NS
17:            BF ← NF
18:        end if
19:        ar ← exp(T/(1+T))
20:        if ar > randomuniform(0, 10) then
21:            PS ← NS
22:            BF ← NF
23:        end if
24:        Temp ← Temp × α
25:    end while
26:    return PS
27: end for
```

Just like GA, an initial population (*Pop*) is first created from *FHPS*. From this population, a random proof sequence (PS) is then generated that is passed through the annealing process (Steps 9-25 in Algorithm 8), where it is evolved until its fitness is equal to the fitness of the target proof sequence from *PD*. Besides annealing, the algorithm consists of two main procedures, *Fitness* and *Get_Neighbor* (GN), which are explained next.

Fitness values guide the SA toward the best solution(s) (proof sequences). Here the fitness value is the total number of *HPS* in the random proof sequence that matches the *HPS* in the position of the original (target) proof sequence. Algorithm 2 (from Section 4.1) presents the procedure for calculating the fitness value of a proof sequence.

In the annealing process, a neighbor random sequence is first generated. Algorithm 9 presents the procedure for getting the neighbor solution. The selected location value is changed from its original value in the *Get_Neighbor*. For a proof sequence, a randomly chosen genes value i is replaced by a random *HPS* from the current population *Pop*. It is important to point out here that the standard mutation operator of GA and the get neighbor procedure in SA are quite similar.

Algorithm 9 Get Neighbor.

Input: P_1: A proof sequence
Output: A neighbor proof sequence

1: **procedure** GN(P_1)
2: $ind \leftarrow$ randomint(1, length(P_1))
3: $alter \leftarrow$ randomsample(Pop, 1) ▷ (*1-length* proof sequence form Pop)
4: $P_1[ind] \leftarrow alter$ ▷ ($P_1[ind] \neq$ alter)
5: return P_1
6: **end procedure**

After the *Get_Neighbor* procedure, the fitness of the neighbor solution is calculated. The randomly generated proof sequence and the neighbor sequence is then compared. If the fitness of the neighbor is better, then it is selected. Otherwise, an acceptance rate (Step 19 in Algorithm 8) is used to select one out of the two sequences. The acceptance rate depends on temperature. Finally, the temperature *Temp* is decreased with the following formula:

$$\text{Temp} = \text{Temp} \times \alpha$$

where the value of α is in the range of $0.8 < \alpha < 0.99$. The process of annealing is repeated (Steps 9-24 in Algorithm 8) until the random proof sequence fitness matches with the target proof sequence or *Temp* reaches the minimum value (*Temp_min*). In our case, we set the value of *Temp* such that the SA always terminates when the random proof sequence matches with the target proof sequence. The process that distinguishes SA from GA is the annealing process.

Besides the annealing process, another main concept in SA is the acceptance probability. SA checks whether the new solution is better than the previous solution. If the new solution is worse than the present solution, it may still pick the new solution with some probability known as the *acceptance probability* that governs whether to switch to the worst solution or not. This way, we may avoid the local optimum by exploring other solutions. Now this may seem worse or unacceptable at present, but it could lead SA to the global optimum. For this purpose, we chose the *acceptance probability* by using acceptance rate (AR) formula as:

$$\text{AR} = exp\left(\frac{T}{1+T}\right)$$

where T is the current temperature. We performed experiments to check the effect of AR on the performance of SA for proof searching. From the simulation results presented in the next section, it was observed that this parameter effects the performance of SA, but it is negligible.

5 GA and SA based results and discussion

The proposed GA and SA algorithms, described in the previous section, are implemented in Python and the code can be found at [40]. To evaluate the proposed approaches, experiments were carried on a fifth generation Core $i5$ processor and 8 GB of RAM. Some initial and important results obtained by applying the proposed GA and SA based approaches on *PD* are discussed in this section.

We first investigated the performance of the proposed GA for finding the proofs of theorems in 14 HOL4 theories available in its library. These theories are: *Transcendental, Arithmetic, RichList, Number, Sort, Bool, BinaryWords, FiniteMap, InductionType, Combinator, Coder, Encoder, Decoder* and *Rational*. We selected five to twenty theorems/lemmas from each theory and in total, we have proof sequences for 300 theorems/lemmas and 89 distinct *HPS* in the *PD*. Table 2 lists some of the important theorems/lemmas from the theories. For example, $L1$ (Lemma 1) from the transcendental theory proves the property for the exponential bound of a real number x. Similarly, $T2$ is the theorem for the positive value of sine when the given value is in the range $[0 - 2]$. $T10$ from the *R*ational theory is the dense theorem that proves that there exists a rational number between any two real numbers.

The GA was run with the different crossover and mutation operators on the considered theorems/lemmas ten times. Fitness values in Table 3 represent the total *HPS* that is used in the complete proof and this value is kept the same for respective theorems and lemmas in all crossover and mutations operators. The generations column shows how many times a random proof sequences goes through GA operators to reach the target proof sequence. The time column represents how much time (in seconds) is taken by the GA to find the complete proof for a theorem. We found that different crossover operators with the same mutation operator required almost the same number of generations to find the target proofs. However, with *MPIM* (Algorithm 7), the target proofs are found in less generations as compared to *SM* (Algorithm 6). It is important to point out that the probability in *UC* (Algorithm 5) has no noticeable effect on the average generation count of the GA. That is why we select the probability ($p = 0.5$) for *UC*.

Just like GA, we investigate the performance of the proposed SA for finding the proofs of theorems/lemmas in 14 HOL4 theories and the obtained results are listed in

Table 2 A sample of theorems/lemmas in six HOL4 theories

HOL Theory	No.	HOL4 Theorems
	L1	`⊢ ∀x. 0<=x∧xv<= inv(2) ==> exp(x) <= 1+2*x`
Transcendental	T1	`⊢ ∀ x. (\n. (^exp_ser) n (x pow n)) sums exp(x)`
	T2	`⊢ ∀ x. 0 < x∧ x < 2 ==> 0 < sin (x)`
Arithmetic	T3	`⊢ ∀n a b. 0 < n ==>((SUC a MOD n = SUC b MOD n) = ( a MOD n = b MOD n ))`
RichList	T4	`⊢ ∀m n. ((l:'a list). ((m + n)=(LENGTH l))==> ( APPEND ( FIRSTN n l ) ( LASTN m l ) = l)`
	T5	`⊢ ∀n m. ( m <= n ==> (iSUB T n m = n - m)) ∧ (m < n ==> (iSUB F n m = n - SUC m))`
Number		
	T6	`⊢ ∀ n a. 0 < onecount n a ∧ 0 < n ==> ( n = 2 EXP (onecount n a - a ) - 1 )`
Sort	T7	`⊢ (PERM L [x] = (L = [x]))∧(PERM [x] L = (L = [x]))`
	T8	`⊢ PERM = TC PERM_SINGLE_SWAP`
	T9	`⊢ ∀ x y. abs_rat ( frac_add ( rep_rat ( abs_rat x ) ) y ) = abs_rat ( frac_add x y )`
Rational		
	T10	`⊢ ∀ r1 r3. rat_les r1 r3 ==> ?rat_res r1 r2 ∧ rat_les r2 r3`

Table 4. The comparison of SA with GA for *T2* is shown in the second part of Table 4. For the GA, a different crossover operator has no great effect on the overall performance of the GA. However, using the MPIM operator allowed to find the target proof sequences considerably more quickly than using the SM operator. For *T2*, SA is found to be faster (30232 generations) than the GA with different crossover and mutation operators. For this particular example, SA is approximately sixty times faster than GA with different crossover operators and SM. Whereas, it is approximately ten times faster than the GA with different crossover operators and MPIM.

The average number of generations for the SA and GA with different crossover and mutation operators to reach the target proof sequences in the whole dataset are shown in Table 5. GA with different crossover and *MPIM* operators is approximately fourteen times faster than the GA with different crossover operators and *SM*. A possible explanation for this is that the *SM* changes the *HPS* at a single location of the sequence, while *MPIM* changes two locations. Thus, *MPIM* explores a more diverse solution as compared to *SM*. Whereas, SA is six times faster than GA with *MPIM* and different crossover operators. The main reasons for this is that in SA, only one procedure (GN) is called. On the other hand, in GA, two procedures (crossover and mutation) are called.

Population diversity greatly influences a GA's ability to pursue a fruitful exploration as it iterates from a generation to another [41]. The proof searching process with GA can be trapped in a local optima due to the loss of diversity through premature convergence of the *HPS* in the population. This makes the diversity maintenance and computation one of the fundamental issues for the GA. We studied population diversity with two measures. The first one being the standard deviation of fitness SD_f, whose values in the *Pop* of *HPS* is measured as:

$$SD_f = \sqrt{\frac{\sum_{i=1}^{N}(f_i - \bar{f})^2}{N-1}}$$

where N is the total number of proof sequences, f_i is the fitness of the ith proof sequence and $\bar{f}$ is the *mean* of the fitness values. As the fitness values for random proof sequences remain the same (after evolution) for all crossover and mutation operators, so SD_f is 14.12 with a mean of 12.05 for the GA. The second measure that is used to investigate the variability of *HPS* in *Pop* and the extent of deviation (dispersion) for the proof sequences as a whole is the standard deviation of time (SD_t), which is measured as:

$$SD_t = \sqrt{\frac{\sum_{i=1}^{N}(t_i - \bar{t})^2}{N-1}}$$

where t_i is the time taken by the GA to find the correct ith proof sequence and $\bar{t}$ is the *mean* of the time values.

Table 6 lists the calculated SD_t for all the proofs in the *PD* along with their mean for different crossover and mutation operators. A low *SD* indicates that the data (time values to find respective *HPS* in proof sequences) is less spread out and is clustered closely around the *mean* average values. Whereas a high *SD* means that the data is spread apart from the *mean*. *SM* is found to be approximately fourteen times slower than *MPIM*. That is why we have more time points for *SM* than *MPIM*, which makes the SD_t and the respective *mean* higher for *SM*.

Table 3 Results for the proposed GA

T/L	C* & M*	Fit**	Generations	Time(s)	C & M	Fit	Generations	Time (s)
L1	SPC/SM	54	1,903,765	55.43	SPC/MPIM	54	314,043	9.52
T1	SPC/SM	58	2,103,765	60.10	SPC/MPIM	58	334,043	10.33
T2	SPC/SM	81	1,947,597	93.56	SPC/MPIM	81	392,822	12.89
T3	SPC/SM	66	2,473,394	62.35	SPC/MPIM	66	191,162	6.61
T4	SPC/SM	19	297,179	4.72	SPC/MPIM	19	38,307	0.93
T5	SPC/SM	23	501,813	8.30	SPC/MPIM	23	33,655	0.71
T6	SPC/SM	30	709,484	13.09	SPC/MPIM	30	34,776	0.79
T7	SPC/SM	17	264,263	4.11	SPC/MPIM	17	21,136	0.40
T8	SPC/SM	42	811,951	28.49	SPC/MPIM	42	39,302	1.41
T9	SPC/SM	23	554,111	9.30	SPC/MPIM	23	45,309	0.90
T10	SPC/SM	23	546,136	9.21	SPC/MPIM	23	51,552	1.01
L1	MPC/SM	54	1,488,005	27.21	MPC/MPIM	54	105,521	3.29
T1	MPC/SM	58	1,540,467	35.93	MPC/MPIM	58	153,644	5.01
T2	MPC/SM	81	1,898,305	80.38	MPC/MPIM	81	191,699	7.69
T3	MPC/SM	66	1,128,636	31.54	MPC/MPIM	66	104,784	3.60
T4	MPC/SM	19	358,182	7.01	MPC/MPIM	19	24,960	0.48
T5	MPC/SM	23	384,539	7.19	MPC/MPIM	23	42,750	0.83
T6	MPC/SM	30	738,037	10.21	MPC/MPIM	30	73,408	1.13
T7	MPC/SM	17	276,087	5.32	MPC/MPIM	17	19,997	0.43
T8	MPC/SM	42	1,245,801	25.67	MPC/MPIM	42	101,795	2.52
T9	MPC/SM	23	411,625	7.73	MPC/MPIM	23	275,78	0.63
T10	MPC/SM	23	480,625	8.26	MPC/MPIM	23	25,314	0.55
L1	UC/SM	54	1,652,013	61.83	UC/MPIM	54	63,277	1.86
T1	UC/SM	58	1,682,200	68.32	UC/MPIM	58	126,097	2.92
T2	UC/SM	81	2,348,878	101.63	UC/MPIM	81	312,328	8.21
T3	UC/SM	66	1,662,751	44.81	UC/MPIM	66	257,215	7.48
T4	UC/SM	19	706,950	11.12	UC/MPIM	19	20,702	0.41
T5	UC/SM	23	819,903	14.97	UC/MPIM	23	71,614	1.37
T6	UC/SM	30	867,183	17.21	UC/MPIM	30	74,635	1.53
T7	UC/SM	17	321,183	6.16	UC/MPIM	17	20,263	0.42
T8	UC/SM	42	804,969	20.53	UC/MPIM	42	29,606	0.95
T9	UC/SM	23	625,908	11.38	UC/MPIM	23	130,303	2.50
T10	UC/SM	23	716,950	13.07	UC/MPIM	23	90,425	1.94

* Crossover and mutation ** Fitness

We also checked the amount of memory used by GA (shown in Table 6) while searching for proofs. Moreover, we noticed that the GA using different crossover and mutation operators requires approximately the same memory while searching for proofs and their optimization in *PD*.

With *acceptance probability*, SA may accept a new solution obtained with the *GN* procedure that is worst than the present solution. The reason for this is that there is always a possibility that the worst solution could lead the SA to the global optimum. In our proposed SA, we chose the *acceptance probability* with the *AR* formula ($AR = exp(\frac{T}{1+T})$). This *AR* is then compared with a random number generated within the range (2.71820060604849, 2.71825464604849).

The range is selected after experimenting with the following values: $Temp = 100000.0$, $Temp\_min = 0.00001$, and $\alpha = 0.99954001$. If the value of *AR* is greater than the random number generated within the above range, then the worse solution is going to be picked. We also used a counter named acceptance rate counter (*ARC*) that counts how many times the worst solution is picked. By simulation, we came to know that this factor does not play any significant role in the overall generation count or time. This is because of the fact that in our case, we do not have any local optimum.

Table 4 Results for SA and comparison with GA

T/L	Fitness	Generations	Time (s)
L1	54	17,636	0.56
T1	58	21,144	0.82
T2	81	30,232	1.11
T3	66	22,919	0.65
T4	19	3,924	0.10
T5	23	5,057	0.09
T6	30	4,370	0.08
T7	17	1,892	0.02
T8	42	16,767	0.32
T9	23	7,734	0.16
T10	23	6,997	0.14
T2	GA(SPC/SM)	1,947,597	93.56
T2	GA(MPC/SM)	1,898,305	80.38
T2	GA(UC/SM)	2,348,878	101.63
T2	GA(SPC/MPIM)	392,822	12.89
T2	GA(MPC/MPIM	191,699	7.69
T2	GA(UC/MPIM)	312,328	8.21

SPC = single point crossover, MPC = multi point crossover, UC = uniform crossover, SM = standard mutation, MPIM = modified pairwise interchange mutation

SA in our case tends to find only one global solution for each random proof sequence based on the fitness value. Table 7 shows that the average generation count of SA for all theorems/lemmas in the *PD*, with and without the acceptance rate, is almost the same with negligible difference.

Next, we checked how much time the SA and GA take on average to find the *HPS* in the random proof sequence that matches with the *HPS* in the target sequence (shown in Fig. 3). The runtime difference when applying the SA and GA with various crossover and mutation operators to find the correct *HPS* in a proof sequence is negligible. It is observed that SA was able to quickly find the matched *HPS* as compared to the GA with different crossover and mutation operators. For GA, the time to find the *HPS* increases for each following *HPS*. However, this was not

Table 5 Average total generation count for SA and GA

	Ave. Generation Count	Total Time	Memory
SA	1,327,268	39.27 s	3907 Mb
GA(SPC/SM)	123,513,780	4770.50 s	5545 Mb
GA(MPC/SM)	120,580,649	3914.47 s	5463 Mb
GA(UC/SM)	123,441,934	4471.79 s	5507 Mb
GA(SPC/MPIM)	9,352,574	506.25 s	5419 Mb
GA(MPC/MPIM)	8,851,855	463.34 s	5516 Mb
GA(UC/MPIM)	8,704,233	491.97 s	5289 Mb

Table 6 SD_t, mean and total time for the GA

C & M	Mean	SDt	Time (s)	Memory
SPC/SM	93.12	17.28	4770.50 s	5545 Mb
MPC/SM	93.12	17.21	3914.47 s	5463 Mb
UC/SM	97.93	18.41	4471.79 s	5507 Mb
SPC/MPIM	6.46	1.23	506.25 s	5419 Mb
MPC/MPIM	5.85	1.12	463.34 s	5516 Mb
UC/MPIM	5.83	1.19	491.97 s	5289 Mb

the case for the SA. The time taken by the SA to find each matched *HPS* was uniform.

The longest proof in the *PD* is for the theorem *T*2 (positive value of sine) and it consists of 81 *HPS*. Here we call this theorem *PSF*. The runtime of the SA and GA to find all matched 81 *HPS* in *PSF* with different crossover and mutation operators is shown in Fig. 4. Those generations are shown on the x-axis where both algorithms were able to find the *HPS* in a random proof sequence that matches with the *HPS* in *PSF*. Generations where *HPS* does not match are excluded. We observed that in most of the generations, both algorithms were unable to find the same *HPS* in a random proof sequence and *PSF*. For GA, the time 0 in generations 39-44 indicates that the random proof sequences evolved by the GA have not matched the *HPS* in *PSF*. That is why, it takes 121 generations on average to find the complete proof. On the other hand, the behavior of SA to find the matched *HPS* is uniform and it was able to find all the matched *HPS* in 81 generations.

In each generation, the probability for the SA and GA to find the complete correct proof for *PSF* is listed in Table 8. SA has high probability as compared to the GA with different crossover and mutation operators. The performance of both algorithms is much better than proof searching with a pure random searching approach. For example, the probability (which is very low as compared to SA and GA) for a pure random search to find a valid proof is also listed in Table 8. Even for the theorems with smaller (fitness of 10) proof sequences, the probability is still in the magnitude of 10^{-21}.

We argued earlier that using brute force approach (BFA) for proof searching is infeasible. To support this argument, we also implemented a BFA in Python that can be found at [40]. The BFA takes a lot of time when executed

Table 7 Performance of SA with and without *AR*

Results without AR		Results with AR (*ARC* = 122640)	
Avg. Gen. Count	Time	Avg. Gen. Count	Time
1327268	39.74 s	1427268	42.43 s

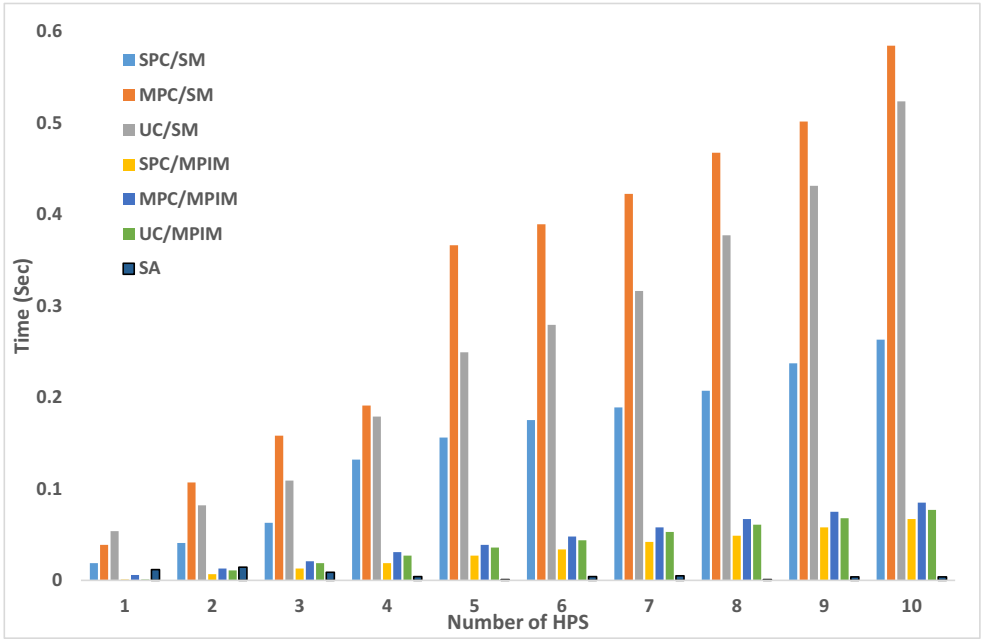


Fig. 3 Time used by the SA and GA to find the first ten matched *HPS*

on proof sequences in the *PD*, even for theorems with smaller fitness values. For example, Table 9 lists the results obtained with the BFA. The attempts column shows how many times (iterations) the approach tried to find the target proof sequence. For a theorem with a fitness value of 4, it took 124 seconds and 13239202 attempts, that is approximately 106767 attempts per second, to find the target proof sequence.

For a theorem with a fitness value of 6, the program kept running for more than 50 hours and it was still unable to find the target proof sequence.

Overall, it was observed through various experiments that the proposed GA and SA are able to quickly optimize and automatically find the correct proofs for theorems/lemmas in different HOL4 theories. SA was observed to be much faster than GA and in turn utilized less memory. Besides

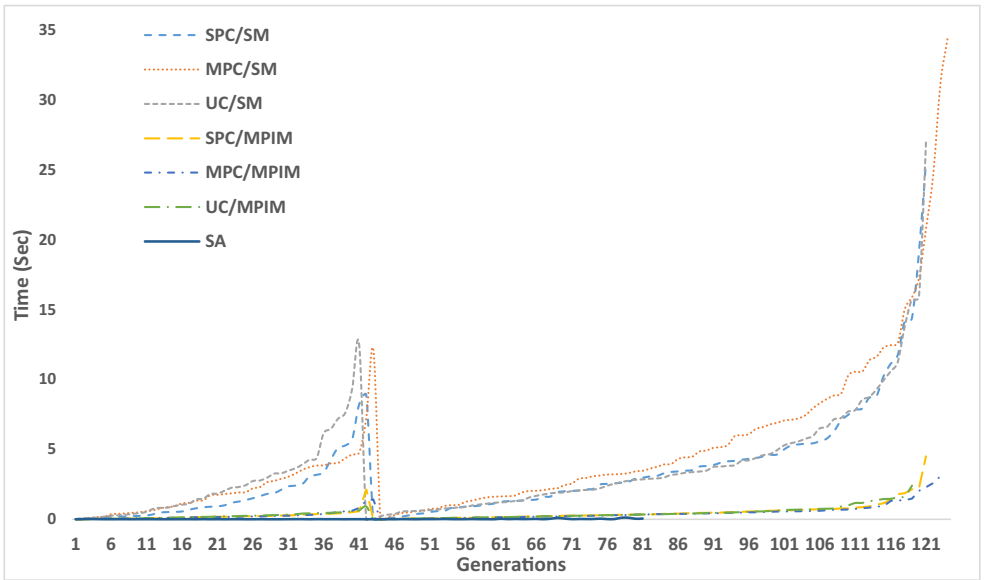


Fig. 4 Total time and generations for the *PSF* theorem with SA and GA

Table 8 Comparison of SA, GA and Pure Random Search (PRS)

Algorithm	Probability
SA	2.12×10^{-3}
GA(CO/SM)	4.84×10^{-6}
GA(CO/MPIM)	5.04×10^{-5}
PRS	1.62×10^{-197}

HOL4, both evolutionary and heuristic based approaches can also be used for proof searching and proof optimization in other proof assistants, such as Coq [4] and PVS [5].

6 Proof learning with LSTM

The proof searching approaches in Section 4 are unable to learn the proof process. They are efficient in evolving random proof sequences to target proof sequences. The main focus now in theorem provers, particularly in ITPs, is to make the proof development process as automatic as possible. This will not only ease the proving process for users but also reduce the time and efforts users spend while interacting with ITPs. In this context, proof learning and prediction is an important task. With the advancement and evolution in computing capabilities and the fast progress in deep learning techniques, we believe that RNNs are suitable to provide an effective proof learning mechanism for formal proofs because of their instrumental success in the AlphaGo [42] and usefulness in the tasks related to logical inference and reasoning [43, 44], automated translation [45], conversation modeling [46] and knowledge base completion [47]. Therefore, we propose to use a variant of RNN known as LSTM for the task of proof learning.

During the proof development process, HOL4 users are required to formalize their inputs with (1) *HPS*, and (2) arguments for those *HPS*. For example, the tactic (*HPS*) *DISCH_TAC* moves the antecedent of an implicative goal into assumptions. Similarly, *GEN_TAC* strips the outermost universal quantifier from the conclusion of a goal. Tactic arguments (called dependencies in [16]) provide more detailed information about *HPS*. For example, the *HPS Induct_on 'n'* applies induction on a variable n. In the proposed proof learning and prediction with RNN, we focus on *HPS* only. Since automatic reasoning in ITPs is a hard problem due to the undecidability of the underlying higher-order logic [48, 49], the aim is not to provide the support for fully automated reasoning. Instead, our aim is to provide an approach that can analyze existing HOL4 proofs to learn the proof process and on the basis of learning, predict proof-steps/tactics (*HPS*).

RNNs have a simple structure of repeating units that allows the flow of information. Standard RNNs are usually built with *tanh* activation function as shown in Fig. 5a. With x_t, hs_t and y_t as the input, hidden state and output respectively, the hs_t and y_t for the unit t can be calculated as:

$$hs_t = tanh(W_{xhs}x_t + W_{hshs}hs_{t-1} + b_{hs}), \quad (1)$$

$$y_t = (W_{yhs}hs_t + b_y) \quad (2)$$

where W_{xhs}, W_{hshs} and W_{yhs} represent the weight variable for input, hidden unit and output, respectively, and b_{hs}, b_y are the biases in the unit.

In principle, RNNs can store and manipulate past information to produce the desired information as output. As we are dealing with long proofs sequences, RNNs are not suitable for learning and prediction due to their short term memory and the vanishing gradient problem [50].

This problems occurs when RNNs are trained with gradient based methods (e.g back-propagation). It describes the situation where RNNs are unable to propagate useful gradient information from the output end back to the layers near the input end. As more layers are added, the gradients approach zero, making the network hard to train.

For example, the derivative of the gradient that passes through the *tanh* activation function is smaller than 1 for all inputs except 0. Then the state of the unit t can be represented as:

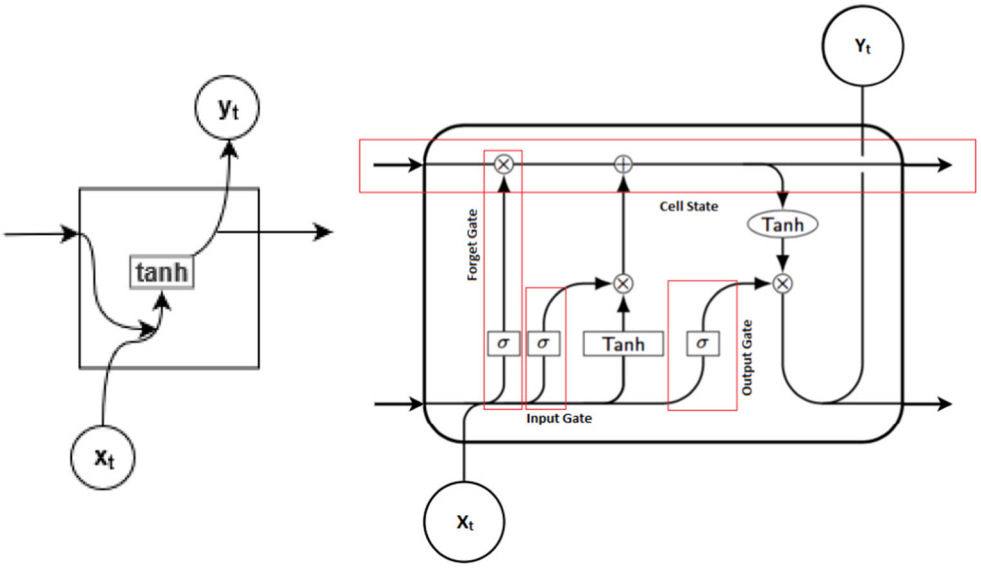
$$\begin{aligned} hs_t &= tanh(W_{xhs}x_t + W_{hshs}hs_{t_1} + b_{hs}) \\ &= tanh(W_{xhs}x_t + W_{hshs}tanh(W_{xhs}x_{t_1} \\ &\qquad + W_{hshs}hs_{t_2} + b_{hs})) \\ &= \ldots\ldots\ldots\ldots \end{aligned}$$

with the increase in t, the effect of x_1 keeps decreasing, which apparently contributes to the vanishing gradient problem.

One way to avoid the problem is to use *ReLU* in place of the *tanh* or *sigmoid* (σ) function. However, *ReLU* helps

Table 9 Results for BFA

Proof Sequence	Fit	Time	Attempts
GEN_TAC REWRITE_TAC	2	0.001 s	67
STRIP_TAC SRW_TAC METIS_TAC	3	0.078 s	4292
GEN_TAC BETA_TAC Q.SPEC_TAC ASM_REWRITE_TAC	4	124 s	13239202
RW_TAC CASES_ON CASES_ON FULL_SIMP_TAC PROVE_TAC	5	4412s	922370015



(a) RNN cell (b) LSTM cell

Fig. 5 Difference between RNN and LSTM (Figures courtesy of [51])

in avoiding the problem but does not rectify the problem completely.

LSTMs [38] are RNNs that are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior. LSTMs have a chain like structure (similar to RNNs), but with different unit: four neural network layers instead of a single one as shown in Fig. 5b. A LSTM cell has 3 gates: (1) Input gate, (2) Forget gate, and (3) Output gate. Each gate is a layer with an associated weight and bias (for example, W_f and b_f for the forget gate, W_i and b_i for the input gate and W_o and b_o for the output gate).

The forget gate decides whether to keep the information from previous hidden state or to delete it. This decision is made by a sigmoid layer in the forget gate that looks at the previous hidden state (hs_{t-1}) and current input (x_t), and outputs a number between 0 (means to keep information) and 1 (means to delete information). The input gate decides whether a given information is worth remembering and the σ in this layer decides which value to update. The output gate decides whether the information at a given step is important and should be used or not. Let f_t, i_t and o_t represents the output of forget gate, input gate and output gate layers respectively, then:

$$f_t = \sigma(W_f.[hs_{t-1}, x_t] + b_f) \tag{3}$$
$$i_t = \sigma(W_i.[hs_{t-1}, x_t] + b_i) \tag{4}$$
$$o_t = \sigma(W_o.[hs_{t-1}, x_t] + b_o) \tag{5}$$

Besides three gates, there is a $tanh$ layer between the input and output gates that creates a vector of new candidate (temporal) values (denoted as $\bar{C}_t$) for the current time step, which can be added to the state. The new cell state (C_t) is calculated with $\bar{C}_t$, old cell state (C_{t-1}), f_t and i_t. Finally, the output of the LSTM cell is based on the $tanh$ layer (between output gate and cell state) and o_t. It is important to point out that the σ in the output gate layer decides which information from the cell state will go to the output. The new cell state finally passes through $tanh$ to push its values in between -1 and 1.

$$\bar{C}_t = tanh(W_C.[hs_{t-1}, x_t] + b_C) \tag{6}$$
$$C_t = f_t \times C_{t-1} + i_t + C_t \tag{7}$$
$$Y_t = o_t \times tanh(C_t) \tag{8}$$

where . is the dot product operator (between vectors), $\times$ is the pointwise multiplication operator (between a real number and a vector) and $[y, z]$ is the concatenation operator (between vectors).

The dataset *PD* contains the proof sequences, where each line represents the proof sequence for one theorem/lemma. The dataset is pre-processed further, where each line that contains *HPS* is split into lists of characters that are transformed into vectors (tensors) so that LSTM can process them one by one. Each character in *HPS* is mapped to a distinct number $\{char \rightarrow number\}$. The character with a low number indicates that the respective character is more popular (occurs more) among others in the *PD*. In the tensor, the characters in a *HPS* are replaced with their respective number. For example, the tactic GEN_TAC is transformed into the following tensor:

$$GEN\_TAC \rightarrow [28 \quad 46 \quad 4 \quad 5 \quad 15 \quad 11 \quad 19]$$

The main reason why the arguments for *HPS* is not selected is that these parameters depends on the specification (particularity on variables and functions declarations)

inside the theory and on the proof goal. This means that arguments for a particular *HPS* can be different for different theories and different proof goals. During the reasoning process in HOL4, a proof goal can be considered as a context-*HPS* pair, where context contains the information about the current hypotheses, variables and the goal that needs proving. The goal may contains a set of subgoals. The user is required to guide the proof process towards completion by suggesting which *HPS* (and arguments) to use. We believe that adding arguments information would restrict the learning model to work well for only one (or some related) theory. Moreover, it will also add more complexity and will increase the computation time for the model.

Our aim is to train the LSTM model in such a way that for a given input, the model generates the desired output. For that, the created tensor is divided into input and output (target) batches. Let us assume (for simplicity) that the total length of the proof sequence is 8. For this sequence, the first input batch can be set to 5 initial tensor values. Our requirement from the model is to predict the next character for a list of previous characters. This requirement can be satisfied by creating a target batch that subscripts the tensor with a $n + 1$ shift compared to the input batch, where n represents the length of input batch. Table 10 shows how the model predicts the next item with the above mentioned settings for input and output batches.

The model generates input/target pairs according to the hyperparameters values that are listed in Table 11. All these pairs create a one training epoch. Within every epoch, the model iterates through every batch where it is provided with input/target pairs.

The LSTM model for proof learning and prediction is heavily influenced by the model available at github.com/gsurma/text_predictor and is implemented in Python using the Tensorflow library [52].

The model completes 2 training epochs approximately in each 100 iterations. On average, the model completes 100 iterations in approximately 550 seconds on a fifth generation Core *i*5 processor with 8 GB of RAM. Compared to proof searching approaches, the learning model is computationally slow. We evaluated the effectiveness of LSTM in modeling the given data (proof sequences) with the loss function (also known as cost function). The learning curve is shown in Fig. 6. The loss functions decreases with increase in the iterations and epochs. This means that the model learning rate was very high at the start and after 105 epochs, it stopped learning. After that, the learning behavior for the model is uniform for the next epochs and iterations meaning that the model is not learning anything new.

Table 10 A sample for input/output batches for LSTM

HPS	S	I	M	P	_	T	A	C
tensor	85	32	56	31	5	15	11	19
input 1	85	32	56	31	5			
output 1		32	56	31	5	15		
input 2		32	56	31	5	15		
output 2			56	31	5	15	11	
input 3			56	31	5	15	11	
output 3				31	5	15	11	19

Table 11 Hyperparameters and their values for LSTM network

Batch Size	32
Sequence Length	25
Learning Rate	0.01
Decay Rate	0.97
Hidden Layer Size	256
Cells Size	2

One justification for this behavior is that the dataset contains 89 *HPS*, where each *HPS* is composed from specific English characters. So the dataset is restricted in nature such that it contains limited vocabulary.

The similarity curve is shown in Fig. 7, where the proof sequences predicted by the model are compared with the proofs in the *PD*. The highest similarity (approximately 18%) was achieved in 95-105 epochs. A sample of predicted proof sequences for some epochs and iterations is listed in Table 12. The main limitations with the approach is the computational time: the model is too slow and takes alot of time in the learning process. From results, we can say that out of 300 proof sequences in the *PD*, the model was able to correctly predict 54 proofs sequences.

The LSTM model learned the proof sequences in HOL4 theories from scratch and initially it had no knowledge or understanding of *HPS*. Furthermore, it learned only from a relatively small dataset and we believe that results would probably be even better with a larger dataset. RNN and LSTM were also used in [29] to predict the correct tactics in one Coq theory. Both models (implemented with Keras library in Python) produced output in the form of a probability distribution. The *n*-correctness rate of tactics was used to check whether a tactic can be used in the new proofs of Coq goals. Moreover, cost entropy (that measures the difference between two probability distributions) was used as loss function in both models. These preliminary results indicate that the research direction of linking and integrating evolutionary/heuristic and neural networks techniques with HOL4 is worth pursuing. These approaches may have a considerable impact to advance and accumulate human knowledge, especially in the fields of formal logic, deep learning and computation.

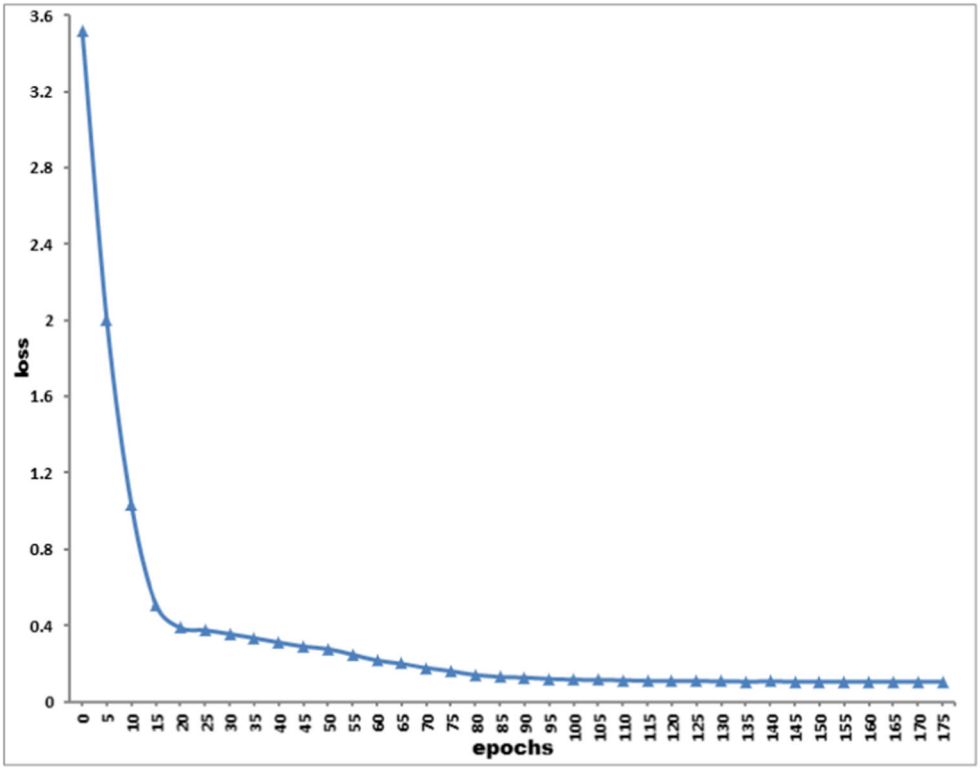


Fig. 6 Learning curve

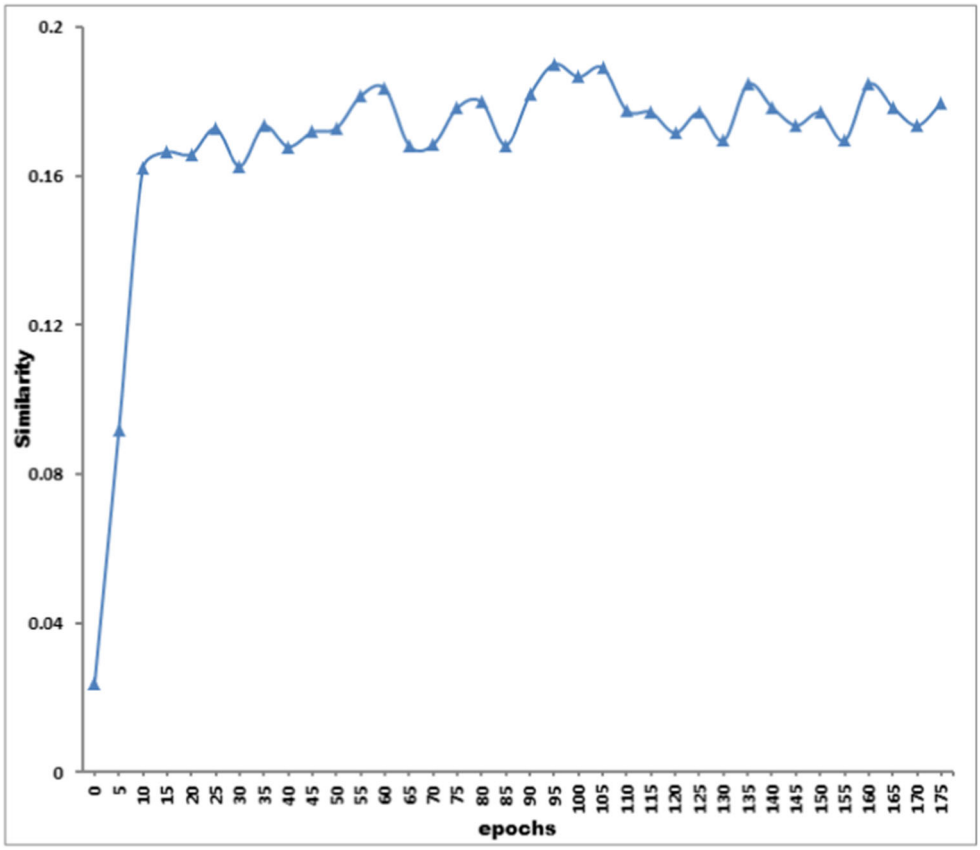


Fig. 7 Similarity curve

Table 12 A sample of predicted proofs sequences

Ite.	Epochs	Predictions	Time
0	0	Q.P.LTWKIFAT RJ TMFME_TSKA.UOKI2KHP JDTA_H J2R R.FPXPBIM PG_LFK_XUHPETLX ESILX LIQANOWUE_FECT1F AN NJJSFBCHBS HABKOOM. DHG_UMKO._D CRRC2STT SLAFPBE__XMR.U.1WD_ERHATANKS_BFEOMRTT UCXLXQ.FRAESD SAPI2SDTI.WIIAVDNL.RHXSIPR STS2O AVA2T MVERNR RDADSNTERFAA_L _A.EED.SXLB.SJ.V L1_1XMF2 KF22HQM.Q1BAHNWLSGATCEYY _MSHIRD OMVHJWU G G2TRU_SLQEU_W222.EJH_YLNAL	0 s
2500	44	AC SIMP_TAC STRIP_TAC PROVE_TAC STRIP_TAC Q.EXISTS_TAC Q.EXISTS_TAC SRW_TAC SRW_TAC METIS_TAC GEN_TAC CASE_TAC ASM_SIMP_TAC POP_ASSUM STRIP_ASSUME_TAC ASM_SIMP_TAC SUBST1_TAC POP_ASSUM Q.EXISTS_TAC STRIP_TAC EQ_TAC SIMP_TAC RW_TAC POP_ASSUM SIMP_TAC PROVE_TAC PROVE_TAC REWRITE_TAC STRIP_TAC MP_TAC MATCH_MP_TAC REWRITE_TAC ONCE_REWRITE_TAC RW_TAC FULL_SIMP_TAC METIS_TAC METIS_TAC	11059.6 s
5900	100	Q.EXISTS_TAC SRW_TAC SRW_TAC STRIP_TAC MATCH_MP_TAC Q.EXISTS_TAC SRW_TAC SRW_TAC PROVE_TAC STRIP_TAC DISCH_THEN MP_TAC MATCH_MP_TAC ASSUME_TAC CONJ_TAC MP_TAC DISCH_THEN REWRITE_TAC STRIP_TAC RW_TAC RW_TAC PROVE_TAC PROVE_TAC STRIP_TAC STRIP_TAC STRIP_TAC PROVE_TAC STRIP_TAC ASM_SIMP_TAC MATCH_MP_TAC CONJ_TAC RW_TAC PROVE_TAC RW_TAC MP_TAC ASM_SIMP_TAC DISCH_THEN RW_TAC RES_TAC RW_TAC PROVE_TAC'	25873.2 s
11200	200	STRIP_TAC MP_TAC INDUCT_TAC FULL_SIMP_TAC ASM_SIMP_TAC PROVE_TAC STRIP_TAC MATCH_MP_TAC PROVE_TAC POP_ASSUM SUBST1_TAC POP_ASSUM ALL_TAC Q.EXISTS_TAC SRW_TAC METIS_TAC Q.EXISTS_TAC SRW_TAC ALL_TAC SRW_TAC EQ_TAC SRW_TAC Q.EXISTS_TAC SRW_TAC METIS_TAC SIMP_TAC GEN_TAC STRIP_TAC METIS_TAC HO_MATCH_TAC SRW_TAC ASM_CASES_TAC Q.EXISTS_TAC SRW_TAC HO_MATCH_TAC STRIP_TAC SUBST1_TAC FULL_SIMP_TAC	77110.7 s

7 Conclusion

ITPs require user interaction with the proof assistants to guide and find the proof for a particular goal, which can make the proof development process cumbersome and time consuming, in particular for long and complex proofs. We introduced two proof searching approaches in this paper for the possible linkage between evolutionary and heuristic algorithms, such as GA and SA, with theorem provers, such as HOL4, to make the proof finding and development process easier. Both GA and SA were used to optimize and find the correct proofs in different HOL4 theories. Moreover, the performance of SA is compared with GA and it was found that SA performed better than GA. However, the proof searching approaches are unable to learn the proof process. For the tasks of proof guidance and automation, a deep neural network (LSTM) was used that is trained on HOL4 theories for the learning purposes. After training, the model is able to correctly predict the proofs sequences for HOL4 proofs.

The proposed work leads to several directions for future work. First, we would like to make the proof searching process more general in nature to evolve frequent proof steps to compound proof strategies for guiding the proofs of new conjectures. We also intend to perform more experiments with headless chicken macromutation [53] to investigate the usefulness of crossover operators in GA for proof searching and optimization in HOL4. Moreover, stochastic optimization techniques, such as particle swarm optimization [54], and heuristic search algorithms, such as monte carlo tree search [55], or the hybrid approaches such as PS-ACO algorithm [56] could be considered for proof searching. Another direction is to take advantage of the Curry-Howard isomorphism for sequent calculus [57] that provides a direct relation between programming and proofs, where finding proofs can be viewed as writing programs. With such correspondence, a SA or GA can be used to write programs (proofs) and HOL4 proof assistant for simplification and verification by computationally evaluating the programs. For the proof learning approach, it would be interesting to optimize the structure of LSTM network and use other deep learning techniques such as Gated recurrent unit (GRU) [58] for better results. Moreover, predicting the arguments for *HPS* is another interesting area. This will enable us to fully automate the proof development process for new goals.

Funding The work was partially supported by the Guangdong Science and Technology Department (under grant no. 2018B010107004) and the National Natural Science Foundation of China under grant no. 61772038 and 61532019.

Compliance with Ethical Standards

Conflict of interests The authors declare that they have no conflict of interest.

Ethical Approval This paper does not contain any studies with human participants or animals performed by any of the authors.

References

1. Hasan O, Tahar S (2015) Formal verification methods. In: Encyclopedia of Information Science & Technology, 3rd edn. IGI Global, pp 7162–7170
2. Kaliszyk C, Chollet F, Szegedy C (2017) Holstep: A machine learning dataset for higher-order logic theorem proving. CoRR arXiv:1703.00426
3. Slind K, Norrish M (2008) A brief overview of HOL4. In: Proceedings of International Conference on Theorem Proving in Higher Order Logics (TPHOLs), pp 28–32
4. Bertot Y, Casteran P (2004) Interactive theorem proving and program development: Coq'Art: The calculus of inductive construction. Springer Publisher
5. Owre S, Shankar N, Rushby JM, Stringer-Calvert DWJ (2001) PVS System guide, PVS prover guide PVS language reference. Technical report, SRI International
6. Wiedijk F (Accessed on 3 January 2020) Formalizing 100 theorems, available at: http://www.cs.ru.nl/~freek/100
7. Hales TC, Adams M, Bauer G, Dang DT, Harrison J, Hoang TL, Kaliszyk C, Magron V, McLaughlin S, Nguyen TT, Nguyen TQ, Nipkow T, Obua S, Pleso J, Rute JM, Solovyev A, Ta AHT, Tran TN, Trieu DT, Urban J, Vu KK, Zumkeller R (2017) A formal proof of the Kepler conjecture. Forum Math Pi 5 e2:1–29
8. Gonthier G, Asperti A, Avigad J, Bertot Y, Cohen C, Garillot F, Roux SL, Mahboubi A, O'Connor R, Biha SO, Pasca I, Rideau L, Solovyev A, Tassi E, Thery L (2013) A machine-checked proof of the Odd Order theorem. In: Proceedings of International Conference on Interactive Theorem Proving (ITP), pp 163–179
9. Leroy X (2009) Formal verification of a realistic compiler. Commun ACM 52(7):107–115
10. Blanchette JC, Haslbeck MPL, Matichuk D, Nipkow T (2015) Mining the archive of formal proofs. In: Proceedings of International Conference on Intelligent Computer Mathematics (CICM), pp 3–17
11. Harrison J, Urban J, Wiedijk F (2014) History of interactive theorem proving. In: Computational Logic, volume 9 of Handbook of the History of Logic, pp 135–214
12. Kaliszyk C, Urban J (2015) Learning-assisted theorem proving with millions of lemmas. J Symb Comput 69:109–128
13. Färber M, Brown CE (2016) Internal guidance for Satallax. In: Proceedings of International Joint Conference on Automated Reasoning (IJCAR), pp 349–361
14. Gauthier T, Kaliszyk C, Urban J (2017) TacticToe: Learning to reason with HOL4 tactics. In: Proceedings of International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), pp 125–143
15. Irving G, Szegedy C, Alemi AA, Eén N, Chollet F, Urban J (2016) Deepmath - Deep sequence models for premise selection. In: Proceedings of Annual Conference on Neural Information Processing Systems (NIPS), pp 2235–2243
16. Kaliszyk C, Mamane L, Urban J (2014) Machine learning of Coq proof guidance: First experiments. In: Proceedings of International Symposium on Symbolic Computation in Software Science (SCSS), pp 27–34
17. Nawaz MS, Sun M, Fournier-Viger P (2019) Proof guidance in PVS with sequential pattern mining. In: Proceedings of International Conference on Fundamentals of Software Engineering (FSEN), pp 45–60
18. Nawaz MZ, Hasan O, Nawaz MS, Fournier-Viger P, Sun M (2020) Proof searching in HOL4 with genetic algorithm. In: Proceedings of Annual ACM Symposium on Applied Computing (SAC), pp 513–520
19. Huang SY, Chen YP (2017) Proving theorems by using evolutionary search with human involvement. In: Proceedings of Congress on Evolutionary Computation (CEC), pp 1495–1502
20. Yang LA, Liu JP, Chen CH, Chen YP (2016) Automatically proving mathematical theorems with evolutionary algorithms and proof assistants. In: Proceedings of Congress on Evolutionary Computation (CEC), pp 4421–4428
21. Koza JR (1993) Genetic programming - On the programming of computers by means of natural selection. MIT Press Cambridge, Massachusetts
22. Duncan H (2007) The use of data-mining for the automatic formation of tactics. PhD Thesis, University of Edinburgh, UK
23. Alama J, Heskes T, Kühlwein D, Tsivtsivadze E, Urban J (2014) Premise selection for mathematics by corpus analysis and kernel methods. J Auto Reaso 52(2):191–213
24. Loos SM, Irving G, Szegedy C, Kaliszyk C (2017) Deep network guided proof search. In: Proceedings of International Conference on Logic for Programming, Artificial Intelligence and Reasoning (LPAR), pp 85–105
25. Wang M, Tang Y, Wang J, Deng J (2017) Premise selection for theorem proving by deep graph embedding. In: Proceedings of Annual Conference on Neural Information Processing Systems (NIPS), pp 2786–2796
26. Whalen D (2016) Holophrasm: a neural automated theorem prover for higher-order logic. CoRR arXiv:1608.02644
27. Gauthier T, Kaliszyk C (2015) Premise selection and external provers for HOL4. In: Proceedings of International Conference on Certified Programs and Proofs (CPP), pp 49–57
28. Kaliszyk C, Urban J (2015) Hol(y)hammer: Online ATP service for HOL light. Mathe Comp Sci 9(1):5–22
29. Zhang X, Li Y, Hong W, Sun M (2019) Using recurrent neural network to predict tactics for proving component connector properties in Coq. In: Proceedings of International Symposium on Theoretical Aspects of Software Engineering (TASE), pp 107–112
30. Holland JH (1975) Adaptation in natural and artificial systems. University of Michigan Press, Ann Arbor
31. Mitchell M (1996) An introduction to genetic algorithms. MIT Press Cambridge, Massachusetts
32. Hong T, Wang H, Lin W, Lee WY (2002) Evolution of appropriate crossover and mutation operators in a genetic process. Appl Intell 16(1):7–17
33. Bertsimas D, Tsitsiklis J (2013) Simulated annealing. Stati Sci 8(1):10–15
34. Delahaye D, Chaimatanan S, Mongeau M (2019) Simulated annealing: From basics to applications. In: Handbook of Metaheuristics, pp 1–35
35. Yao X, McKay RI (2001) Simulated evolution and learning: an introduction. Appl Intell 15(3):151–152
36. Graves A (2012) Supervised sequence labelling with recurrent neural networks. Springer Publisher
37. Medsker L, Jain LC (1999) Recurrent neural networks: Design and applications. CRC Press
38. Hochreiter S, Schmidhuber J (1997) Long short-term memory. Neur Comp 9(8):1735–1780
39. Fournier-Viger P, Lin JCW, Kiran RU, Koh YS, Thomas R (2017) A survey of sequential pattern mining. Data Sci Patt Recogn 1(1):54–77
40. Python codes and HOL4 data, available at: github.com/MuhammadzohaibNawaz/PRS-GA-and-SA-in-Python
41. Nsakanda AL, Price WL, Diaby M, Gravel M (2007) Ensuring population diversity in genetic algorithms: a technical note with application to the cell formation problem. Euro J Operat Rese 178(2):634–638

42. Silver D, Huang A, Maddison CJ, Guez A, Sifre L, van den Driessche G, Schrittwieser J, Antonoglou I, Panneershelvam V, Lanctot M, Dieleman S, Grewe D, Nham J, Kalchbrenner N, Sutskever I, Lillicrap TP, Leach M, Kavukcuoglu K, Graepel T, Hassabis D (2016) Mastering the game of Go with deep neural networks and tree search. Nature 529(7587):484–489
43. Kaiser L, Sutskever I (2016) Neural GPUs learn algorithms In: International Conference on Learning Representations, (ICLR). Poster
44. Sukhbaatar S, Szlam A, Weston J, Fergus R (2015) End-to-end memory networks. In: Proceedings of Annual Conference on Neural Information Processing Systems (NIPS). pp 2440–2448
45. Wu Y, Schuster M, Chen Z, Le QV, Norouzi M, Macherey M, Krikun M, Cao Y, Gao Q, Macherey Q, Klingner J, Shah A, Johnson M, Liu X, Kaiser L, Gouws S, Kato Y, Kudo T, Kazawa H, Stevens K, Kurian G, Patil N, Wang W, Young C, Smith J, Riesa J, Rudnick A, Vinyals O, Corrado G, Hughes M, Dean J (2016) Google's neural machine translation system: Bridging the gap between human and machine translation. CoRR arXiv:1609.08144
46. Vinyals Q, Le QV (2015) A neural conversational model. CoRR arXiv:1506.05869
47. Socher R, Chen D, Manning CD, Ng AY (2013) Reasoning with neural tensor networks for knowledge base completion. In: Proceedings of Annual Conference on Neural Information Processing Systems (NIPS), pp 926–934
48. Nawaz MS, Malik M, Li Y, Sun M, Lali MI (2019) A survey on theorem provers in formal methods. CoRR arXiv:1912.03028
49. Huet GP (1973) The undecidability of unification in third order logic. Informa Cont 22(3):257–267
50. Hochreiter S (1998) The vanishing gradient problem during learning recurrent neural nets and problem solutions. Int J Uncertain Fuzz 6(2):107–116
51. Olah C (2015) Understanding LSTM networks, available at: https://colah.github.io/posts/2015-08-Understanding-LSTMs
52. Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M, Kudlur M, Levenberg J, Monga R, Moore S, Murray DG, Steiner B, Tucker PA, Vasudevan V, Warden P, Wicke M, Yu Y, Zheng X (2016) Tensorflow: A system for large-scale machine learning. In: Proceedings of USENIX Symposium on Operating Systems Design and Implementation (OSDI), pp 265–283
53. Jones T (1995) Crossover, macromutationand, and population-based search. In: Proceedings of the 6th International Conference on Genetic Algorithms (ICGA), pp 73–80
54. Kennedy J, Eberhart R (1995) Particle swarm optimization. In: Proceedings of International Conference on Neural Networks (ICNN), pp 1942–1948
55. Browne CB, Powley E, Whitehouse D, Lucas SM, Cowling I, Rohlfshagen P, Tavener S, Perez D, Samothrakis S, Colton S (2012) A survey of monte carlo tree search methods. IEEE Trans Comp Intell AI 4(1):1–43
56. Shuan B, Chen J, Li Z (2011) Study on hybrid PS-ACO algorithm. Appl Intell 34(1):64–73
57. Santo JE (2015) Curry-howard for sequent calculus at last! In: Proceedings of International Conference on Typed Lambda Calculi and Applications (TLCA), pp 165–179
58. Cho K, van Merrienboer B, Bahdanau D, Bengio Y (2014) On the properties of neural machine translation: Encoder-decoder approaches. In: Proceedings of Workshop on Syntax, Semantics and Structure in Statistical Translation (SSST@EMNLP), pp 103–111

M. Saqib Nawaz received the B.S. degree in computer systems engineering from the University of Engineering and Technology, Peshawar, Pakistan in 2011, the M.S. degree in computer science from the University of Sargodha, Pakistan in 2014, and the Ph.D. degree from Peking University, Beijing, China, in 2019. He is currently a Postdoctoral Fellow with the Harbin Institute of Technology, Shenzhen, China. His research interests include formal methods (theorem provers and model checkers), evolutionary computation, and the use of machine learning and data mining in Software Engineering.

M. Zohaib Nawaz received his Bachelors degree from University of Sargodha, Pakistan in 2016 and Masters degree from National University of Sciences and Technology (NUST), Pakistan in 2020. Currently, he is Lecturer in department of Computer Science & IT, University of Sargodha, Pakistan since 2018. His research interest includes Artificial Intelligence and Formal Methods. He is a member of ACM.

Osman Hasan received his BEng (Hons) degree from the University of Engineering and Technology, Peshawar Pakistan in 1997, and the MEng and PhD degrees from Concordia University, Montreal, Quebec, Canada in 2001 and 2008, respectively. Before his PhD, he worked as an ASIC Design Engineer from 2001 to 2004 at LSI Logic. He worked as a postdoctoral fellow at the Hardware Verification Group (HVG) of Concordia University for one year until August 2009. Currently, he is an Associate Professor and the Dean of the School of Electrical Engineering and Computer Science of National University of Science and Technology (NUST), Islamabad, Pakistan. He is the founder and director of System Analysis and Verification (SAVe) Lab at NUST, which mainly focuses on the design and formal verification of energy, embedded and e-health related systems. He has received several awards and distinctions, including the Pakistans Higher Education Commissions Best University Teacher (2010) and Best Young Researcher Award (2011) and the Presidents gold medal for the best teacher of the University from NUST in 2015. Dr. Hasan is a senior member of IEEE, member of the ACM, Association for Automated Reasoning (AAR) and the Pakistan Engineering Council.

Philippe Fournier-Viger (Ph.D) is full professor at the Harbin Institute of Technology (Shenzhen), China. Five years after completing his Ph.D., he came to China and became full professor at the Harbin Institute of Technology (Shenzhen), after obtaining a title of national talent from the National Science Foundation of China. He is associate editor-in-chief of the Applied Intelligence journal. He has published more than 280 research papers in refereed international conferences and journals, which have received more than 6000 citations. He is the founder of the popular SPMF open-source data mining library, which provides more than 170 algorithms for identifying various types of patterns in data. The SPMF software has been used in more than 800 papers since 2010 for many applications from chemistry, smartphone usage analysis restaurant recommendation to malware detection. He is editor of the book High Utility Pattern Mining: Theory, Algorithms and Applications published by Springer in 2019, and co-organizer of the Utility Driven Mining and Learning workshop at KDD 2018 and ICDM 2019, and 2020. His research interests include data mining, frequent pattern mining, sequence analysis and prediction, big data, and applications.

Meng Sun is currently a Professor at School of Mathematical Sciences, Peking University, China. He received Bachelor and PhD degrees in Applied Mathematics from Peking University in 1999 and 2005. He then spent one year as a postdoctoral researcher at National University of Singapore. From 2006 to 2010, he worked as a scientific staff member at CWI, Amsterdam, Netherlands. He has been a faculty member of Peking University since 2010. His research interests include Software Engineering, Formal Methods, Software Verification and Testing, Coalgebra Theory, Cyber Physical Systems and Big Data Analysis.

Affiliations

M. Saqib Nawaz[1] · M. Zohaib Nawaz[2,3] · Osman Hasan[2] · Philippe Fournier-Viger[1] · Meng Sun[4]

M. Saqib Nawaz
msaqibnawaz@hit.edu.cn

M. Zohaib Nawaz
mnawaz.mscs16seecs@seecs.edu.pk

Osman Hasan
osman.hasan@seecs.edu.pk

Meng Sun
sunmeng@math.pku.edu.cn

1 School of Humanities and Social Sciences, Harbin Institute of Technology (Shenzhen), Shenzhen, China

2 School of Electrical Engineering and Computer Science, National University of Sciences and Technology, Islamabad, Pakistan

3 Department of Computer Science and IT, University of Sargodha, Sargodha, Pakistan

4 School of Mathematical Sciences, Peking University, Beijing, China