Proof Searching in HOL4 with Genetic Algorithm

M. Zohaib Nawaz
National University of Sciences and Technology, Islamabad, Pakistan
mnawaz.mscs16seecs@seecs.edu.pk

Osman Hasan
National University of Sciences and Technology, Islamabad, Pakistan
osman.hasan@seecs.edu.pk

M. Saqib Nawaz
Harbin Institute of Technology, Shenzhen, China
msaqibnawaz@hit.edu.cn

Philippe Fournier-Viger
Harbin Institute of Technology, Shenzhen, China
philfv8@yahoo.com

Meng Sun
Peking University, Beijing, China
sunmeng@math.pku.edu.cn

ABSTRACT

Proof searching and proof automation are the two most desired properties in interactive theorem provers (ITPs) as they generally require manual user guidance, which can be quite cumbersome. In this paper, we provide an evolutionary proof searching approach for the HOL4 proof assistant, where a genetic algorithm (GA) with different crossover and mutation operators is used to search and optimize the proofs in different HOL theories. 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 with GA operators (three crossover and two mutation) till their fitness match the fitness of the target proof sequences. Various crossover and mutation operators are used to compare their effect on the performance of GAs in proof searching. Obtained results suggest that integrating GAs with HOL4 allows us to efficiently support proof finding and optimization.

CCS CONCEPTS

• **Theory of computation → Evolutionary algorithms**; • **Computing methodologies → Genetic algorithms**; • **Software and its engineering →** *Formal methods.*

KEYWORDS

HOL4, Genetic algorithm, Crossover, Mutation, Proof sequences, Fitness

SAC '20, March 30-April 3, 2020, Brno, Czech Republic

ACM ISBN 978-1-4503-6866-7/20/03…$15.00
https://doi.org/10.1145/3341105.3373917

ACM Reference Format:
M. Zohaib Nawaz, Osman Hasan, M. Saqib Nawaz, Philippe Fournier-Viger, and Meng Sun. 2020. Proof Searching in HOL4 with Genetic Algorithm. In *The 35th ACM/SIGAPP Symposium on Applied Computing (SAC '20), March 30-April 3, 2020, Brno, Czech Republic.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3341105.3373917

1 INTRODUCTION

Theorem proving is a key formal verification method and is widely 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 for the system are then verified using *theorem provers* [7] based on deductive reasoning. The initial objective to develop theorem provers was to enable mathematicians to prove theorems using computer programs within a sound environment. However, these mechanical tools have evolved with time and now play a vital role in the modeling and reasoning about complex and large-scale systems, especially safety-critical systems.

Theorem provers can be categorized into two main types: Automated theorem provers (ATPs) and interactive theorem provers (ITPs) [15]. ATPs are generally based on propositional and first-order logic (FOL) an involve development of computer programs that can automatically perform logical reasoning. However, FOL is less expressive in nature and cannot be used to define complex problems. On the other hand, ITPs are based on higher-order logic (HOL), which allows quantification over predicates and functions and thus offers support for rich logical formalisms such as dependent and (co)inductive types as well as recursive functions. This expressive power leads to the undecidability problem, i.e., the reasoning process cannot be 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 [21], Coq [1] and PVS [19].

Most studies on designing proof assistants aim at facilitating the user in the proof checking process while ensuring

proof correctness. Nevertheless, users often have to do a lot of repetitive work in ITPs to prove non-trivial tasks that is not only laborious but consumes a large amount of time. Proof guidance and proof automation along with proof searching are extremely desirable features for ITPs. To support these features, machine learning and data mining techniques have been used [3, 6, 10–12, 17] for proof searching, proof automation and for the development of proof tactics/strategies. In this paper, we propose to use evolutionary algorithms to find and evolve proofs in ITPs due to their suitability to solve search and optimization problems.

The focus of this paper is on proof searching and optimization in HOL4 using evolutionary algorithms. In HOL4, proof scripts for a particular theory are stored in a separate file. A particular proof goal in HOL4 depends on the specifications inside the theory and it can be completed with different combinations of proof steps and tactics. Because a HOL4 theory often contains too much information, it is inefficient to apply a brute force or pure random search approach. The proposed idea is to use a Genetic Algorithm (GA) for proof searching where an initial population (a set of potential solution) is first created from frequent HOL proof steps. 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 theorems/lemmas. Three crossover and two mutation operators are compared to access their effect on the overall performance of GAs for proof searching and optimization.

A few studies have considered integrating GAs in ITPs. For example, a GA was used with the Coq proof assistant [9, 22] to automatically find formal proofs of theorems. However, the approach can only be used to successfully find the proofs of easy 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 [13] and a pairwise combination (that focuses only on crossover based approach) were used in [2] on patterns (simple tactics) discovered in Isabelle proofs to evolve them into compound tactics. However, in their approach, Isabelle's proofs were represented using a tree structure, which are linearized, such that the proofs are split into separate sequences, and weights are assigned to these sequences. However, linearization leads to the loss of important connections (information) between different branches of the proofs due to which interesting patterns and tactics may be lost in the evolution process. In this work, the dataset for the proof sequences contains all the necessary important information that is required for the discovery of frequent proof steps, through which initial population for the GA is generated.

The rest of this paper is organized as follows: Section 2 briefly discusses the HOL4 theorem prover and GAs. Section 3 presents the proposed evolutionary approach where a GA with different crossover and mutation operators is used to find and optimize random HOL4 proofs. Evaluation of the proposed approach on different theories available in the HOL4 library is presented in Section 4. Finally, Section 5 concludes the paper with some open research directions.

2 PRELIMINARIES

A brief introduction to the HOL4 proof assistant and GA is provided in this section.

HOL4: HOL4 is an ITP that utilizes the simple type theory along with Hindley-Milner polymorphism for the implementation of higher-order logic. The logic in the HOL4 system is represented in the strongly-typed functional programming language meta language (ML). An ML abstract data type is used to represent higher-order logic theorems and the only way to interact with the theorem prover is by executing ML procedures that operate on values of these data types. Its theories are a collection of valid HOL types, constants, definitions, axioms, and theorems which are generally stored as an ML file. Users can reload a HOL theory into the system and can utilize the corresponding definitions and theorems right away. All proofs in HOL4 are ultimately performed by the computer according to a small set of primitive inference rules. For example, the tactic *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 and *CONJ_TAC* reduces a conjunctive goal into two separate sub-goals.

HOL supports two types of interactive proof methods: forward and backward. In forward proof, the user starts with previously proved theorems and applies inference rules to reach the desired 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 goals into simple sub-goals. In the backward proof method, the user starts with the desired theorem or the main goal and specifies tactics to reduce it to simpler intermediate sub-goals. The above steps are repeated for the remaining intermediate goals until we are left with no further sub-goals and this concludes the proof for the desired theorem. More details on HOL4 can be found in [21].

Genetic Algorithms: GAs [8] 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 ($x = x_1, x_2 ..., x_n$). Each x_i (genes) refers to a particular characteristics of the chromosome [16]. 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.

Crossover operator of GAs 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 as 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. More details on GAs can be found in [8, 14].

3 PROOF SEARCHING WITH A GA

The proposed structure (flowchart) of the GA that is used in this paper to find and optimize the proofs of theorems/lemmas in HOL4 theories is shown in Figure 1.

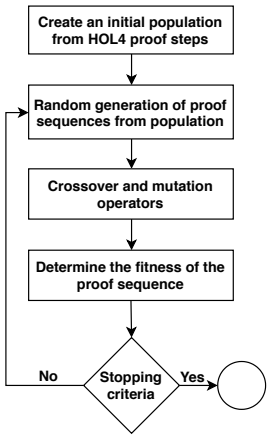

Figure 1: Flowchart of the GA for proofs searching and optimization in HOL4

The proof development process in HOL4 is interactive in nature and it follows the *lambda calculus proof representation*. Proofs in HOL4 are costructed 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. User then applies proof commands and tactics to solve the proof goal. The action resulting from a proof command and tactics is referred to as a HOL4 proof step (*HPS*) here. A *HPS* may either prove the goal or generates 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.

After proof development, HOL4 saves the proof scripts of a theory in a separate proof file. Inside a theory, a particular proof goal for a theorem or lemma depends on the specifications and it can be completed by applying the *HPS* in different orders. This makes it difficult to automatically find the proof for a goal or to carry out a brute force or pure random search for proof searching. 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.

The data available in HOL4 proof files is first converted into a proper computational format so that a GA 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 $|PSS|$ denotes the set cardinality. For example, consider that PS = {*RW, PROVE_TAC, FULL_SIMP_TAC, REPEATGEN_TAC, DISCH_TAC*}. The set {*RW, FULL_SIMP_TAC, REPEATGEN_ 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, ⟨{*RW, PROVE_TAC*}, {*FULL_SIMP_TAC*}, {*GEN_TAC, DISCH_TAC*}⟩ 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 four proof sequences.

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*}⟩

Algorithm 1 presents the pseudocode of the GA that can be used to find the proofs in the HOL4 theories that contain the *HPS* used for the verification of theorems and lemmas in those theories. An initial population (*Pop*) is first created from frequent *HPS* (*FHPS*) that are discovered with various sequential pattern mining (SPM) techniques [4]. From population, two random proof sequences (P_1 and P_2) are generated. Random proof sequences first goes through crossover operation where 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 continuous until randomly generated proof sequences match with the proof sequences from the *PD*. The fitness values guide the GA 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 presents the procedure for calculating the fitness value of a proof sequence.

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)
7:          Child ← Mutation(C)
8:          if Fitness(Child) < Fitness(P) then
9:              repeat
10:         else
11:             bFitness ← Fitness(Child)
12:             bChild ← Child
13:         end if
14:     until (Fitness(Child) = Fitness(P))
15:     return bFitness, bChild
16: end for
```

The fitness procedure 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*):

RP = MAP_EVERYTHING_TAC, RULE_ASSUM_TAC, X_GEN_TAC, SRW_TAC, AP_TERM_TAC, DISCH_TAC, DECIDE_TAC, RW_TAC
TP = POP_ASSUM, REAL_ARITH_TAC, X_GEN_TAC, COND_CASES_TAC, AP_TERM_TAC, RULE_ASSUM_TAC, X_GEN_TAC, RW_TAC

Algorithm 2 Fitness

Output: Integer that represents the fitness of a proof sequence (*Pseq*)

```
1:  procedure FITNESS(Pseq)
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
```

The *Fitness* procedure will return 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:

P_1 = SRW_TAC, MAP_EVERYTHING_TAC, X_GEN_TAC, AP_TERM_TAC, RULE_ASSUM_TAC, DISCH_TAC, DECIDE_TAC, RW_TAC
P_2 = REAL_ARITH_TAC, POP_ASSUM, X_GEN_TAC, COND_CASES_TAC, RW_TAC, RULE_ASSUM_TAC, X_GEN_TAC, AP_TERM_TAC

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 $cp = 4$:

P'_1 = SRW_TAC, MAP_EVERYTHING_TAC, X_GEN_TAC, COND_CASES_TAC, RW_TAC, RULE_ASSUM_TAC, X_GEN_TAC, AP_TERM_TAC
P'_2 = REAL_ARITH_TAC, POP_ASSUM, X_GEN_TAC, AP_TERM_TAC, RULE_ASSUM_TAC, DISCH_TAC, DECIDE_TAC, RW_TAC

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

Algorithm 3 Single Point Crossover

Output: Child proof sequence

```
1:  procedure SPC(P1, P2)
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) > Fitness(P2) 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 = *SRW_TAC, MAP_EVERYTHING_TAC, X_GEN_TAC, COND_CASES_TAC, RW_TAC, DISCH_TAC, DECIDE_TAC, RW_TAC*
P'_2 = *REAL_AIRTH_TAC, POP_ASSUM, X_GEN_TAC, AP_TERM_TAC, RULE_ASSUM_TAC, RULE_ASSUM_TAC, X_GEN_TAC, 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 = *SRW_TAC, POP_ASSUM, X_GEN_TAC, COND_CASES_TAC, RULE_ASSUM_TAC, DISCH_TAC, X_GEN_TAC, RW_TAC*
P'_2 = *REAL_ARITH_TAC, MAP_EVERYTHING_TAC, X_GEN_TAC, AP_TERM_TAC, RW_TAC, RULE_ASSUM_TAC, DECIDE_TAC, 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 are then checked and *UC* returns the sequence having the highest fitness.

Algorithm 4 Multi Point Crossover

Output: Child proof sequence

```
1:  procedure MPC(P1, P2)
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) > Fitness(P2)) then
14:         return P1
15:     else
16:         return P2
17:     end if
18: 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 which helps avoiding getting stuck in local optima. In *SM*, the selected location value is changed from its original value with some probability. This probability is 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 = *SRW_TAC, POP_ASSUM, X_GEN_TAC,* **DECIDE_TAC**, *RULE_ASSUM_TAC, DISCH_TAC, X_GEN_TAC, RW_TAC*

Algorithm 5 Uniform Crossover

Output: Child proof sequence

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

Algorithm 6 Standard Mutation

Output: Mutated child proof sequence

```
1:  procedure SM(P1)
2:      ind ← randomint(1, size)
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, this GA was unable to find the target proof sequence with *PIM* as it was only interchanging the values between two gene in the random proof sequence. To address this issue, we have modified the *PIM* procedure such that the two selected gene values are replaced with random *HPS* from the *Pop* rather than interchanging the values. For instance, by applying *PIM* on the proof sequence P_1, the following mutated proof sequence can be obtained:

P'_1 = *SRW_TAC,* **REWRITE_TAC**, *X_GEN_TAC, DECIDE_TAC, RULE_ASSUM_TAC,* **BETA_TAC**, *X_GEN_TAC, RW_TAC*

The reason to use more than one crossover and mutation operator 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.

Algorithm 7 Modified Pairwise Interchange Mutation

Output: Mutated child proof sequence

1: **procedure** MPIM(P_1)
2: $mp_1 \leftarrow$ randomint(1, length(P_1))
3: $mp_2 \leftarrow$ randomint(1, length(P_1)) ▷ $mp_1 \neq mp_2$
4: $ng, alter \leftarrow$ randomsample(Pop, 2)
5: $P_1[mp_1] \leftarrow ng$ ▷ ($P_1[mp_1] \neq$ ng)
6: $P_1[mp_2] \leftarrow alter$ ▷ ($P_1[mp_2] \neq$ alter)
7: return P_1
8: **end procedure**

4 RESULTS AND DISCUSSION

The GA with different versions of the crossover and mutation operators is implemented in Python and the code can be found at [5]. To evaluate the proposed approach, experiments were carried on a fifth generation Core *i*5 processor and 8 GB of RAM. Some initial and important results obtained by applying the GA on *PD* with different crossover and mutation operators are discussed in this section.

We investigate the performance of the proposed GA for finding the proofs of theorems in six HOL4 theories available in the library. These theories are: *Transcendental, Arithmetic, RichList, Number, Sort* and *Rational*. We selected ten theorems from each theory and in total, we have proof sequences for sixty theorems/lemmas in the *PD*. Table 2 lists some of the important theorems/lemmas from the theories. For example, *L*1 (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 sin when the given value is in the range $[0 - 2]$. $T10$ from the *R*ational theory is the dense theorem that proves that in between any two real numbers, there exists a rational number.

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

HOL Theory	No.	HOL4 Theorems
Transcendental	L1	⊢ ∀x. 0<=x∧xv<= inv(2) ==> exp(x) <= 1+2*x
	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)
Number	T5	⊢ ∀n m. (m <= n ==> (iSUB T n m = n - m)) ∧ (m < n ==> (iSUB F n m = n - SUC m))
	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])) ⊢ PERM = TC PERM_SINGLE_SWAP
	T8	
Rational	T9	⊢ ∀ x y. abs_rat (frac_add (rep_rat (abs_rat x)) y) = abs_rat (frac_add x y)
	T10	⊢ ∀ r1 r3. rat_les r1 r3 ==> ?rat_res r1 r2 ∧ rat_les r2 r3

The GA was run with the different crossover and mutation operators on considered theorems/lemmas ten times. Fitness values in Table 3 represents the total *HPS* that are used in the complete proof and this value is the same for respective theorems and lemmas in all crossover and mutations operators. The generations columns 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 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*. The average generations for the GA with different crossover and mutation operators to reach the target proof sequences in the whole dataset are shown in Table 4. *MPIM* is approximately ten times faster than *SM*. The possible reason is that *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*.

Table 3: Results for the proposed GA

T/L	C* & M*	Fit**	Generations	Time	C & M	Fit	Generations	Time
L1	SPC/SM	54	1903765	55.43	SPC/MPIM	54	314043	9.52
T1	SPC/SM	58	2103765	60.10	SPC/MPIM	58	334043	10.33
T2	SPC/SM	81	1947597	93.56	SPC/MPIM	81	392822	12.89
T3	SPC/SM	66	2473394	62.35	SPC/MPIM	66	191162	6.61
T4	SPC/SM	19	297179	4.72	SPC/MPIM	19	38307	0.93
T5	SPC/SM	23	501813	8.30	SPC/MPIM	23	33655	0.71
T6	SPC/SM	30	709484	13.09	SPC/MPIM	30	34776	0.79
T7	SPC/SM	17	264263	4.11	SPC/MPIM	17	21136	0.40
T8	SPC/SM	42	811951	28.49	SPC/MPIM	42	39302	1.41
T9	SPC/SM	23	554111	9.30	SPC/MPIM	23	45309	0.90
T10	SPC/SM	23	546136	9.21	SPC/MPIM	23	51552	1.01
L1	MPC/SM	54	1488005	27.21	MPC/MPIM	54	105521	3.29
T1	MPC/SM	58	1540467	35.93	MPC/MPIM	58	153644	5.01
T2	MPC/SM	81	1898305	80.38	MPC/MPIM	81	191699	7.69
T3	MPC/SM	66	1128636	31.54	MPC/MPIM	66	104784	3.60
T4	MPC/SM	19	358182	7.01	MPC/MPIM	19	24960	0.48
T5	MPC/SM	23	384539	7.19	MPC/MPIM	23	42750	0.83
T6	MPC/SM	30	738037	10.21	MPC/MPIM	30	73408	1.13
T7	MPC/SM	17	276087	5.32	MPC/MPIM	17	19997	0.43
T8	MPC/SM	42	1245801	25.67	MPC/MPIM	42	101795	2.52
T9	MPC/SM	23	411625	7.73	MPC/MPIM	23	27578	0.63
T10	MPC/SM	23	480625	8.26	MPC/MPIM	23	25314	0.55
L1	UC/SM	54	1652013	61.83	UC/MPIM	54	63277	1.86
T1	UC/SM	58	1682200	68.32	UC/MPIM	58	126097	2.92
T2	UC/SM	81	2348878	101.63	UC/MPIM	81	312328	8.21
T3	UC/SM	66	1662751	44.81	UC/MPIM	66	257215	7.48
T4	UC/SM	19	706950	11.12	UC/MPIM	19	20702	0.41
T5	UC/SM	23	819903	14.97	UC/MPIM	23	71614	1.37
T6	UC/SM	30	867183	17.21	UC/MPIM	30	74635	1.53
T7	UC/SM	17	321183	6.16	UC/MPIM	17	20263	0.42
T8	UC/SM	42	804969	20.53	UC/MPIM	42	29606	0.95
T9	UC/SM	23	625908	11.38	UC/MPIM	23	130303	2.50
T10	UC/SM	23	716950	13.07	UC/MPIM	23	90425	1.94

* Crossover and mutation, ** Fitness.

Population diversity greatly influences a GA's ability to pursue a fruitful exploration as it iterates from a generation to another [18]. 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

Table 4: Average total generation count

Crossover	Mutation	Ave. Generation Count
SPC	SM	25483429
MPC	SM	27094322
UC	SM	27145732
SPC	MPIM	2511045
MPC	MPIM	2321555
UC	MPIM	2448802

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 14.25 with a mean of 11.26 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 5 lists the calculated SD_t for all the proofs in the *PD* along with their mean and total time 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 ten 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*. We also checked the amount of memory used by GA (shown in Table 5) while searching for proofs. Moreover, we noticed that the GA using different crossover and mutation operators require approximately the same memory while searching for proofs and their optimization in *PD*.

Table 5: SD_t, mean and total time for the GA

C & M	Mean	SDt	Time	Memory
SPC/SM	9.87	52.82	629.68 s	4590 Mb
MPC/SM	11.50	61.99	834.87 s	4427 Mb
UC/SM	10.77	58.68	692.79 s	4816 Mb
SPC/MPIM	0.33	1.94	83.35 s	4831 Mb
MPC/MPIM	0.34	1.76	99.94 s	4766 Mb
UC/MPIM	0.33	1.57	48.17 s	4936 Mb

Next, we checked how much time on average the GA takes to find the *HPS* in random proof sequences that match with the *HPS* in the target sequence (shown in Figure 2). The runtime difference when applying the GA with various crossover and mutation operators to find the correct *HPS* in a proof sequence is negligible. It is observed that different crossover operators with *SM* takes more time than *MPIM*. The time to find the *HPS* increases for each following *HPS*. *SPC* takes more time than *MPC* and *UC* in *SM*. On the other hand, the runtime behavior using different crossover operators with *MPIM* is uniform.

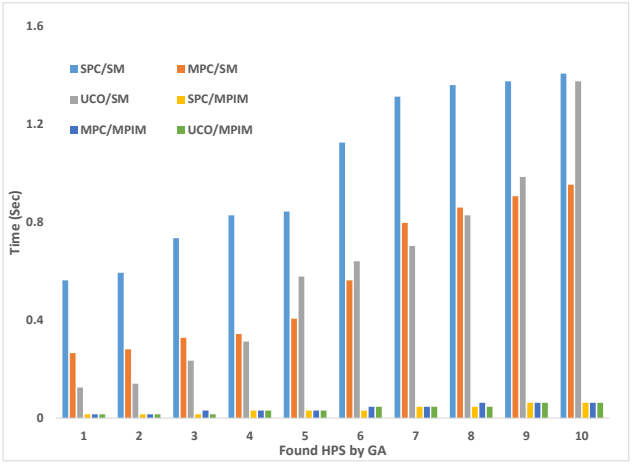

Figure 2: Time used by GA to find the first ten matched *HPS*

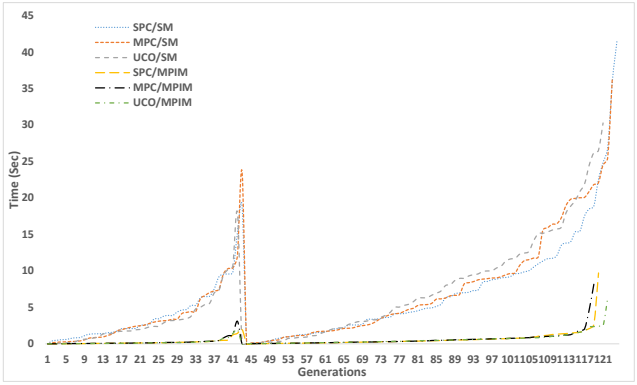

Figure 3: Total time and generations for the *PSF* theorem

The longest proof in the *PD* is for the theorem *T*2 (positive value of sin) and it consists of 81 *HPS*. Here we call this theorem *PSF*. The runtime of the GA to find all matched 81 *HPS* in *PSF* with different crossover and mutation operators is shown in Figure 3. Those generations are shown on the x-axis where the GA was 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, the GA was unable

to find the same *HPS* in a random proof sequence and *PSF*. The time 0 in generations 41-45 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. In each generation of the GA, the probability to find the complete correct proof for *PSF* with different crossover and *SM* is approximately 4.84×10^{-6} and with different crossover and *MPIM* is 5.04×10^{-5}. This is much better than proof searching with a pure random search. For example, the probability for a pure random search to find a valid proof for *PSF* can be: $121 \times 76/\ 60^{81} \approx 1.03 \times 10^{-145}$. For the theorems with smaller (fitness of 10) proof sequences, the probability is in the magnitude of 10^{-14}.

Overall, it was observed through various experiments that the proposed GA is able to optimize and automatically find the correct proofs for theorems/lemmas in different HOL4 theories and thus in turn reduce the memory usage. Besides HOL4, this approach can also be used in other proof assistants such as Coq and PVS. These preliminary results indicate that the research direction of linking and integrating evolutionary algorithms with proof assistants is worth pursuing. This approach may have a considerable impact to advance and accumulate human knowledge, especially in the fields of formal logic and computation.

5 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 an evolutionary approach in this paper for the possible linkage between evolutionary algorithms, such as GAs, with theorem provers, such as HOL4, to facilitate the proof finding and development process. A GA with different crossover and mutation operators is proposed to optimize and find the correct proofs in HOL4 theories. The performance of the GA with three crossover and two mutation operators was compared on the basis of fitness, population diversity measures, time and memory.

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 a compound proof strategies for guiding the proofs of new conjectures. Moreover, stochastic optimization techniques, such as particle swarm optimization, and heuristic search algorithms, such as monte carlo tree search can be considered for proofs searching. Another direction is to take advantage of Curry-Howard isomorphism for sequent calculus [20] that provides a direct relation between programming and proofs, where finding proofs can be viewed as writing programs. With such correspondence, a variant of GA called linear genetic programming can be used to write programs (proofs) and HOL4 proof assistant for simplification and verification by computationally evaluating the programs.

REFERENCES

[1] Y. Bertot and P. Casteran. 2003. *Interactive theorem proving and program development: Coq'Art: The calculus of inductive construction.* Springer.

[2] H. Duncan. 2007. *The use of data-mining for the automatic formation of tactics.* Ph.D. Dissertation. University of Edinburgh, UK.

[3] M. Färber and C. E. Brown. 2016. Internal guidance for Satallax. In *Proceedings of IJCAR, 2016 (LNCS)*, Vol. 9706. Springer, 349–361.

[4] P. Fournier-Viger, J. C. W. Lin, R. U. Kiran, Y. S. Koh, and R. Thomas. 2017. A survey of sequential pattern mining. *Data Science and Pattern Recognition* 1(1) (2017), 54–77.

[5] GA implementation. [n.d.]. Available at: github.com/muhammadzohaibnawaz/GAHOL4.

[6] T. Gauthier, C. Kaliszyk, and J. Urban. 2017. TacticToe: Learning to reason with HOL4 tactics. In *Proceedings of LPAR, 2017 (EPiC Series in Computing)*, Vol. 46. 125–143.

[7] O. Hasan and S. Tahar. 2015. Formal verification methods. In *Encyclopedia of Information Science & Technology, 3rd edition.* IGI Global, 7162–7170.

[8] J. H. Holland. 1975. *Adaptation in natural and artificial systems.* University of Michigan Press, Ann Arbor, MI, USA.

[9] S. Y. Huang and Y. P. Chen. 2017. Proving theorems by using evolutionary search with human involvement. In *Proceedings of CEC, 2017.* IEEE, 1495–1502.

[10] G. Irving, C. Szegedy, A. A. Alemi, N. Eén, F. Chollet, and J. Urban. 2016. DeepMath - Deep sequence models for premise selection. In *Proceedings of NIPS, 2016.* ACM, 2235–2243.

[11] C. Kaliszyk, F. Chollet, and C. Szegedy. 2017. HolStep: A machine learning dataset for Higher-Order Logic theorem proving. *CoRR* abs/1703.00426 (2017).

[12] C. Kaliszyk, L. Mamane, and J. Urban. 2014. Machine Learning of Coq Proof Guidance: First Experiments. In *Proceedings of SCSS, 2014 (EPiC Series in Computing)*, Vol. 30. 27–34.

[13] J. R. Koza. 1993. *Genetic programming - On the programming of computers by means of natural selection.* MIT Press.

[14] M. Mitchell. 1996. *An introduction to genetic algorithms.* MIT Press.

[15] M. S. Nawaz, M. Malik, Y. Li, M. Sun, and M. I. Lali. 2019. A survey on theorem provers in Formal methods. arXiv:cs.SE/1912.03028.

[16] M. S. Nawaz and M. Sun. 2018. A formal design model for genetic algorithms operators and its encoding in PVS. In *Proceedings of BDIOT, 2018.* ACM, 186–190.

[17] M. S. Nawaz, M. Sun, and P. Fournier-Viger. 2019. Proof guidance in PVS with sequential pattern mining. In *Proceedings of FSEN, 2019 (LNCS)*, Vol. 11761. Springer, 45–60.

[18] A. L. Nsakanda, W. L. Price, M. Diaby, and M. Gravel. 2007. Ensuring population diversity in genetic algorithms: A technical note with application to the cell formation problem. *European Journal of Operational Research* 178 (2007), 634–638.

[19] S. Owre, N. Shankar, J. M. Rushby, and D. W. J. Stringer-Calvert. 2001. *PVS system guide, PVS prover guide, PVS language reference.* Technical Report. SRI International.

[20] J. E. Santo. 2015. Curry-Howard for sequent calculus at last!. In *Proceedings of TLCA, 2015 (LIPIcs)*, Vol. 38. 165–179.

[21] K. Slind and M. Norrish. 2008. A brief overview of HOL4. In *Proceedings of TPHOL, 2008.* Springer, 28–32.

[22] L. A. Yang, J. P. Liu, C. H. Chen, and Y. P. Chen. 2016. Automatically proving mathematical theorems with evolutionary algorithms and proof assistants. In *Proceddings of CEC, 2016.* IEEE, 4421–4428.