An Evolutionary/Heuristic-Based Proof Searching Framework for Interactive Theorem Prover

M. Saqib Nawaz[a], M. Zohaib Nawaz[b,c], Osman Hasan[b], Philippe Fournier-Viger[a,*], Meng Sun[d]

[a]*School of Humanities and Social Sciences, Harbin Institute of Technology (Shenzhen), Shenzhen, China*
[b]*School of Electrical Engineering and Computer Science, National University of Sciences and Technology (NUST), Islamabad, Pakistan*
[c]*Department of Computer Science and IT, University of Sargodha, Sargodha, Pakistan.*
[d]*School of Mathematical Sciences, Peking University, Beijing, China*

Abstract

The proof development process in interactive theorem provers (ITPs) requires the users to manually search for proofs by interacting with proof assistants. The activity of finding the correct proofs can become quite cumbersome and time consuming for users. To make the proof searching process easier in proof assistants, we provide an evolutionary/heuristic-based framework. The basic idea for the framework is to first generate random proof sequences from a population of frequently occurring proof steps that are discovered with sequential pattern mining. Generated proof sequences are then evolved till their fitness match the fitness of the target (or original) proof sequences. Three algorithms based on the proposed framework are developed using the Genetic Algorithm (GA), Simulated Annealing (SA) and Particle Swarm Optimization (PSO). Extensive experiments are performed to investigate the performance of the proposed algorithms using the HOL4 proof assistant. Results have shown that the proposed algorithms can efficiently evolve the random sequences to obtain the target sequences. In comparison, PSO performed better than SA and SA performed better than GA. In general, the experimental results suggest that combining evolutionary/heuristic algorithms with proof assistants allow efficient support for proof finding/optimization.

Keywords: Fitness, Genetic Algorithm, HOL4, Particle Swarm Optimization, Proof Searching Framework, Simulated Annealing

*Corresponding author
Email addresses: `msaqibnawaz@hit.edu.cn` (M. Saqib Nawaz), `mnawaz.mscs16seecs@seecs.edu.pk` (M. Zohaib Nawaz), `osman.hasan@seecs.edu.pk` (Osman Hasan), `philfv8@yahoo.com` (Philippe Fournier-Viger), `sunmeng@math.pku.edu.cn` (Meng Sun)

1. Introduction

Theorem proving is a popular formal verification method that is used for the analysis of both hardware and software systems. Theorem proving consists of two main activities: (1) Using appropriate mathematical logic for the modeling of systems that need to be analyzed, and (2) Proving important/critical properties for the system using *theorem provers* [25] based on deductive reasoning. The initial objective to develop theorem provers was to enable mathematicians to develop formal proofs for theorems using computer tools. Thus, theorem provers allowed mathematicians to set up mathematical theories, define important properties and do logical reasoning to verify the properties. However, these mechanical tools have evolved in the 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 [30].

There are two main classes of theorem provers: Automated theorem provers (ATPs) and interactive theorem provers (ITPs). ATPs are based on propositional and first-order logic (FOL). They deal with the development of computer tools that can automatically perform logical reasoning. However, the main limitation of ATPs is their inability to model and reason about complex systems as the logic they use is FOL, which is less expressive in nature. Additionally, the problem of search space (combinatorial) explosion does not allow ATPs to be scaled to the large mathematical libraries. ITPs, on the other hand, are based on higher-order logic (HOL). Compared to FOL, HOL allows rich logical formalisms and provides ITPs the expressiveness that is required to define and reason about complex systems. However, this expressive power leads to the undecidability problem in ITPs, i.e., the reasoning process cannot be made fully automated in HOL and some sort of human guidance is always needed during the process of proof searching and development. This is the reason that ITPs are also known as *proof assistants*. Some well-known proof assistants are HOL4 [58], Coq [6], Isabelle/HOL [50], PVS [52] and Lean [10].

In ITPs, the user guides the proof process by providing the proof goal and by applying proof commands and tactics to prove the goal [66]. Generally, the user in ITP needs to add mathematical details and carry out lots of repetitive work while verifying a non-trivial theorem (proof goal) [20, 45]. This heavy interaction makes the overall proving process a laborious and a time consuming activity for a user. For example, the list of formalizing 100 classic mathematical theorems using ITPs is maintained at [63].The writing and verification process for many of these theorems required several months or even years to complete (approximately 20 years for the Kepler conjecture proof in HOL Light [24] and twice as much for the Feit-Thompson theorem in Coq [21]) and the complete proofs contain thousands of low-level inference steps. Other real-life systems where ITPs are used for the analysis and verification include the compilers [39, 40], microkernels [23, 37], distributed systems [64], file systems [4, 9] and conference system [34].

The specifications and proof goals for a given system or problem is collectively called a theory in ITPs. Developing the formal proof for a goal mainly depends on the specifications available in a theory or a set of theories along with different combinations of existing proof commands, inference rules, intermediate states and tactics. Thus, the proof development process is a search problem, where the aim is to find a sequence of deductions that leads from presumed facts to the given conjecture (unproved proof goal). However, a theory often contains many definitions and theorems [8, 32], so 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. The proof scripts for theories in ITPs can be combined together to develop a more complex computer-understandable corpora. With the recent evolution in computing systems, it is now possible to use artificial intelligence (AI) techniques, such as machine learning, data mining and deep learning, on such corpora for guiding the proof search process, for proof automation and for the development of proof tactics/strategies, as indicated in several studies [3, 14, 19, 30, 31, 48, 69]. Recent studies [5, 27, 41, 53, 55, 59, 66] focused on developing large-scale datasets and learning environments, based on machine learning and neural networks, to search for proofs and to automate the interaction in different ITPs.

For proof searching and optimization in HOL4, we proposed an evolutionary approach [49]. The basic idea of the approach was to use a Genetic Algorithm (GA) for proof searching where an initial population (a set of potential solutions) is first created from frequent HOL4 proof steps that are discovered using a sequential pattern mining (SPM)-based proof guidance approach [48]. 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 match the fitness of original (target) proofs for the considered theorems/lemmas. The reason for using different crossover and mutation operators was to compare their effect on the overall performance of GAs in proof searching. The approach was successfully used on six theories taken from the HOL4 library. This proof searching approach [49] was quite efficient in evolving random proofs efficiently. However, alternative proof searching approaches could be developed. Thus efforts are made in this work to further investigate the applicability of evolutionary and heuristic techniques in the HOL4 proof assistant. This paper extends the work published in [49] with the following contributions:

1. Two heuristic-based approaches are proposed, where Simulated Annealing (SA) and Particle Swarm Optimization (PSO) are used for proof searching and optimization in HOL4. The performance of SA and PSO are compared with that of GA [49] for various parameter values. It is found that both SA and PSO perform better than GA for proof finding and optimization and PSO further outperforms SA. Moreover, we elaborate on how pure

random search and brute force approach are not suitable for this task.

2. The Headless Chicken Test is performed to examine the usefulness of crossover operators in GA [49]. In the test, GA with normal crossovers are compared with the same GA that employs the randomized version of the crossovers. It is observed that GA with normal crossovers and GA with random crossovers perform similarly with negligible difference. Moreover, convergence speed of the proposed proof searching approaches is investigated.
3. For experimental evaluation, we have selected 14 theories from the HOL4 library out of which, we selected 300 different theorems and lemmas in total. 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 briefly discusses the HOL4 theorem prover. Section 3 provides a brief introduction for the evolutionary/heuristic algorithms (GA, SA, and PSO). Section 4 presents the proposed proof searching approaches, where GA, SA and PSO 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 discusses the related work. Finally, Section 7 concludes the paper while highlighting some future directions.

2. HOL4

HOL4 [58], the successor of HOL98, is a member of the HOL theorem prover family. Other three members of the family are HOL Light, ProofPower and HOLZero. HOL4 utilizes the simple type theory along with Hindley-Milner polymorphism for the implementation of HOL. The logic in HOL4 is represented in the meta language (ML), which is a strongly-typed functional programming language. Theorems/lemmas are defined using the ML abstract data types and the only way to interact with the HOL4 proof assistant is by executing ML procedures that operate on values of these data types. A theory in HOL4 is a collection of definitions, axioms, types, theorems and proofs, as well as tactics [2]. Users can reload a theory into the HOL4 and can use the corresponding definitions and theorems in the theory right away. Here, we provide a simple example of the factorial function as a case study.

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 ($\wedge$). An elaborative example for the factorial function that calculates the factorial of 3 is provided for better understanding 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 the ML function g, which starts a fresh goalstack. For the factorial function, suppose we want to prove that for any natural number n, the factorial function will always be greater than 0, i.e., $\forall n.$ $(0 <$ factorial $n)$. Using g and the HOL notation, this statement can be specified as:

g‘!(n:num). (0 < factorial n)’

Theorems/lemmas can be proved in HOL4 by two ways: (1) forward method and (2) backward method. In the forward proof method, 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 aforementioned theorem for factorial is proved using the backward proof method. The first step is to remove the *forall-quantifiers* using the *GEN_TAC* tactic. The second step is to apply induction on natural number n. This divides the main theorem (proof goal) into two sub-goals. The base case (first sub-goal) is proved by using the *RW_TAC* tactic with the definition of $factorial$ function. For the recursive case (second sub-goal), *RW_TAC* is again used to expand the factorial definition with respect to its specifications. In the *arithmeticTheory*, there is a theorem called $LESS\_MULT2$ that proves that for any two natural numbers (n and m), their product (multiplication) will always be greater than 0, i.e., $\forall n, m.$ $(0 < n \times m)$. *MATCH_MP_TAC* is used with $LESS\_MULT2$ to further simplify the sub-goal, which is then proved by using the *RW_TAC*. 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 []);
```

where e is an ML function that is used in HOL4 before applying any proof step/tactic.

3. Evolutionary/Heuristic Algorithms

Evolutionary and heuristic algorithms optimize a given problem by iteratively improving the candidate solutions with respect to a given measure of quality (objective function). These algorithms are now used to solve not only the optimization problems, but are popular in various other fields too, such as bioinformatics, scheduling applications, artificial intelligence, robotics and control engineering. Next, we describe the evolutionary/heuristic algorithms that we use in this work.

3.1. Genetic Algorithms

GAs [26, 44], that are based on biological evolution principles, can explore a huge search space (population) to find near optimal solutions to complex problems. The four main steps of a GA are: (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. For a specific problem, GA starts by randomly generating a set of chromosomes to form a population. It then iteratively evolves the chromosomes with two operators and evaluates them using a fitness function f. The function generally takes a chromosome as parameter and finds a score that indicates how good the candidate solution is.

The crossover operator in GA plays an important role and is used to guide the search toward the best solutions. A crossover operator combines two selected chromosomes (called parents chromosomes) to produce potentially better chromosomes (called child chromosomes). If appropriate crossing point(s) are chosen, then the combination of sub-chromosomes (also known as building blocks) from parents may produce better childs. The mutation operator is applied to a parent chromosome and carries out some random changes to one or more genes. The mutation process 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 help GAs to make progress during the evolution and search process.

3.2. Simulated Annealing

Simulated annealing (SA) [7, 11] is a well-known metaheuristic method to solve both unconstrained and bound-constrained optimization problems. SA 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 four main steps of SA to solve any problem are: (1) problem configuration, (2) neighborhood configuration, (3) objective function and (4) cooling/annealing schedule. More specifically, SA starts by creating a random initial solution. In each iteration, the current solution is replaced by a random "neighbor" solution. The neighbor solution is selected if it is better than the previous solution or with a probability that depends on the difference between the corresponding function values and on a global parameter T (called the temperature), whose value is decreased gradually after each iteration.

For some optimization and computational problems, SA performs better than GA because it can find the optimal solution with point-by-point iteration rather than a search over a population of solutions. On the other hand, SA is very similar to the Hill-Climbing algorithm with one main difference: at high temperature, SA switches to a worse neighbor that helps in avoiding the local optima.

3.3. Particle Swarm Optimization

Particle swarm optimization (PSO) [36] is a population-based stochastic optimization algorithm. The development of PSO was inspired from the behavioral nature of collective fish schooling and birds flocking. In PSO, each individual (called particle) can roam around in the search space with a velocity factor that constantly gets updated by each particle's personal experience and the experience of the best particle (which, in most cases, is a neighbor). The four main steps of PSO are: (1) initialization of parameters and swarm positions, (2) updating the fitness of each swarm particle, (3) updating p_{best} and g_{best} and (4) updating the particles' velocity and position.

For a specific problem, PSO starts by randomly distributing a set of particles in the search space and evaluates these particles using a fitness function. The function takes a particle as input and returns a score indicating how good the particle is (i.e. how close it is to the problem's goal). In each iteration, the particles are updated while keeping track of the following two parameters: (1) the best solution of every particle (known as local best (p_{best})), and (2) the best fitness value (known as global best (g_{best})) that is obtained so far, among all neighbors. Particles gradually move towards the optimum solutions by exchanging information with each other through p_{best}, g_{best} and a velocity factor. In this paper, we use GA, SA and SPO for the problem of proof searching in HOL4 and more details about them are provided later in Section 4.

4. Proposed Proof Searching Framework

We propose a generic evolutionary/heuristic-based framework that can be used to find and optimize the proofs of theorems/lemmas in HOL4. We name this framework PSF and its schematic is shown in Figure 1.

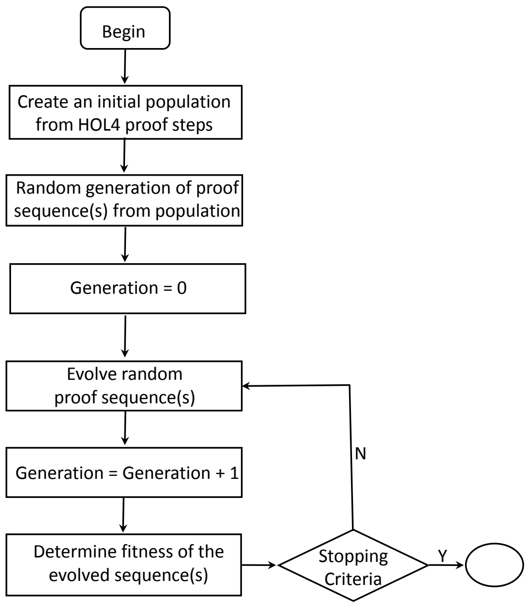
Begin
Create an initial population from HOL4 proof steps
Random generation of proof sequence(s) from population
Generation = 0
Evolve random proof sequence(s)
Generation = Generation + 1
Determine fitness of the evolved sequence(s)
Stopping Criteria
N
Y

Figure 1: The proposed proof searching framework (PSF)

The interactive proof development process in HOL4 follows the *lambda calculus proof representation*. Formal proofs in HOL4 are built 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 cooperates with the HOL4 proof assistant and applies proof commands and tactics to solve the proof goal. HOL4 also offers automatic proof procedures. A tactic in HOL4 is a function that takes a proof goal and returns a sub-goal(s) 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 complete proof script of a particular theorem or lemma depends on the application of the *HPS* in a specific order makes automatic proof searching 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.

We propose to convert the data available in HOL4 proof files to a proper computational format so that evolutionary/heuristic algorithms 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 $PS = \{$*FULL_SIMP_TAC, PROVE_TAC, RW, DISCH_TAC, REPEAT_GEN_TAC, REWRITE_TAC*$\}$. Now, the set $\{$*FULL_SIMP_TAC, RW, DISCH_TAC, REWRITE _TAC*$\}$ is a proof step set that contains four *HPS*. 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, $\langle\{$*FULL_SIMP_TAC, PROVE_TAC*$\}$, $\{$*RW*$\}$, $\{$*DISCH_TAC, REPEAT_GEN_TAC, REWRITE_TAC*$\}\rangle$ is a proof sequence, which has three *PSS* and six *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.

Table 1: A sample proof dataset

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

4.1. PSF-GA

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

An initial population (*Pop*) is first created from frequent *HPS* (*FHPS*) that are discovered with various sequential pattern mining (SPM) techniques [15]. From the initial population, two random proof sequences (P_1 and P_2) are generated. Both P_1 and P_2 go through the crossover operation where the child proof sequences are generated and their fitness are evaluated. The child having the better fitness value goes through the mutation operation that generates one mutated child sequence. If the fitness of the mutated child sequence 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

Algorithm 1 Flow of the GA

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

```
1: $Pop \leftarrow FHPS$
2: for each $P \in PD$ do
3:     $P_1 \leftarrow$ randomseq($Pop$, length($P$))
4:     $P_2 \leftarrow$ randomseq($Pop$, length($P$))          $\triangleright P_1 \neq P_2$
5:     repeat
6:         $C \leftarrow Crossover(P_1, P_2, P)$
7:         $Child \leftarrow Mutation(C)$
8:         if $Fitness(Child, P) < Fitness(P, P)$ then
9:             repeat
10:        else
11:            $bFitness \leftarrow Fitness(Child, P)$
12:            $bChild \leftarrow Child$
13:        end if
14:    until $(Fitness(Child, P) = Fitness(P, P))$
15:    return $bFitness, bChild$
16: end for
```

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

the original (target) proof sequence. Algorithm 2 presents the procedure for calculating the fitness value of a proof sequence.

In each generation of GA, the priority of the randomly generated proof sequences are ranked according to the fitness values calculated through the procedure listed in Algorithm 2. This procedure evaluates the resemblance of a given solution to the optimum solution (in our case, the target solution) by comparing each gene i of a random proof sequence (*Pseq*) with the genes of the target (P). The fitness (represented as f) is set to 0 initially and increased by 1 if the genes in *Pseq* and P are equal. This is explained with a simple example. Consider the following random proof sequence (*RP*) and the target sequence (*TP*):

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

The *Fitness* procedure returns 3 as three *HPS* are the same in both sequences (at Positions 2, 5 and 7 respectively).

Algorithm 3, 4 and 5 present the pseudocode of the three crossover operators. The symbol **o** in these algorithms represents the concatenation. Simple examples are used next to explain these three crossover operators. Let P_1 and P_2 be:

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

Consider that both sequences have the same length, denoted as n. Let one position cp ($1 \leq cp \leq n$) be randomy chosen as crossing point. For $cp = 5$, single point crossover (*SPC*) generates the following child sequences for P_1 and P_2:

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

Child proof sequences fitness values are then checked and *SPC* returns the child proof sequence with the highest fitness value.

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($P_1, P_2, P$)
 2:     $size \leftarrow$ min(length($P_1$), length($P_2$))
 3:     $cp \leftarrow$ randomint(1, $size$)                              ▷ $(1 \leq cp \leq size)$
 4:     $P_1 \leftarrow P_1[1, cp]$ o $P_2[cp + 1$, length($P_2$)]
 5:     $P_2 \leftarrow P_2[1, cp]$ o $P_1[cp + 1$, length($P_1$)]
 6:     if ($Fitness(P_1, P) > Fitness(P_2, P)$) then
 7:         return $P_1$
 8:     else
 9:         return $P_2$
10:     end if
11: end procedure
```

With the multi point crossover (MPC) operator, two crossing points are selected. Let cp_1 and cp_2 represent two crossing points ($1 \leq cp_1 < cp_2 \leq n$). For $cp_1 = 3$ and $cp_2 = 5$, the new child proof sequences generated by *MPC* for P_1 and P_2 are:

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

The fitness of child proof sequences are evaluated last and MPC returns the child proof sequence that has the highest fitness value.

In uniform crossover (*UC*), each element (gene) of the proof sequences is assigned to the child sequences with a probability value p. For two parent sequences, if p is 0.5, then the child sequences will have approximately half of the genes from the first parent and the other half from the second parent. Two child proof sequences for P_1 and P_2 after UC with $p = 0.5$ are:

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

The randomized nature of UC and the probability can generate different child in different runs. Finally, the fitness of child proof sequences are checked and the UC returns the child with the high fitness value.

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($P_1, P_2, P$)
2:     $size \leftarrow$ min(length($P_1$), length($P_2$))
3:     $cp_1 \leftarrow$ randomint(1,$size$)
4:     $cp_2 \leftarrow$ randomint(1,$size$)
5:     if $cp_2 > cp_1$ then
6:         $cp_2 \leftarrow cp_2 + 1$
7:     else
8:         $cp_2 \leftarrow cp_1$
9:         $cp_1 \leftarrow cp_2$
10:    end if
11:    $P_1 \leftarrow P_1[1, cp_1]$ o $P_2[cp_1 + 1, cp_2]$ o $P_1[cp_2 + 1$,length($P_1$)]
12:    $P_2 \leftarrow P_2[1, cp_1]$ o $P_1[cp_1 + 1, cp_2]$ o $P_2[cp_2 + 1$,length($P_2$)]
13:    if ($Fitness(P_1, P) > Fitness(P_2, P)$) then
14:        return $P_1$
15:    else
16:        return $P_2$
17:    end if
18: end procedure
```

In GAs, the mutation operation is used after the crossover operation. The standard mutation (*SM*) operator can add random information to the search and evolution process to avoid getting stuck in a local optimum. Algorithm

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($P_1, P_2, P$)
2:     $size \leftarrow$ min(length($P_1$), length($P_2$))
3:     $p \leftarrow 0.5$
4:     for $i$ in range($size$) do
5:         if unifromreal[0,1] $\leq p$ then
6:             $P_1[i] \leftarrow P_2[i]$
7:             $P_2[i] \leftarrow P_1[i]$
8:         end if
9:     end for
10:    if ($Fitness(P_1, P) > Fitness(P_2, P)$ then
11:        return $P_1$
12:    else
13:        return $P_2$
14:    end if
15: end procedure
```

6 presents the pseudocode for the *SM* operator. It first selects a location and changes it with some probability (called mutation probability) denoted as p_m. Here for a proof sequence, a randomly chosen genes value i is replaced by a random *HPS* from the *Pop*. For example, a mutation of P_1 is:

P_1' = *MAP_EVERYTHING_TAC, SRW_TAC, AP_TERM_TAC, X_GEN_TAC,* **REWRITE_TAC**, *RULE_ASSUM_TAC, RW_TAC, DECIDE_TAC*

Algorithm 6 Standard Mutation

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

```
1: procedure SM($P_1$)
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
```

The pairwise interchange mutation (*PIM*) operator, a variant of *SM*, selects and interchanges two arbitrary genes in a proof sequence. But for proof searching, we empirically observed that a GA was unable to find the target proof sequence with *PIM*. The reason was that *PIM* only interchanges the genes at two selected locations in the random proof sequence. To address this problem, 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, with modified *PIM* (listed in Algorithm 7) on the proof sequence P_1, the following mutated proof sequence can be obtained:

P_1' = *MAP_EVERYTHING_TAC, SRW_TAC,* **RW**, *X_GEN_TAC, DISCH_TAC, RULE_ASSUM_TAC,* **REWRITE_TAC**, *DECIDE_TAC*

Algorithm 7 Modified Pairwise Interchange Mutation

Input: P_1: A proof sequence
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
```

The main reason of using more than one crossover and mutation operators is to investigate their effect on the overall performance of the PSF-GA. Note that random proof sequence(s) goes through crossover and mutation operators with a probability of 1 in each generation. This reduces the total number of iterations performed by the PSF-GA.

4.1.1. Illustrated Example

Table 1 is used to explain the working of PSF-GA. In Table 1, we have five proof sequences and the population (Pop) contains 18 distinct HPS. Consider the original proof sequence P = *GEN_TAC, CONJ_TAC, MP_TAC* and Fitness(P, P) = 3. Two random proof sequences (P_1 and P_2) are generated from the Pop. Suppose that P_1 = *PROVE_TAC, CASES_TAC, DISCH_TAC* and P_2 = *GEN_TAC, MP_TAC*, RW. Then assume that SPC is used for crossing with $cp = 2$. New proof sequences are obtained as P_1 = *PROVE_TAC, CASES_TAC, RW* and P_2 = *GEN_TAC, MP_TAC, DISCH_TAC*. As one HPS at the first position is the same in P and P_2, so Fitness(P_2, P) = 1 and Fitness(P_1, P) = 0. Thus, P_2 is selected for the mutation operation.

Suppose that SM is selected and the random number $ind = 3$ is generated, and a randomly selected HPS from Pop is $alter$ = *MP_TAC*. So new P_2 = *GEN_TAC, MP_TAC, MP_TAC*. Now two HPS are the same in P and P_2 (at the first and third position). Hence, Fitness(P_2, P) = 2. As Fitness(P, P) > Fitness(P_2, P), so the new P_1 and P_2 will again go through the crossover and mutation operations. This process continues till the fitness of one random proof sequence (either P_1 or P_2) matches with the fitness of P.

4.2. PSF-SA

Algorithm 8 presents the pseudocode of the PSF-SA that is used to find the proofs in HOL4 theories. As for PSF-GA, an initial population (Pop) is first created from *FHPS* in PSF-SA. From *Pop*, a random proof sequence (PS) is then generated that passes 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*. PSF-SA consists of two main procedures, *Fitness* and *Get_Neighbor* (GN), which are explained next.

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: $Pop \leftarrow FHPS$
 2: for each $P \in PD$ do
 3:     $OF \leftarrow Fitness(P, P)$
 4:     $PS \leftarrow$ randomseq($Pop$, length($P$))
 5:     $BF \leftarrow Fitness(PS, P)$
 6:     if $BF \geq OF$ then
 7:         return $PS$
 8:     end if
 9:     while ($Temp > Temp\_min$) do
10:         $NS \leftarrow get\_neighbor(PS)$
11:         $NF \leftarrow Fitness(NS, P)$
12:         if $NF == OF$ then
13:             return NS
14:         end if
15:         if $NF > BF$ then
16:             $PS \leftarrow NS$
17:             $BF \leftarrow NF$
18:         end if
19:         $ar \leftarrow exp(\frac{T}{1+T})$
20:         if $ar >$ randomuniform$(0, 10)$ then
21:             $PS \leftarrow NS$
22:             $BF \leftarrow NF$
23:         end if
24:         $Temp \leftarrow Temp \times \alpha$
25:     end while
26:     return $PS$
27: end for
```

Fitness values guide the PSF-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 *Get_Neighbor*. For a proof sequence, a randomly chosen gene value i is replaced by a random *HPS* from the current population *Pop*. Note that the *SM* 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 is applied, the fitness of the randomly generated proof sequence is compared with its neighbor sequence. 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 the temperature (*Temp*) parameter. Finally, the *Temp* value is decreased according to 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 annealing process is repeated (Steps 9-24 in Algorithm 8) until the fitness of the random proof sequence matches with that of 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. Note that the annealing process distinguishes SA from GAs.

The second main concept in SA, besides the annealing process, is the acceptance probability. In each iteration, SA evaluates the fitness of the new solution. If the fitness of the new solution is not better than the previous solution, SA can still select the new solution with a probability called the *acceptance probability*. *Acceptance probability* governs whether to switch to the worst solution or not. This allows the SA to explore other solutions to avoid getting stuck at the local optimum. For this purpose, we chose the *acceptance probability* by using the following acceptance rate (AR) formula:

$$AR = exp(\frac{Temp}{1 + Temp}) \tag{1}$$

The next section describes experiments to examine the effect of AR on the performance of PSF-SA. From the simulation results, it was observed that this parameter affects the performance of PSF-SA, but it is negligible.

4.2.1. Illustrated Example

The working of PSF-SA is explained using Table 1. Consider the original proof sequence P = *GEN_TAC, CONJ_TAC, MP_TAC*, and hence OF =

Fitness(P, P) = 3. One random proof sequence (PS) is generated from *Pop*. Suppose *PS = PROVE_TAC, CASES_TAC, DISCH_TAC*, so BF = Fitness(PS, P) = 0. The proof sequence PS goes through the $Get\_Neighbor$ procedure. In the $Get\_Neighbor$, suppose *ind* = 2 and *alter* = *CONJ_TAC*, so NS (the neighbor of PS) = *PROVE_TAC, CONJ_TAC, DISCH_TAC* and $NF = 1$ as one HPS at the second position matches in both NS and P. As $NF > BF$, so NS is assigned to PS and new *PS = PROVE_TAC, CONJ_TAC, DISCH_TAC* with $BF = 1$.

NS was assigned to PS as $NF > BF$. If $NF \leq BF$ after the $Get\_Neighbor$ procedure, then SA can still replace the PS with NS if the probability calculated using Equation (1) is greater than a random number generated in the range (0, 10). Finally, the value of $Temp$ is decreased with the factor α. The above process continues till the value of $Temp$ becomes less than or equal to $Temp\_min$. In the next section, we discuss the optimal values we selected for $Temp$, $Temp\_min$ and α.

4.3. PSF-PSO

Algorithm 10 presents the pseudocode of the PSF-PSO that is used to find the proofs in HOL4 theories.

Following the same process, an initial population (Pop) is first created from *FHPS*. From this population, a random proof sequence (PS) that represents a particle is generated. In each iteration, the position of the particle is updated by adding the updated velocity to the current particle position. This process continues until the particle fitness is equal to the fitness of the target proof particle from the *PD*.

In PSO, the general equations for updating the velocity and position of a particle are:

$$v_i^{t+1} = (w \times v_i^t) + (c_1 \times r_1(0,1) \times (p_{best} - x_i^t)) + (c_2 \times r_2(0,1) \times (g_{best} - x_i^t)) \tag{2}$$

$$x_i^{t+1} = x_i^t + v_i^{t+1} \tag{3}$$

where v_i^t and x_i^t are the current velocity and position, respectively, of a particle, $v_i^{(t+1)}$ and x_i^{t+1} are the updated velocity and position, respectively, at time $t+1$. The three weighting coefficients w, c_1 and c_2 are the acceleration constants for cognitive and social components. Whereas r_1 and r_2 represent random number i in the range (0,1).

In each iteration of PSO, the first term $(w \times v_i^t)$ in Equation (2) includes the previous velocity and makes up the momentum component. The second term $(c_1 \times r_1(0,1) \times (p_{best} - x_i^t))$ includes the previous best position and makes up the cognitive component and the third term $(c_2 \times r_2(0,1) \times (g_{best} - x_i^t))$ takes the best previous position of the neighborhood and constitutes the social component.

Algorithm 10 Flow of the PSO

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

```
 1: $Pop \leftarrow FHPS$
 2: for each $P \in PD$ do
 3:     $g_{best} \leftarrow Fitness(P, P)$
 4:     $PS \leftarrow$ randomseq($Pop$, length($P$))
 5:     $p_{best} \leftarrow Fitness(PS, P)$
 6:     $FS \leftarrow ()$
 7:     if $p_{best} \geq g_{best}$ then
 8:         return $PS$
 9:     end if
10:     while ($p_{best} < g_{best}$) do
11:         for $i$ in range(length($PS$)) do
12:             if $PS[i] = P[i]$ then
13:                 $FS.append[i]$
14:             end if
15:         end for
16:         Calculate updated velocity $UV$ using Equation 4
17:         $NS \leftarrow update\_position(PS, UV, FS)$
18:         $NF \leftarrow Fitness(NS, P)$
19:         if $NF = g_{best}$ then
20:             return NS
21:         end if
22:         if $NF > p_{best}$ then
23:             $PS \leftarrow NS$
24:             $p_{best} \leftarrow NF$
25:         end if
26:     end while
27:     return $PS$
28: end for
```

According to the nature of our problem, the velocity in Equation (2) is adapted to be an integer number. For that, we modified Equation (2) as:

$$UV = \lfloor (w + (c_1 \times r_1(0,1) \times (p_{best} - fit)) + (c_2 \times r_2(0,1) \times (g_{best} - fit))) \rfloor \tag{4}$$

where UV represents the updated velocity, p_{best} and g_{best} represents the fitness values for the current random proof sequence, and the original proof sequence respectively. The fit represents the fitness of sequence in the current iteration. Note that the acceleration constants do not play any role in our case so their values are fixed to 1.

Equation (4) for updating the velocity basically indicates how many positions are required to be changed in a random proof sequence so that it reach the next position. In the position update process, randomly selected position values of a proof sequence are changed from their original values. This means that for a given random proof sequence, a randomly chosen value i is replaced by a random

HPS from the current population *Pop*. As the position update function depends on the velocity factor, the continuously changing nature of velocity proved to be unfavorable in our case as PSF-PSO kept on running for more than two hours and still was unable to completely evolve large random proof sequences.

To solve the aforementioned velocity update issue, we keep track of those positions in the random proof sequences that have matched its value with the target proof sequence by using an array called *fixedSlots(FS)*. While updating the positions of a random proof sequence, we check whether the position in the sequence, which is to be replaced with a random *HPS*, is already present in *FS* or not. If the position is present in FS, then another random number is generated for a different position. If the position is not present, then that particular position is updated with a random HPS from *Pop*. Algorithm 11 presents the procedure for updating the position.

Algorithm 11 Update_Position

Input: PS: A proof sequence, UV: updated velocity, and FS: fixedSlots array
Output: Updated Sequence

```
1: procedure UPDATE_POSITION($PS, UV, FS$)
2:     for $i$ in range(0, $UV$) do
3:         $rp \leftarrow$ randomint(1, length($PS$))
4:         if ($rp \notin FS$) then
5:             $alter \leftarrow$ randomsample($Pop$, 1)   ▷ (1-length proof sequence form Pop)
6:             $PS[rp] \leftarrow alter$                  ▷ ($PS[rp] \neq$ alter)
7:         end if
8:     end for
9:     return $PS$
10: end procedure
```

4.3.1. Illustrated Example

Just like for PSF-GA and PSF-SA, Table 1 is used to explain the working of PSF-PSO. Suppose P = *CASES_TAC, SUBGOAL_THEN, DISCH_TAC, BETA_TAC*, so g_{best} = Fitness(P, P) = 4. Consider that the *PS* = *PROVE_TAC, CONJ_TAC, MP_TAC, X_GEN_TAC*, so p_{best} = Fitness(PS, P) = 0. As $p_{best} = 0$, thus the $fixedSlots$ (FS) array is empty. Assume that the UV using Equation (4) is 1. This means that one position will be changed in PS by applying the *update_position* procedure. Suppose that generated $rp = 3$ and $alter$ = $DISCH\_TAC$, so NS = *PROVE_TAC, CONJ_TAC, DISCH_TAC, X_GEN_AC* and NF = Fitness(NS, P) = 1. As $NF > p_{best}$, thus NS will be assigned to PS and p_{best} becomes 1.

As still $p_{best} < g_{best}$, the PS again goes through the *update_position* process. This time $FS = \{3\}$ as one HPS at location 3 is same in both PS and P. Suppose that the generated number obtained by Equation (4) is 2. This means that two positions in PS will be changed. The *update_position* procedure must generate these two random positions other than 3 because this value is already the same in PS and P. Suppose it generated the first random position as $rp =$ 1 and $alter = BETA\_TAC$. Hence $NS =$ *BETA_TAC, CONJ_TAC, DISCH_TAC, X_GEN_TAC*. Assume that the second time, the *update_position* procedure generates $rp = 2$ and $alter =$ *SUBGOAL_THEN*. Thus, the new $NS =$ *BETA_TAC, SUBGOAL_THEN, DISCH_TAC, X_GEN_TAC*. As $p_{best} < NF$ (Fitness (NS, P)), so p_{best} would be updated to 2 and the NS will be assigned to PS. The above process continues till $p_{best} = g_{best}$.

5. Experimental Evaluation

The proposed evolutionary/heuristic-based algorithms are implemented in Python and the code can be found at [54]. To evaluate the proposed approaches, experiments were carried on a computer equipped with a fifth generation Core $i5$ processor and 4 GB of RAM. For GA, the value of p for UC is set to 0.5. For SA, maximum temperature ($Temp$), minimum Temperature (T_min) and α are set to 100000, 0.00001 and 0.99954001 respectively. For PSA, coefficients w, c_1, and c_2 are set to 1. Some important results obtained by applying the proposed algorithms on *PD* are discussed in this section.

We first investigated the performance of PSF-GA for finding the proofs of theorems/lemmas in 14 HOL4 theories available in its library. These theories are: *Transcendental, Arithmetic, RichList, Number, Sort, Bool, BinaryWords, FiniteMap, InductionType, Combinator, Coder, Encode, Decode* and *Rational*. We selected five to twenty theorems/lemmas from each theory. In total, the *PD* contains 300 proof sequences and 93 distinct *HPS*. 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 Rational theory is the dense theorem that proves that there exists a rational number between any two real numbers.

The PSF-GA was run with the different crossover and mutation operators on the considered theorems/lemmas ten times. The fitness values in Table 3 represents the total *HPS* that is used in the complete proof and these values are same for respective theorems and lemmas in all crossover and mutation operators. The generations column indicates how many times a random proof sequence goes through GA operators to reach the target proof sequence. The time column represents the amount of time (in seconds) used by the GA to find the complete proof for a theorem/lemma. 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*.

Next, we investigate the performance of PSF-SA for finding the proofs of theorems/lemmas in 14 HOL4 theories. The obtained results are listed in Table

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

HOL Theory	**No.**	**HOL4 Theorems**
Transcendental	L1	`⊢ ∀x. 0<=x∧x <= 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])`
	T8	`⊢ PERM = PERM_SINGLE_SWAP`
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 ==> ?r2. rat_res r1 r2 ∧ rat_les r2 r3`

4. The comparison of PSF-SA with PSF-GA for *T2* is shown in the third part of Table 4. For PSF-GA, a different crossover operator has no great effect on the overall performance of PSF-GA. However, using the MPIM operator allowed to find the target proof sequences considerably more quickly than using the SM operator. For *T2*, PSF-SA is found to be faster (39,028 generations) than PSF-GA with different crossover and mutation operators. For this particular example, PSF-SA is approximately sixty seven times faster than PSF-GA with different crossover operators and SM. Whereas, it is approximately fourteen times faster than the PSF-GA with different crossover operators and MPIM.

The results for PSF-PSO to find the proofs of theorems/lemmas in 14 HOL4 theories are listed in Table 4. The comparison of PSF-PSO with PSF-SA and PSF-GA for *T2* is shown in the third part of Table 4. For *T2*, PSF-PSO is found to be faster (17,456 generations) than both PSF-SA and PSF-GA with different crossover and mutation operators. For this particular example, PSF-PSO is approximately two times faster than PSF-SA. There are two possible reasons for this. First, the update velocity function in some generations may allow the particle to change the HPS in large number of positions. Second, the $fixedSlots$ array helps the particle to not make changes in those positions where the HPS in both random particle and original particle match. This avoid the mismatching of HPS at already matched positions in both particles.

The average number of generations for the three algorithms to reach the target proof sequences in the whole dataset are shown in Table 5. PSF-GA with different crossover and *MPIM* operators is approximately fourteen times faster than PSF-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

Table 3: Results for the proposed PSF-GA algorithm

T/L	**C* & M***	**Fit****	**Generations**	**Time**	**C & M**	**Fit**	**Generations**	**Time**
L1	SPC/SM	54	1,768,791	33.43	SPC/MPIM	54	154,043	3.22
T1	SPC/SM	58	2,094,931	36.80	SPC/MPIM	58	204,043	6.73
T2	SPC/SM	81	2,231,664	48.56	SPC/MPIM	81	500,500	14.89
T3	SPC/SM	66	2,527,404	41.35	SPC/MPIM	66	291,162	6.61
T4	SPC/SM	19	593,052	6.72	SPC/MPIM	19	38,307	0.81
T5	SPC/SM	23	675,215	9.05	SPC/MPIM	23	33,655	0.87
T6	SPC/SM	20	338,105	4.25	SPC/MPIM	20	24,776	0.51
T7	SPC/SM	17	264,263	3.36	SPC/MPIM	17	21,136	0.40
T8	SPC/SM	42	971,951	19.49	SPC/MPIM	42	94,172	1.91
T9	SPC/SM	23	654,111	8.45	SPC/MPIM	23	38,309	0.80
T10	SPC/SM	23	695,671	9.69	SPC/MPIM	23	45,552	1.01
L1	MPC/SM	54	1,488,005	27.21	MPC/MPIM	54	205,521	3.89
T1	MPC/SM	58	1,840,467	35.93	MPC/MPIM	58	153,644	5.01
T2	MPC/SM	81	2,713,867	53.84	MPC/MPIM	81	589,292	16.14
T3	MPC/SM	66	2,128,636	37.54	MPC/MPIM	66	259,784	5.26
T4	MPC/SM	19	458,182	5.25	MPC/MPIM	19	24,960	0.48
T5	MPC/SM	23	487,539	5.91	MPC/MPIM	23	32,750	0.83
T6	MPC/SM	20	495,812	5.82	MPC/MPIM	20	28,091	0.63
T7	MPC/SM	17	276,087	3.22	MPC/MPIM	17	19,997	0.43
T8	MPC/SM	42	1,245,801	23.67	MPC/MPIM	42	101,795	2.22
T9	MPC/SM	23	469,625	6.73	MPC/MPIM	23	50,780	1.19
T10	MPC/SM	23	680,625	8.96	MPC/MPIM	23	35,314	0.70
L1	UC/SM	54	1,652,013	31.83	UC/MPIM	54	114,277	2.51
T1	UC/SM	58	1,976,025	36.32	UC/MPIM	58	126,097	4.72
T2	UC/SM	81	2,905,410	56.63	UC/MPIM	81	589,292	17.15
T3	UC/SM	66	2,662,751	44.81	UC/MPIM	66	257,215	5.21
T4	UC/SM	19	706,950	9.12	UC/MPIM	19	20,702	0.41
T5	UC/SM	23	819,903	11.97	UC/MPIM	23	51,614	1.37
T6	UC/SM	20	407,183	4.89	UC/MPIM	20	26,635	0.58
T7	UC/SM	17	321,183	6.16	UC/MPIM	17	20,263	0.48
T8	UC/SM	42	1,786,216	25.53	UC/MPIM	42	115,606	2.15
T9	UC/SM	23	625,908	8.38	UC/MPIM	23	28,030	.51
T10	UC/SM	23	716,950	9.16	UC/MPIM	23	45,925	1.44

* Crossover and mutation ** Fitness

as compared to *SM*. Whereas, PSF-SA is approximately six times faster than PSF-GA with *MPIM* and different crossover operators. Similarly, PSF-PSO is approximately 1.7 times faster than PSF-SA. The main reasons for this is that in PSF-SA and PSF-PSO, only one procedure ($Get\_Neighbor$ in PSF-SA and $Update\_Position$ in PSF-PSO) is called. On the other hand, in PSF-GA, two procedures (crossover and mutation) are called.

Population diversity greatly influences the ability of GAs to make progress as it iterates from one generation to another [51]. The proof searching process with PSF-GA can be trapped in a local optimum due to the loss of diversity through premature convergence of the *HPS* in the population. This makes diversity maintenance as one of the fundamental issues for GAs in general. Next, we studied population diversity in PSF-GA with two measures. The first one being the standard deviation of fitness SD_f, which is measured as:

$$SD_f = \sqrt{\frac{\sum_{i=1}^{N}(f_i - \bar{f})^2}{N-1}} \tag{5}$$

Table 4: Results for PSF-SA and PSF-PSO and comparison with PSF-GA

T/L	Technique	Fitness	Generations	Time (s)
L1	PSF-SA	54	20,192	0.490
T1		58	20,476	0.497
T2		**81**	**39,028**	**0.986**
T3		66	30,560	0.628
T4		19	7,301	0.113
T5		23	9,892	0.142
T6		20	6,675	0.0539
T7		17	5,278	0.082
T8		42	15,020	0.261
T9		23	9,584	0.192
T10		23	9,639	0.162
L1	PSF-PSO	54	11,034	0.716
T1		58	12,842	0.785
T2		**81**	**17,456**	**1.85**
T3		66	15,651	1.391
T4		19	6,302	0.0815
T5		23	8,285	0.193
T6		20	5,884	0.098
T7		17	3,320	0.052
T8		42	10,019	0.245
T9		23	6,317	0.0848
T10		23	6,317	0.0841
T2	PSF-GA(SPC/SM)	81	2,231,664	48.56
T2	PSF-GA(MPC/SM)	81	2,713,867	53.84
T2	PSF-GA(UC/SM)	81	2,905,410	56.63
T2	PSF-GA(SPC/MPIM)	81	500,500	14.89
T2	PSF-GA(MPC/MPIM	81	524,272	16.14
T2	PSF-GA(UC/MPIM)	81	589,292	17.15

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

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 12.02 with a mean of 14.03 for PSF-GA. The second measure used to examine the variability of *HPS* in the *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:

$$SD_t = \sqrt{\frac{\sum_{i=1}^{N}(t_i - \bar{t})^2}{N-1}} \tag{6}$$

Table 5: Average total generation count for PSF-PSO, PSF-SA, and PSF-GA

Algorithm	**Ave. Generation Count**	**Total Time**	**Memory**
PSF-PSO	783,502	20.76 s	3495 Mb
PSF-SA	1,357,268	20.54 s	3507 Mb
PSF-GA(SPC/SM)	119,062,973	1463.48 s	3395 Mb
PSF-GA(MPC/SM)	122,378,292	1583.30 s	3463 Mb
PSF-GA(UC/SM)	124,115,903	1616.69 s	3507 Mb
PSF-GA(SPC/MPIM)	8,833,888	174.25 s	3450 Mb
PSF-GA(MPC/MPIM)	9,141,943	188.34 s	3427 Mb
PSF-GA(UC/MPIM)	8,704,233	170.491 s	3382 Mb

where t_i is the time taken by PSF-GA to find the correct *ith* proof sequence and $\bar{t}$ is the *mean* of the time values. The calculated SD_t for all the proof sequences in the *PD* is listed in Table 6 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*. As PSF-GA with *SM* was slower (fourteen times) than PSF-GA with *MPIM*, there is more time points for *SM* than *MPIM*, which makes the SD_t and the respective *mean* higher for *SM*. Moreover, the calculated SD_t and mean for PSF-SA and PSF-PSO are also listed in Table 6. Similar to PSF-GA, the SD_f for PSF-SA and PSO-PSO is 12.02 with a mean of 14.03. The amount of memory used by the three algorithms while searching for proofs is also listed in Table 6. It can be seen that three algorithms require approximately the same memory while searching for proof sequences in the *PD*.

Table 6: SD_t, mean and total time for three algorithms

Algorithm	**Mean**	**SDt**	**Time**	**Memory**
PSF-GA(SPC/SM)	4.876	7.52	1463.48 s	3395 Mb
PSF-GA(MPC/SM)	4.580	7.04	1583.30 s	3463 Mb
PSF-GA(UC/SM)	4.667	7.09	1616.69 s	3507 Mb
PSF-GA(SPC/MPIM)	0.538	1.14	174.25 s	3450 Mb
PSF-GA(MPC/MPIM)	0.585	1.12	188.34 s	3427 Mb
PSF-GA(UC/MPIM)	0.583	1.19	170.491 s	3382 Mb
PSF-SA	0.062	0.101	20.54 s	3507 Mb
PSF-PSO	0.074	0.262	20.76 s	3495 Mb

Statistical tests are usually performed to investigate whether one algorithm offers a significant improvement (or not) over another for a given problem. The studies [1, 12, 17] suggested to use non-parametric statistical tests for comparing evolutionary/heuristic algorithms. Thus, the Friedman test [16] is used on the means for the times and generations taken by the three proposed algorithms to find the correct proofs for target proof sequences in the *PD*. The following hypotheses are tested:

H0T*: The means for the times taken by the proof searching approaches for each proof sequence in* PD *are same.*

H1T*: The mean number of times for at least one algorithm is different from the others.*

H0G*: The means for the generations taken by the proof searching approaches for each proof sequence in* PD *are same.*

H1G*: The mean number of generations for at least one algorithm is different from the others.*

The results of the Friedman test indicate that both $H1T$ and $H1G$ should be accepted (p < 0.05) with results of 1876.38 and 1577.21, respectively. The Wilcoxon test [65] results for the mean times and number of generations for the three algorithms are listed in Table 7 and Table 8, respectively. The results in both tables show that PSF-PSO is significantly better than PSF-SA and PSF-SA is significantly better than different versions of PSF-GAs. For time (Table 7), it can bee seen that PSF-GA with SM and different crossover operators (PSF-GA(SPC/SM, MPC/SM and UC/SM)) are not significantly better than each other. The same is the case with GA with MPIM and different crossover operators (PSF-GA(SPC/MPIM, MPC/MPIM and UC/MPIM)). However for generations (Table 8), the opposite is true for PSF-GA with same mutation operator and different crossover operators. Combining these statistical results with previous results suggests that PSF-PSO is significantly better than PSF-SA and PSF-SA is significantly better than GAs.

Table 7: Wilcoxon p-value matrix for times

	SPC/SM	**MPC/SM**	**UC/SM**	**SPC/MPIM**	**MPC/MPIM**	**UC/MPIM**	**PSF-SA**	**PSF-PSO**
SPC/SM	—	9.83E-01	4.54E-01	6.08E-51	6.08E-51	6.08E-51	6.08E-51	6.08E-51
MPC/SM	9.83E-01	—	2.75E-01	6.08E-51	6.08E-51	6.08E-51	6.08E-51	6.08E-51
UC/SM	4.54E-01	2.75E-01	—	6.08E-51	6.08E-51	6.08E-51	6.08E-51	6.08E-51
SPC/MPIM	6.08E-51	6.08E-51	6.08E-51	—	5.23E-02	5.60E-01	6.46E-51	6.08E-51
MPC/MPIM	6.08E-51	6.08E-51	6.08E-51	5.23E-02	—	6.40E-02	6.46E-51	6.14E-51
UC/MPIM	6.08E-51	6.08E-51	6.08E-51	5.60E-01	6.40E-02	—	6.14E-51	6.08E-51
PSF-SA	6.08E-51	6.08E-51	6.08E-51	6.46E-51	6.46E-51	6.14E-51	—	5.07E-10
PSF-PSO	6.08E-51	6.08E-51	6.08E-51	6.08E-51	6.14E-51	6.08E-51	5.07E-10	—

Table 8: Wilcoxon p-value matrix for generations

	SPC/SM	**MPC/SM**	**UC/SM**	**SPC/MPIM**	**MPC/MPIM**	**UC/MPIM**	**PSF-SA**	**PSF-PSO**
SPC/SM	—	2.03E-11	4.63E-42	6.08E-51	6.08E-51	6.08E-51	6.08E-51	6.08E-51
MPC/SM	2.03E-11	—	2.80E-042	6.08E-51	6.08E-51	6.08E-51	6.08E-51	6.08E-51
UC/SM	4.63E-42	2.80E-042	—	6.08E-51	6.08E-51	6.08E-51	6.08E-51	6.08E-51
SPC/MPIM	6.08E-51	6.08E-51	6.08E-51	—	1.17E-03	4.09E-02	6.46E-51	6.08E-51
MPC/MPIM	6.08E-51	6.08E-51	6.08E-51	1.17E-03	—	8.89E-05	6.08E-51	6.08E-51
UC/MPIM	6.08E-51	6.08E-51	6.08E-51	4.09E-02	8.89E-05	—	6.08E-51	6.08E-51
PSF-SA	6.08E-51	6.08E-51	6.08E-51	6.08E-51	6.08E-51	6.08E-51	—	8.81E-51
PSF-PSO	6.08E-51	6.08E-51	6.08E-51	6.08E-51	6.08E-51	6.08E-51	8.81E-51	—

In GAs, the idea of crossover is simple: selecting two chromosomes (solutions) as parents and combining some parts from parents to generate better child chromosomes. This intuition has been formalized in [26] with the concept of building blocks used in schema theory. The mechanics of crossover provides a way to implement this idea. Thus, all types of crossover share the same idea but the mechanics to implement the idea may vary considerably. For example, SPC uses a single crossing point and MPC employs two crossing points. Investigating the usefulness of crossover operators in GAs is an important research topic. The standard procedure (or test) to determine the usefulness of crossover operators is to compare a GA that implement crossover with a GA without crossover. If better performance is obtained with GA with crossover, then it proves the usefulness of crossover. Jones [29] argued that this conclusion is not justifiable and made a distinction between the crossover idea and the crossover mechanics. It was shown through various experiments in [29] that crossover mechanics alone can be used effectively for search and evolution even in the absence of the crossover idea. For this, a testing method (called headless chicken test (HCT)) was proposed to examine the usefulness of crossover for a particular problem instance. Next, we investigate the usefulness of three crossover operators (SPC, MPC, and UC) in PSF-GA using the HCT. In the HCT, the PSF-GA with normal crossovers are compared with the identical PSF-GA that uses the random version of crossovers as shown in Figure 2.

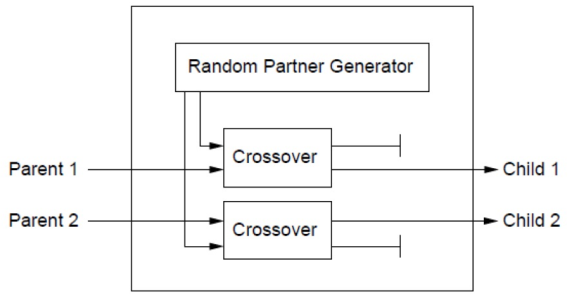

Figure 2: HCT with random crossover

For the two parents, the random crossover operator generates two random individuals and uses them in crossover with the parents. Thus, instead of recombining two parents as a normal crossover would do, two random individuals are created and crossed with parents in a random crossover. The normal crossover (that can be either SPC or MPC) procedure is the one used in the GA that is being compared to the GA with random crossover. GA with random crossover has no direct communication between parents as the random crossover does not work directly with parents. As the individuals involved in the crossover process is chosen randomly, the operation is purely mechanical and has nothing of the spirit of crossover. Despite the identical mechanical re-arrangement, the operation is not a crossover in reality. Another argument regarding the fact that this operation is not really a crossover is that two parents are not required. For example, for MPC, one can simply select the crossing points and set the loci between the points to randomly chosen alleles. This means that it is clearly a macromutation.

Using this test, one can distinguish the gains the GA is making through the idea of crossover from those made simply through the mechanics. If GA is not making any additional progress due to the idea of crossover, one might do as well simply by using macromutations. The poor performance of original GA with normal crossover compared to the GA with random crossover indicates the absence of well-defined building blocks. Table 9 shows the results for PSF-GA with original crossovers and its randomized versions. The + sign with the name of crossover operators represents the randomized version of that crossover. For generations, it can be seen that the original PSF-GA with normal crossover operators performs as PSF-GA with randomized crossover operators without any significant differences. This shows the availability of well-defined building blocks for the crossover operators in PSF-GA.

Table 9: HCT results for the PSF-GA

PSF-GA	**Ave. Generation Count**	**Time (Sec)**
SPC/SM	119,062,973	1463.48 s
MPC/SM	122,378,292	1583.30 s
UC/SM	124,115,903	1616.69 s
SPC/MPIM	8,833,888	174.25 s
MPC/MPIM	9,141,943	188.34 s
UC/MPIM	8,704,233	170.49 s
SPC+/SM	116,739,893	2122.57 s
MPC+/SM	121,801,416	2002.83 s
UC+/SM	122,226,585	2114.20 s
SPC+/MPIM	8,405,133	214.79 s
MPC+/MPIM	8,349,399	201.81 s
UC+/MPIM	8,966,029	242.44 s

SA can select the new solution obtained with the *GN* procedure that is not better than the present solution with the *acceptance probability.* The reason for this is that there is always a possibility that the new solution could lead the SA to the global optimum. In PSF-SA, we chose the *acceptance probability* (AR) using Equation (1), which is then compared with a random number generated within the range (2.71825400004040, 2.71825464604849).

The above 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 range (2.71825400004040, 2.71825464604849), then the new solution (that is not better than present) is selected. We used a counter named acceptance rate counter (ARC) that keeps track of how many times the new solution is picked. By simulation, it was found that this factor does not play huge 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. PSF-SA finds only one global solution for each random proof sequence based on the fitness value. The average generation count of PSF-SA for all proof sequences in the *PD* with and without the acceptance rate is listed in Table 10. The obtained results are the same with negligible difference.

Table 10: Performance of PSF-SA with and without AR

Results without AR		**Results with AR ($ARC = 123675$)**	
Avg. Gen. Count	Time (S)	Avg. Gen. Count	Time (S)
1,357,268	20.54 s	1,480,943	21.92 s

Next, we checked how much time the algorithms take on average to find the *HPS* in the random proof sequence that matches with the *HPS* in the target sequence (PSF-GA time is shown in Figure 3 and PSF-SA, PSF-PSO are shown in Figure 4). The runtime difference when applying the three algorithms to find the correct *HPS* in a proof sequence is negligible. It is observed that PSF-PSO and PSF-SA were able to quickly find the matched *HPS* as compared to the PSF-GA with different crossover and mutation operators. For three algorithms, the time to find the *HPS* increases for each following *HPS*. On average, the time taken by the PSF-SA and PSF-PSO to find *HPS* was almost same. .

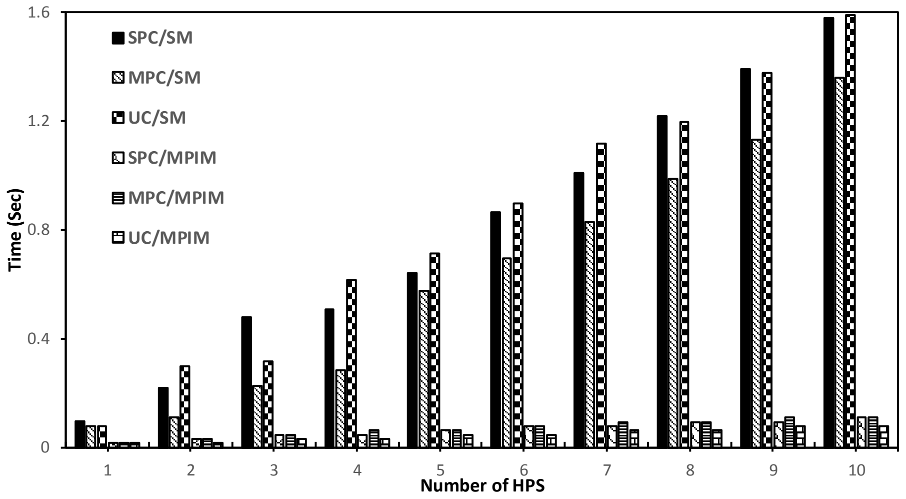

Figure 3: Time used by the PSF-GA to find the first ten matched *HPS*

The longest proof in the *PD* is for theorem $T2$ (positive value of sine) and it consists of 81 *HPS*. Here we call this theorem $PVoS$. The runtime of the PSF-GA to find all matched 81 *HPS* in $PVoS$ with different crossover and mutation operators is shown in Figure 5 and the runtime of PSF-SA and PSF-PSO is shown in Figure 6. Those generations are shown on the x-axis where the three algorithms were able to find the *HPS* in a random proof sequence that matches with the *HPS* in $PVoS$. Generations where *HPS* does not match are excluded. We observed that in most of the generations, the algorithms were unable to find

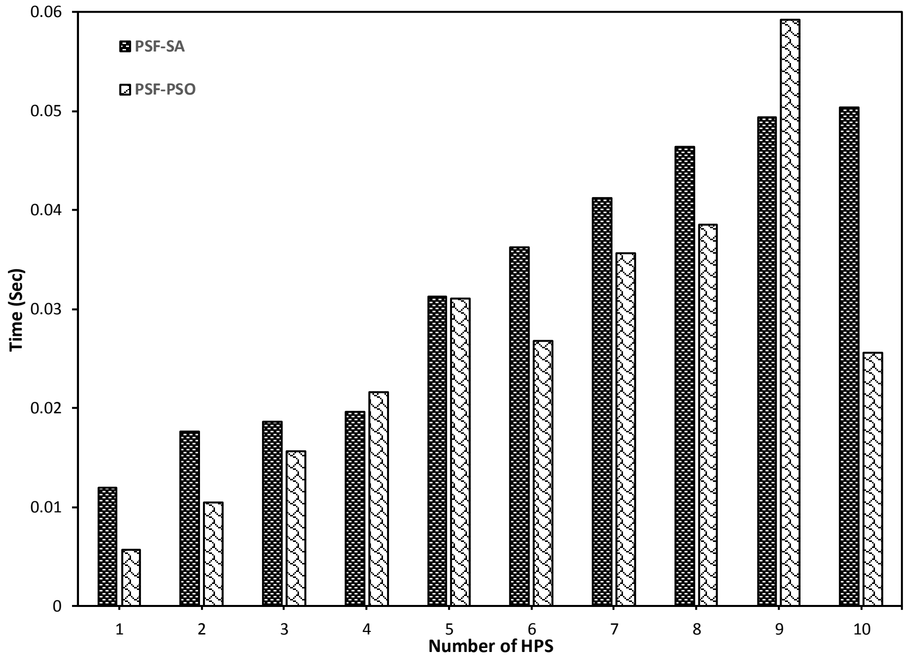

Figure 4: Time used by the PSF-SA and PSF-PSO to find the first ten matched *HPS*

the same *HPS* in a random proof sequence and *PVoS*. For the three algorithms, it was observed that the time to find the *HPS* increases with increase in the number of *HPS*. Moreover, with increase in generations, the performance of algorithms tend to decrease for fitness. This means that with more generations, the algorithms become slow in finding the correct *HPS* for long proof sequence as compared to earlier generations.

In each generation, the probability for the three algorithms to find the complete correct proof for *PVoS* is listed in Table 11. PSF-PSO and PSF-SA have high probability compared to PSF-GA. Overall, the performance of the three algorithms is far better than proof searching with a pure random search. For example, the probability (which is very low as compared to PSF-GA, PSF-SA and PSF-PSO) for a pure random search to find a valid proof is also listed in Table 11. Even for the theorems with smaller (fitness of 10) proof sequences, the probability is still in the magnitude of 10^{-16}.

The brute force approach (BFA) tries to produce a possible candidate for the solution (original proof sequence in this work) by enumerating all the possible candidates. In the Introduction, we argued that using BFA for proof searching is infeasible. To support this argument, we also implemented a BFA in Python. The BFA takes a lot of time when run on the proof sequences from the *PD*, even for proof sequences with smaller fitness values. For example, Table 12 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

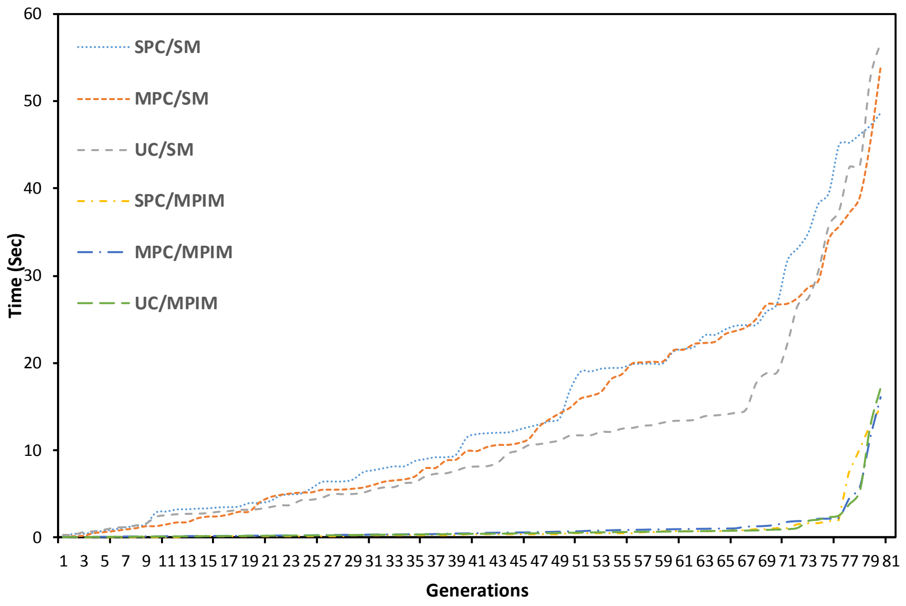

Figure 5: Time and generations for the $PVoS$ theorem with PSF-GA

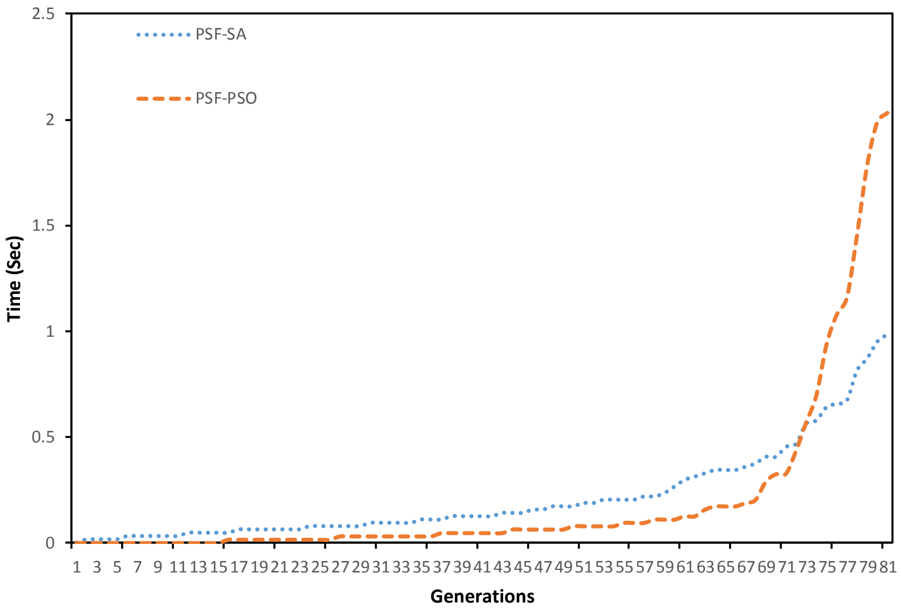

Figure 6: Time and generations for the $PVoS$ theorem with PSF-SA and PSF-PSO

with four different HPS, BFA took 19.76 seconds and 5,444,048 attempts, that

Table 11: Comparison of PSF-PSO, PSO-SA, PSO-GA and Pure Random Search (PRS)

Algorithm	**Probability**
PSF-PSO	4.64×10^{-3}
PSF-SA	2.07×10^{-3}
PSF-GA(CO/SM)	3.09×10^{-5}
PSF-GA(CO/MPIM)	1.50×10^{-4}
PRS	2.69×10^{-156}

is approximately 275,508 attempts per second. For a theorem with ten different *HPS*, the program was unable to find the target proof sequence even after running for more than 3 hours. Note that for a theorem with a fitness value of 10, there are 93^{10} total possible candidate proof sequences.

Table 12: Results for BFA

Proof Sequence	**Fit**	**Time**	**Attempts**
GEN_TAC REWRITE_TAC	2	0.0009 s	160
SRW_TAC METIS_TAC STRIP_TAC	3	1.138 s	348,925
BETA_TAC SUBST_TAC CONJ_TAC Q.SPEC_TAC	4	19.796 s	5,444,048
RW_TAC SUBGOAL_THAN CASES_ON CASES_ON MATCH_MP SRW_TAC	5	1972.07 s	574,262,133

Lastly, we compared the three algorithms in terms of convergence speed to investigate how fast the algorithms were able to converge towards the optimal solution. For the first 20,000 generations, the convergence speed of the three algorithms for *PVoS* is shown in Figure 7. PSF-PSO converges very fast and found the correct *HPS* within 17,500 generations. Similarly, PSF-SA is at second position by finding 76 correct *HPS* in 20,000 generations. Whereas PSF-GA(MPC/MPIM) is fast at the beginning reaching the speed of the other two algorithms but gets slow after 3,000 iterations till the end. On the other hand, PSF-GA(MPC/SM) convergence speed is linear and slow from the start. At 20,000 generations, PSF-GA(MPC/MPIM) find approximately 66 correct *HPS*, whereas PSF-GA(MPC/SM) find approximately 5 correct *HPS*.

Overall, it was observed through various experiments that the proposed three algorithms are able to quickly optimize and automatically find the correct proofs for theorems/lemmas in different HOL4 theories. PSF-PSO was found to be much faster than PSF-SA and PSF-GA. Whereas, PSF-SA performed better than PSF-GA. Besides HOL4, the proposed evolutionary/heuristic-based approaches can also be used for proof searching and proof optimization in other proof assistants, such as Coq [6] and PVS [52].

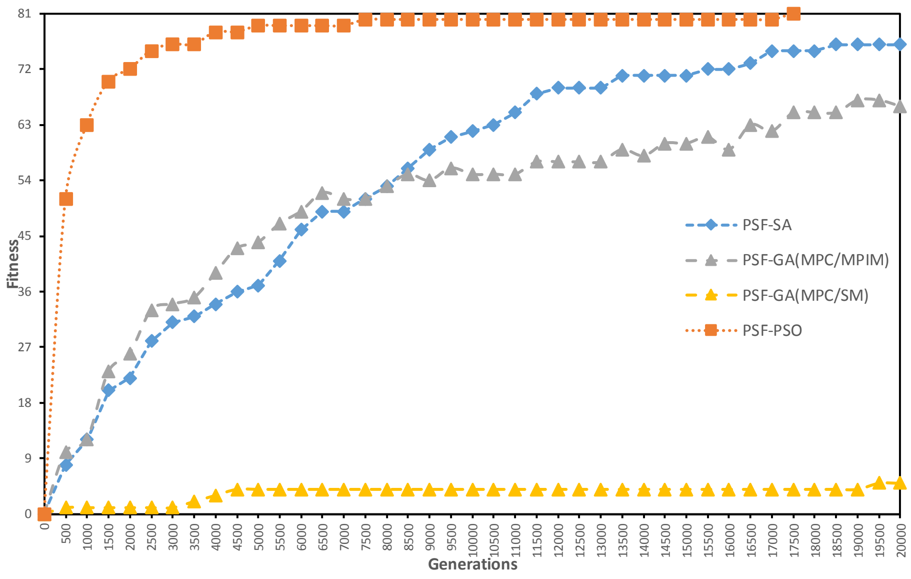

Figure 7: Convergence performance

6. Related Work

Our work on using evolutionary/heuristic algorithms in proof assistant is not the first one.

For example, a GA was used in [28, 67] 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 few proof steps. 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. A recent work [45] briefly discussed how evolutionary computation can be used to improve the heuristics of automatic proof search in Isabelle/HOL. The objective is to find heuristics that can select the most promising PSL [46] (which is a proof strategy language for Isabelle/HOL) strategy from various available hand written strategies when applied to a given proof goal. A framework based on GA is provided in [57] to find good search heuristics for the E ATP.

A genetic programming [38] and a pairwise combination (that focuses only on crossover based approach) were used in [13] on patterns (simple tactics) discovered in Isabelle proofs to evolve them into compound tactics. However, Isabelle's proofs were represented using a linearized tree structure where the proofs were divided into separate sequences and weights were assigned to them. However, 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 proposed framework for proof searching can handle proof goals of various length. Moreover, 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 proposed proof searching approaches is generated. In last, no human guidance is required in the proposed approaches during the evolution process for random proof sequences.

On the other hand, machine learning was first used in [60] for reasoning in large theories. For ITPs, [31] used machine learning for the task of premise selection in the Coq system. For HOL Light, a machine learning benchmark was developed in [30] for higher-order logic reasoning. GamePad [27] and CoqGyM [66] provide machine learning environments and benchmarks for the Coq proof assistant. Deep learning was first used in [3], where convolutional neural networks were used for premise selection in large theories from the Mizar Mathematical library. Experiments were done in the ATP E. Similarly, [42] used deep neural networks for internal guidance in E. The applicability of reinforcement learning was demonstrated in [33] on hand engineered features in first order logic using the ATP rlCoP. A deep graph embedding-based learning approach was provided in [61] for premise selection. GRU networks were used in [62] for theorem proving in MetaMath. Some external provers were also developed for the HOL4 theorem prover in [18, 19]. GNNs (graph neural networks) were used in [53] for higher-order proof searching. RNNs and LSTMs were used in [69] for the prediction of tactics in one Coq theory.

Proverbot9001 [55] and HOList [5] provide deep learning environments and benchmarks for the Coq and HOL Light proof assistant respectively. A large dataset for mechanised proofs was developed in [41] and various neural sequence-to-sequence models were used for proposition generation. Neural networks were also used recently in [59] on several datasets for creating conjectures. Similar to related datasets in [5, 27, 30, 31, 41, 55, 59, 66], we also build a dataset for theorems/lemmas in higher-order logic that only consists of tactics and proof procedures.

We used the SPM-based proof process learning approach [48] to find the frequent *HPS* that are then used by the proposed framework for the creation of an initial population for proof searching approaches. The approach [48] is not limited to PVS only, but can be used in other proof assistants as shown in this paper and in [49]. The GA-based proof process learning approach [49] lacks the ability to learn the previously proved facts and to predict the proofs for new unproved theorems/lemmas (conjectures). For this, a deep learning technique (long short-term memory (LSTM)) was used in [47] on various HOL4 theories for learning and predicting proof sequences.

7. Conclusion

Despite the evolution in computing systems, ITPs still depend on user interaction to manually guide the proofs assistants in finding the proof for a particular goal. This interaction makes the proof development process quite cumbersome and time consuming activity for users, in particular for long and complex proofs. We introduced three proof searching approaches in this paper for the possible linkage between evolutionary/heuristic algorithms (such as GA, SA and PSO) with theorem provers (such as HOL4) to make the proof finding and development process easier. The three proof searching approaches were used to optimize and find the correct proofs in different HOL4 theories. Moreover, a performance comparison of the three algorithms showed that both SA and PSO outperformed GA.

The main limitation of the proposed approaches is that they lack the ability to learn the previously proved facts and can be used on proof sequences of formalized theorems/lemmas in HOL4 theories. Nevertheless, obtained results suggest that the research direction of linking and integrating evolutionary/heuristic algorithms with proof assistants is worth pursuing. This kind of framework may have a considerable impact to advance and accumulate human knowledge, especially in the field of automated reasoning. There are several directions for future work, some of which are:

- Making the proof searching process more general in nature to evolve frequent HOL4 proof steps to compound proof tactics for guiding the proofs of new conjectures.

- Implementing the PSO with headless chicken marcomutation [22] for proof searching and compare the results with the proposed PSO approach. Moreover, relatively new optimization algorithms such as Bat Algorithm [68], Artificial Bee Colony [35] and Grey Wolf Optimizer [43] could also be considered for proof searching.

- Taking advantage of the Curry-Howard isomorphism for sequent calculus [56] that provides a direct relation between programming and proofs, where finding proofs can be viewed as writing programs. With such correspondence, a GA, SA or PSO-based approach can be used to write programs (proofs) and HOL4 proof assistant for simplification and verification by computationally evaluating the programs.

- Using deep learning techniques, such as recurrent neural networks, for the predictions of proofs for new theorems (conjunctures). In this regard, neural networks were recently used in [47, 69] to predict and suggest the correct tactics in HOL4 and Coq proof assistants, respectively.

Acknowledgments

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).

References

[1] B. H. Abed-alguni and D. J. Paul. Hybridizing the cuckoo search algorithm with different mutation operators for numerical optimization problems. *Journal of Intelligent Systems*, 29(1):1043–1062, 2020.

[2] K. Aksoy, S. Tahar, and Y. Zeren. Introduction to HOL4 theorem prover. *Sigma Journal of Engineering and Natural Sciences*, 10(2):237–243, 2019.

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

[4] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell, J. Tuong, G. Keller, T. C. Murray, G. Klein, and G. Heiser. Cogent: Verifying high-assurance file system implementations. In *Proceedings ASPLOS 2016*, pages 175–188. ACM, 2016.

[5] K. Bansal, S. M. Loos, M. N. Rabe, C. Szegedy, and S. Wilcox. HOList: An environment for machine learning of higher order logic theorem proving. In *Proceedings of ICML 2019*, volume 97 of *PMLR*, pages 454–463, 2019.

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

[7] D. Bertsimas and J. Tsitsiklis. Simulated annealing. *Statistal Science*, 8(1):10–15, 1993.

[8] J. C. Blanchette, M. P. L. Haslbeck, D. Matichuk, and T. Nipkow. Mining the archive of formal proofs. In *Proceedings of CICM 2015*, volume 9150 of *LNCS*, pages 3–17. Springer, 2015.

[9] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich. Using crash hoare logic for certifying the FSCQ file system. In *Proceddings of USENIX ATC 2016.* USENIX Association, 2016.

[10] L. M. de Moura, S. Kong, J. Avigad, F. van Doorn, and J. von Raumer. The Lean theorem prover (system description). In *Proceedings of CADE 2015*, volume 9195 of *LNCS*, pages 378–388. Springer, 2015.

[11] D. Delahaye, S. Chaimatanan, and M. Mongeau. *Simulated annealing: From basics to applications*, volume 272, pages 1–35. Springer, 2019.

[12] J. Derrac, S. García, D. Molina, and F. Herrera. A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms. *Swarm and Evolutionary Computation*, 1(1):3–18, 2011.

[13] H. Duncan. *The use of data-mining for the automatic formation of tactics.* PhD thesis, University of Edinburgh, UK, 2007.

[14] M. Färber and C. E. Brown. Internal guidance for Satallax. In *Proceedings of IJCAR 2016*, volume 9706 of *LNCS*, pages 349–361. Springer, 2016.

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

[16] M. Friedman. A comparison of alternative tests of significance for the problem of m rankings. *Annals of Mathematical Statistics*, 11(1):86–92, 1940.

[17] S. García, D. Molina, M. Lozano, and F. Herrera. A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: A case study on the CEC'2005 special session on real parameter optimization. *Journal of Heuristics*, 15(6):617–644, 2009.

[18] T. Gauthier and C. Kaliszyk. Premise selection and external provers for HOL4. In *Proceedings CPP 2015*, pages 49–57. ACM, 2015.

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

[20] H. Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009.

[21] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. L. Roux, A. Mahboubi, R. O'Connor, S. O. Biha, I. Paşca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the Odd Order theorem. In *Proceedings of ITP 2013*, volume 7998 of *LNCS*, pages 163–179. Springer, 2013.

[22] J. Grobler and A. P. Engelbrecht. Headless chicken particle swarm optimization algorithms. In *Proceedings of ICSI 2016*, volume 9712 of *LNCS*, pages 350–357. Springer, 2016.

[23] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo. Certikos: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of OSDI*, pages 653–669. USENIX Association, 2016.

[24] T. C. Hales, M. Adams, G. Bauer, D. T. Dang, J. Harrison, T. L. Hoang, C. Kaliszyk, V. Magron, S. McLaughlin, T. T. Nguyen, T. Q. Nguyen, T. Nipkow, S. Obua, J. Pleso, J. M. Rute, A. Solovyev, A. H. T. Ta, T. N. Tran, D. T. Trieu, J. Urban, K. K. Vu, and R. Zumkeller. A formal proof of the Kepler conjecture. *Forum Mathematics, Pi*, 5(e2):1–29, 2017.

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

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

[27] D. Huang, P. Dhariwal, D. Song, and I. Sutskever. GamePad: A learning environment for theorem proving. In *Proceedings of ICLR 2020*, 2019.

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

[29] T. Jones. Crossover, macromutationand, and population-based search. In *Proceedings of ICGA 1995*, pages 73–80. Morgan Kaufmann, 1995.

[30] C. Kaliszyk, F. Chollet, and C. Szegedy. HolStep: A machine learning dataset for higher-order logic theorem proving. *CoRR*, abs/1703.00426, 2017.

[31] C. Kaliszyk, L. Mamane, and J. Urban. Machine learning of Coq proof guidance: First experiments. In *Proceedings of SCSS 2014*, volume 30 of *EPiC Series in Computing*, pages 27–34, 2014.

[32] C. Kaliszyk and J. Urban. Learning-assisted theorem proving with millions of lemmas. *Journal of Symbolic Computations*, 69:109–128, 2015.

[33] C. Kaliszyk, J. Urban, H. Michalewski, and M. Olsák. Reinforcement learning of theorem proving. In *Proceedings of NIPS 2018*, pages 8836–8847, 2018.

[34] S. Kanav, P. Lammich, and A. Popescu. A conference management system with verified document confidentiality. In *Proceddings of CAV 2014*, volume 8559 of *LNCS*, pages 167–183. Springer, 2014.

[35] D. Karaboga. An idea based on honey bee swarm for numerical optimization. Technical Report TR06, Erciyes University, 2005.

[36] J. Kennedy and R. Eberhart. Particle sawrm optimization. In *Proceedings of ICNN 1995*, pages 1942–1948. IEEE, 1995.

[37] G. Klein, J. Andronick, K. Elphinstone, T. C. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Transactions on Computer Systems*, 32(1):2:1–2:70.

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

[39] R. Kumar, M. O. Myreen, M. Norrish, and S. Owens. CakeML: A verified implementation of ML. In *Proceddings of POPL'14*, pages 179–192. ACM, 2014.

[40] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[41] W. Li, L. Yu, Y. Wu, and L. C. Paulson. Modelling high-level mathematical reasoning in mechanised declarative proofs. *CoRR*, abs/2006.09265, 2020.

[42] S. M. Loos, G. Irving, C. Szegedy, and C. Kaliszyk. Deep network guided proof search. In *Proceedings of LPAR 2017*, volume 46 of *EPiC Series in Computing*, pages 85–105, 2017.

[43] S. Mirjalili, S. M. Mirjalili, and A. Lewis. Grey wolf optimizer. *Advances in Engineering Software*, 69:46–61, 2014.

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

[45] Y. Nagashima. Towards evolutionary theorem proving for Isabelle/HOL. In *Proceedings of GECCO (Poster) 2019*, pages 419–420. ACM, 2019.

[46] Y. Nagashima and R. Kumar. A proof strategy language and proof script generation for Isabelle/HOL. In *Proceedings of CADE 2019*, volume 10395 of *LNCS*, pages 528–545. Springer, 2017.

[47] M. S. Nawaz, M. Z. Nawaz, O. Hasan, P. Fournier-Viger, and M. Sun. Proof searching and prediction in HOL4 with evolutionary/heuristic and deep learning techniques. *Applied Intelligence, https://doi.org/10.1007/s10489-020-01837-7*, 2020.

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

[49] M. Z. Nawaz, O. Hasan, M. S. Nawaz, P. Fournier-Viger, and M. Sun. Proof searching in HOL4 with genetic algorithm. In *Proceedings of SAC 2020*, pages 513–520. ACM, 2020.

[50] T. Nipkow, L. C. Paulson, and M. Wenzel. *Isabelle/HOL: A proof assistant for higher-Order logic.* Springer, 2002.

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

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

[53] A. Paliwal, S. M. Loos, M. N. Rabe, K. Bansal, and C. Szegedy. Graph representations for higher-order logic and theorem proving. In *Proceedings of AAAI 2020*, pages 2967–2974, 2020.

[54] Python codes and HOL4 data. Available at:. `github.com/MuhammadzohaibNawaz/E.H-PSF-GA-SA-PSO`.

[55] A. Sanchez-Stern, Y. Alhessi, L. K. Saul, and S. Lerner. Generating correctness proofs with neural networks. In *Proceedings of MAPL@PLDI 2020*, pages 1–10. ACM, 2020.

[56] J. E. Santo. Curry-howard for sequent calculus at last! In *Proceedings of TLCA 2015*, volume 38 of *LIPIcs*, pages 165–179, 2015.

[57] S. Schäfer and S. Schulz. Breeding theorem proving heuristics with genetic algorithms. In *Proceedings of GCAI 2015*, volume 36 of *EPiC Series in Computing*, pages 263–274. EasyChair, 2015.

[58] K. Slind and M. Norrish. A brief overview of HOL4. In *Proceedings of TPHOL 2008*, volume 5170 of *LNCS*, pages 28–32. Springer, 2008.

[59] J. Urban and J. Jakubuv. First neural conjecturing datasets and experiments. In *Proceedings of CICM 2020*, volume 12236 of *LNCS*, pages 315–323. Springer, 2020.

[60] J. Urban, G. Sutcliffe, P. Pudlák, and J. Vyskocil. Malarea SG1- machine learner for automated reasoning with semantic guidance. In *Proceedings of IJCAR*, volume 5195 of *LNCS*, pages 441–456. Springer, 2008.

[61] M. Wang, Y. Tang, J. Wang, and J. Deng. Premise selection for theorem proving by deep graph embedding. In *Proceedings of NIPS 2017*, pages 2786–2796, 2017.

[62] D. Whalen. Holophrasm: A neural automated theorem prover for higher-order logic. *CoRR*, abs/1608.02644, 2016.

[63] F. Wiedijk. Formalizing 100 theorems, available at. `http://www.cs.ru.nl/~freek/100`, Accessed on 18 August 2020.

[64] J. R. Wilcox, D. Woos, P. Panchekha, Z. Tatlock, X. Wang, M. D. Ernst, and T. E. Anderson. Verdi: A framework for implementing and formally verifying distributed systems. In *Proceedings of PLDI 2015*, pages 357–368. ACM, 2015.

[65] F. Wilcoxon. Individual comparisons by ranking methods. *Biometrics Bulletin*, 1(6):80–83, 1945.

[66] K. Yang and J. Deng. Learning to prove theorems via interacting with proof assistants. In *Proceedings of ICML 2019*, volume 97 of *PMLR*, pages 6984–6994. PMLR, 2019.

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

[68] X. Yang. A new metaheuristic bat-inspired algorithm. In *Nature Inspired Cooperative Strategies for Optimization (NICSO)*, pages 65–74, 2010.

[69] X. Zhang, Y. Li, W. Hong, and M. Sun. Using recurrent neural network to predict tactics for proving component connector properties in Coq. In *Proceedings of TASE 2019*, pages 107–112. IEEE, 2019.