Statistical Proof-Patterns in Coq/SSReflect*

Jónathan Heras and Ekaterina Komendantskaya

School of Computing, University of Dundee, UK
{jonathanheras,katya}@computing.dundee.ac.uk

Abstract. ML4PG is an extension to the Proof General interface, allowing the Proof General user to invoke machine-learning algorithms and cluster proofs and proof libraries. In this paper, we show three benchmarking examples of the proof-pattern recognition across different libraries, notations, users, data types and lemma shapes. In the first example, ML4PG is used to detect non-trivial patterns arising in proofs across SSReflect libraries for Linear Algebra, Combinatorics and Persistent Homology. In the second and more applied example, it is used to help in the formulation of auxiliary lemmas when adapting the Computer Algebra methodology of CoqEAL to new domains. Finally, the third, industry-related example shows ML4PG clustering proofs of properties of the Java Virtual Machine in the scenario of team proof-development.
Keywords: Interactive Proofs, Coq, SSReflect, Machine Learning, Clustering.

1 Introduction

Donald Knuth famously compared Computer Programming to Art [15]. This comparison still holds for Interactive Theorem Provers (ITPs) [3]. The successful and efficient ITP programming relies on previous experience and ability to *"creatively"* adapt already used proof techniques and *patterns* in newly constructed proofs. This often requires a combination of mathematical and programming intuition; see e.g. [2]. This explains why the "steep learning curve" is often mentioned as one of the big obstacles to wider adoption of ITPs by professional mathematicians or industries alike.

In this paper, we are making the first steps towards automated detection of significant proof patterns in Coq/SSReflect [9]. Our main goal is to prove the concept: *it is possible to embed a lightweight statistical machine-learning tool into an ITP proof interface, and use it interactively to find non-trivial patterns in existing proofs and to aid new proof development.* Related work on using machine-learning in ITPs concerned hints in lemma generation for Isabelle/HOL [13], proof strategy discovery in Isabelle/HOL [1], speed-up in proof automation in HOL-Light [14] and statistical tactic analysis [8] in Isabelle.

We have described, in a companion paper [16], ML4PG – an extension to Proof General that automatically clusters Coq/SSReflect proofs. The ML4PG package for Proof General features three main functions:

* The work was supported by EPSRC grant EP/J014222/1.

F1. it works on the background of Proof General, and extracts some simple, low-level features from interactive proofs in Coq/SSReflect;

F2. it automatically sends the gathered statistics to a chosen machine-learning interface and triggers execution of a clustering algorithm of the user's choice;

F3. it does some gentle post-processing of the results given by the machine-learning tool, and displays families of related proofs to the user.

Section 2 will give an overview of ML4PG properties, full details of its implementation are given in [16]. In this paper, we do not focus on ML4PG implementation *per se*, although we use it for all the examples and experiments shown in this paper. Our main goal here is to show how useful the automated proof pattern detection can be. For this purpose, we devise three experiments to test ML4PG. Each example is designed to demonstrate a different aspect of proof-pattern recognition. ML4PG and all examples presented throughout this paper are available in [11].

Section 3 focuses on discovery of proof patterns in mathematical proofs across formalisations of apparently disjoint mathematical theories: Linear Algebra, Combinatorics and Persistent Homology. In this scenario, we use statistically discovered proof patterns to advance the proof of a given "problematic" lemma. In this case, a few initial steps in its proof are clustered against several mathematical libraries. In our example, the lemma in question is related to *nilpotent matrices* [5]. This section contains a detailed description of how ML4PG was used to discover some non-trivial proof patterns among 750 lemmas across 5 libraries, and how the detected proof clusters were used to advance the proof for this lemma. Notably, ML4PG discovered that the fundamental lemma of Persistent Homology [10], a result from a completely different context, follows the proof strategy that would suit for the proof of our lemma.

Section 4 tests ML4PG's functionality in a different area – verification of Computer Algebra algorithms as suggested by the CoqEAL methodology [7]. It is a different – but equally common – scenario of proof development: CoqEAL gives a general methodology to follow, but the exact role of over 1000 proofs and definitions in CoqEAL library may be unclear to us. In this case, we equally need guidance in lemma formulation, not only in proofs. In this section, we consider various proofs concerning a fast algorithm to compute the inverse of triangular matrices over a field. Again, ML4PG was able to discover significant clusters, and pointed us exactly to the results which could be used as hints to formulate the necessary lemmas and complete the proofs.

Section 5 takes one further step from mathematical to industrial applications of Coq and ML4PG. For this purpose, we chose the proofs of correctness of the Java Virtual Machine (JVM) given in [12]. Industrial scenario of interactive theorem proving may differ significantly from the mathematical scenario above. Namely, industrial verification tasks often feature a bigger number of routine cases and similar lemmas; and also such tasks are distributed across a team of developers. Here, the inefficiency of automated proving often arises when programmers use different notation to accomplish very similar tasks, and thus a lot of work gets duplicated, see also [6]. We tested ML4PG in exactly such scenario:

we assumed that a programming team has collectively developed proofs of *a) soundness of specification, and b) correctness of implementation* of Java byte code for a dozen of programs computing multiplication, powers, exponentiation, and other functions. Next, we show how ML4PG discovered common patterns among these proofs and relevant lemmas (around 150 training examples in total). The suggested clusters indeed helped to advance the proofs of properties a) and b) for the Java byte code of the factorial function.

These three examples show that statistically discovered proof clusters can find patterns in the proofs across different libraries, theories and even users. ML4PG works on the background of Proof General, and if called, provides clustering results almost instantly; thus, can be used interactively, as a handy tool on request. ML4PG can speed-up proof development by suggesting re-usable proof strategies, providing non-trivial suggestions about analogies between fragments of apparently un-related libraries, and detecting common patterns in proofs developed by a team. Finally, it may be used for educational purposes, as automated proof-pattern recognition may help to smooth the learning curve.

2 Automated Proof-Pattern Discovery with ML4PG

In this section, we give a brief introduction to proof-pattern recognition with ML4PG; full details of implementation can be found in [16,11]. We will use this section to explain the role of features **F1**–**F3** we mentioned in the introduction. Here, we also introduce some technical improvements to [16]: and these are *dynamic lemma numbering* and the *proof patch method.*

Example 1. Consider the following example of a lemma about number series:

Lemma 1 If $g : \mathbb{N} \to \mathbb{Z}$, then

$$\sum_{0 \leq i \leq n} (g(i+1) - g(i)) = g(n+1) - g(0).$$

and its proof in Table 1. Note that important proof features could be gathered in relation to subgoal shapes (left column), as well as tactics (right column).

The discovery of statistically significant features in data is a research area of its own in machine-learning, known as *feature extraction*, see [4]. Irrespective of the particular feature-extraction algorithm used, most pattern-recognition tools will require that the number of selected features is limited and fixed. In ML4PG, we design and implement our own method of proof feature extraction. There were two major challenges:

C1. To make the feature extraction method general enough to work with interactive proofs of any nature and complexity.

Example 2. One could consider general goal features such as "goal shape" (e.g. "associative-shape" or "commutative-shape"), or properties like "the subgoal embeds a hypothesis", "the subgoal is embedded into a hypothesis". However,

Goals and Subgoals	Applied Tactics
$\sum_{i=0}^{n}(g(i+1)-g(i)) = g(n+1)-g(0)$	
	`elim : n => [|n _].`
$\sum_{i=0}^{0}(g(i+1)-g(i)) = g(1)-g(0)$	
	`by rewrite big_nat1.`
$\sum_{i=0}^{n+1}(g(i+1)-g(i)) = g(n+2)-g(0)$	

	`rewrite sumrB big_nat_recr big_nat_recl addrC`
	`addrC -subr_sub -!addrA addrA.`

$g(n+2)+\sum_{i=0}^{n}g(i+1)-$ $\sum_{i=0}^{n}g(i+1)-g(0) = g(n+2)-g(0)$	
	`move : eq_refl; rewrite -subr_eq0; move/eqP => ->.`
$g(n+2)+0-g(0) = g(n+2)-g(0)$	
	`by rewrite sub0r.`
□	
	`Qed.`

Table 1. *Proof for Lemma* $\sum_{i=0}^{n}(g(i+1)-g(i)) = g(n+1)-g(0)$ *in SSReflect.*

gathering such features uniformly across any set of proofs would be hard, especially when working with richer theories and dependent types, where interesting cases of shapes and embeddings may not be automatically resolved by first-order unification; while some such cases would apply to one proof but not another. See Table 1: none of the features mentioned above determines the proof flow.

C2. To respect the restriction on the fixed size of the feature vectors while allowing to data-mine higher-order proofs and formulas of varied length.

To address challenges **C1**–**C2**, we designed a method of implicit tracking of proof properties, called the *proof trace method.* First, ML4PG automatically tracks simple, low level properties that apply to any possible subgoal, e.g. "the top symbol" or "the argument type". Further, these shallow features are taken in relation to the statistics of user actions on every subgoal: how many and what kind of tactics she applied, and what kind of arguments she provided to the tactics. Finally, a few proof-steps are taken in relation to each other. Table 2 illustrates the process of forming such *feature vectors.*

Thus, the *proof trace method* lets the lemma structure show itself through the proof steps it induces. Note that, using the two dimensions of Table 2, we gather statistics both *dynamically* (considering several proof steps instead of just one) and *relationally* (tracking correlation of statistics concerning goal shapes and applied tactics within a few proof steps). An advantage of this method is that it applies uniformly to any Coq library. ML4PG extracts all proof features during Coq compilation.

	tactics	*N tactics*	*arg type*	*arg is hyp?*	*top symbol*	*n subgoals*
g1	elim	1	nat	Hyp	equal	2
g2	rewrite	1	Prop	EL	equal	0
g3	rewrite	1	8×Prop	8×EL	equal	1
g4	move:; rewrite; move/	3	3×Prop	3× EL	equal	1
g5	rewrite	1	Prop	EL	equal	0

Table 2. ***A Table illustrating the work of ML4PG's feature extraction algorithm for the proof in Table 1.*** *Parameters inside the double lines are the extracted features used to form the* ***feature vectors****. Notation* g1-g5 *is used to denote five consecutive subgoals in the derivation. Columns are the properties of subgoals the feature extraction method tracks: the names of applied tactics, their number, types of arguments, link of the proof step to a hypothesis (Hyp), inductive hypothesis (IH) or a library lemma (EL); top symbol of the current goal, and the number of the generated subgoals.*

Clustered proofs may have a varied length, however, challenge **C2.** restricts us to a fixed number of features. As Table 2 shows, the default feature extraction algorithm implemented in ML4PG takes five proof steps to form one feature vector. If the proof is larger, it will form separate feature vectors for proof steps 6-10, 11-15, and so on, using the same feature extraction algorithm as applied to form Table 2. Additionally, ML4PG always extracts features from the last five proof steps the user makes. We call this method the *proof patch method*, as it allows one to cluster longer proofs by statistically analysing their fragments. Potentially, one small proof may resemble a fragment of a bigger proof, see Example 3.

To prepare the output of the feature extraction algorithm of Table 2 for statistical data-mining, ML4PG converts all the features into numbers. E.g., every tactic, type of a tactic argument, and a top symbol of a subgoal is assigned some random number; and then this number is used consistently across all proofs. The most significant algorithm in this numerical conversion concerns lemma numbering, see the forth column of Table 2. It has an effect on clustering results, especially if ML4PG works with big proof libraries, and the lemma numbering gives a big value spread. To avoid inaccuracies of the blind lemma numbering, ML4PG implements *dynamic lemma numbering*: during the Coq compilation and feature extraction, ML4PG starts with numbering first two lemmas Coq compiles, and then continues inductively to cluster more lemmas one by one, and re-number them according to the cluster proximity of one proof to another. As a result, similar lemmas are assigned close values.

Once all proof features are extracted, ML4PG is ready to communicate with *machine-learning interfaces*. Every machine-learning engine has its concrete format to represent feature vectors. ML4PG is built to be modular – that is, the feature extraction is first completed within the emacs environment, where the data is gathered in the format of hash tables, and then these tables are converted to the format of the chosen machine-learning tool (in our case, Matlab or Weka). ML4PG transforms the feature vectors to a *comma separated values* (csv) file in the case of Matlab; or to *arff* files in the case of Weka. In principle, extending the list of machine-learning engines does not require any further modifications to the feature extraction algorithm, but just defining new *translators*.

Next, ML4PG invokes the machine-learning engine. The ML4PG mechanism connecting to machine-learning interfaces is similar to the native mechanism of Proof General used to connect to ITPs. Namely, there is a synchronous communication between ML4PG and the machine-learning interfaces, which runs in the background waiting for ML4PG calls. The user can chose additional proof libraries to be clustered against the current proof: these libraries must be exported with the mechanism provided by ML4PG.

The next configuration option ML4PG offers is the choice of the particular *pattern-recognition algorithm*. We connected ML4PG only to *clustering algorithms* [4] – a family of *unsupervised learning methods*. Unsupervised learning is chosen when no user guidance or class tags are given to the algorithm in advance. There are several clustering algorithms available in Matlab (*K-means* and *Gaussian*) and Weka (*K-means*, *FarthestFirst* and *Expectation Maximisation*); we will use several of them in the coming sections. Clustering techniques divide data into n groups of similar objects (called clusters), where the value of n is a "learning" parameter provided by the user. Increasing the value of n means that the algorithm will try to separate objects into more classes, and, as a consequence, each cluster will contain fewer examples but with higher correlation.

Various numbers of clusters can be useful for proof mining: this may depend on the size of the data set, and on existing similarities between the proofs. ML4PG accommodates such choices. In general, small values of n are useful when searching for general proof patterns which can later be refined by increasing the value of n. However, extreme values are to be avoided: small values of n can produce meaningless over-sized clusters; whereas trivial clusters with just one proof may be found for big values of n.

In the machine-learning literature, there exists a number of heuristics to determine this optimal number of clusters, [20]. We used them as an inspiration to formulate our own algorithm for ML4PG, tailored to the interactive proofs. It takes into consideration the size of the proof library and an auxiliary parameter we introduce here – called *granularity*. This parameter is used to calculate the optimal number of proof clusters, using the formulas of Table 3. As a result, the user does not provide the value of n, but just decides on *granularity* in ML4PG menu, by selecting a value between 1 and 5, where 1 stands for a low granularity (producing big and general clusters) and 5 stands for a high granularity (producing small and precise clusters). See Example 4.

Results of one run of a clustering algorithm may differ from another, even on the same data set. This is due to the fact that clustering algorithms randomly choose examples to start from and form clusters relative to those examples. However, some clusters are found repeatedly – and frequently – in different runs. To gather sufficient statistical data from proofs, ML4PG automatically runs the chosen clustering algorithm 200 times at every call of clustering, and collects the frequencies of each cluster. To judge reliability of clusters, the ML4PG user can

Granularity	Number of clusters
1	$\lfloor l/10 \rfloor$
2	$\lfloor l/9 \rfloor$
3	$\lfloor l/8 \rfloor$
4	$\lfloor l/7 \rfloor$
5	$\lfloor l/6 \rfloor$

Frequency parameter	Frequency Threshold
1	5%
2	15%
3	30%

Table 3. ML4PG formulas computing clustering parameters. Left: the formula computing the number of clusters given the granularity value, where l is the number of lemmas in the library. **Right:** the formula computing frequency thresholds given a frequency parameter.

choose one of the three *frequency thresholds* shown in Table 3. If the frequency of a cluster falls below the pre-set threshold, the corresponding proofs will not be displayed to the user. High frequencies suggest a high correlation among the proofs of a cluster; but lower frequencies can sometimes be useful in search of less obvious statistical correlation between proofs, see Example 5.

In addition to the frequency threshold, there is another parameter which ensures the reliability of the clusters. Clustering algorithm output contains not only clusters but also their *proximity* values. This measure ranges from $+1$, indicating points that are very distant from other clusters, through to 0, indicating points that are not distinctly in one cluster or another, and to -1, indicating points that are probably assigned to the wrong cluster. We have fixed 0.5 as an accuracy threshold, and all the clusters whose measure is under such value are ignored by ML4PG.

Finally, after discarding clusters with low proximity and those which fall below the pre-set threshold, the remaining clusters with lemma names and frequencies are displayed on Proof General panel.

3 Proof Patterns in Mathematical Proofs

Development of interactive provers has led to the creation of big data sets of libraries and development of varied infrastructures for formal mathematical proofs. However, these frameworks usually involve thousands of definitions and theorems (for instance, there are approximately 4200 definitions and 15000 theorems in the case of the formalisation of the Odd Order theorem [18]). It is difficult to trace them to find patterns which could be reused in the proof of a new theorem. Therefore, a tool which could detect proof strategies arising in mathematical proofs across several libraries will make the proof development task easier. In order to illustrate this fact, let us consider the following two lemmas together with Lemma 1.

Lemma 2 Let M be a square matrix and n be a natural number such that $M^n = 0$, then $(1 - M) \times \sum\limits_{i=0}^{n-1} M^i = 1$.

Lemma 3 Let $\beta_i^{j,k} : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to \mathbb{Z}$, then

$$\beta_m^{k,l} - \beta_m^{k,n} = \sum_{1 \leq i \leq k} \sum_{l < j \leq n} (\beta_m^{j,p-1} - \beta_m^{j,p}) - (\beta_m^{j-1,p-1} - \beta_m^{j-1,p}).$$

These three lemmas come from different contexts. Lemma 2 states a result about *nilpotent* matrices (a square matrix M is *nilpotent* if there exists an n such that $M^n = 0$). Lemma 3 is a generalisation of the *fundamental lemma of Persistent Homology*, the actual lemma and its formalisation can be seen in [10]. Finally, Lemma 1 is a basic fact about summations.

When proving Lemma 2, it is difficult, even for the expert user, to get the intuition that she can reuse the proofs of Lemmas 3 and 1. There are several reasons for this. First of all, the formal proofs of these lemmas are in different libraries (the proof of Lemma 2 is in a library about matrices, the proof of Lemma 3 is in a library about Persistent Homology, and the proof of Lemma 1 in a library about basic results in summations); also, it is hard to establish a conceptual connection among them. Moreover, although the three lemmas involve summations, the type of the terms of those summations are different. Therefore, search based on types or keywords would not provide any valuable information. Even search of all the lemmas involving summations is not useful, since there are more than 250 lemmas – a considerable amount for handling them manually.

However, if Lemmas 3 and 1 are suggested when proving Lemma 2, the expert would be able to spot the similarities among them, and notice the following proof pattern.

Proof Strategy 1

1. *Apply induction on n.*
   (a) *Prove the base case (a simple task).*
   (b) *Prove the inductive case:*
      i. *expand the summation,*
      ii. *cancel the terms pairwise,*
      iii. *the only terms remaining after the cancellation are the first and the last one.*

For instance, using the above proof strategy in the case of Lemma 2, the proof of the inductive case is as follows.

Proof.

$$\begin{aligned}(1 - M) \times \sum_{i=0}^{n-1} M^i &= \sum_{i=0}^{n-1} ((1 - M) \times M^i) \\ &= \sum_{i=0}^{n-1} (M^i - M^{i+1}) && \textit{(step i.)} \\ &= M^0 - M^1 + M^1 - M^2 + \ldots + M^{n-1} - M^n && \textit{(step ii.)} \\ &= M^0 - M^n && \textit{(step iii.)} \\ &= 1 - 0 = 1.\end{aligned}$$

□

It is worth noting that we cannot blindly apply Proof Strategy 1, or the tactics applied in the proof of Lemmas 3 and 1 to the proof of Lemma 2 since there are small nuances. Nevertheless, the general idea of the proof pattern can be followed to finish the proof and, with some minor modification, most of the tactics can be reused.

ML4PG will suggest Lemmas 3 and 1 when proving Lemma 2, if we use the following settings for statistical pattern recognition. First of all, we consider 5 SSReflect libraries for clustering: bigop (devoted to generic indexed big operations like $\sum\limits_{i=0}^{n} f(i)$ or $\bigcap\limits_{i \in I} f(i)$), matrix, binomial (which defines combinatorial notions and include several results involving summations), a small library about summations (which includes the proof of Lemma 1) and the library about Persistent Homology formalised in [10]. These libraries involve approximately 750 lemmas, and so the ML4PG will analyse a data set of 750 examples; here, we connect it to the *K-means* clustering algorithm in *Weka*.

Example 3 (Proof patches). Let us note that Proof Strategy 1 can be applied twice in the proof of Lemma 3, firstly in the inner summation and subsequently in the remaining proof. Then, 2 of the 15 suggestions provided by ML4PG in this case come from the proof of Lemma 3.

The configuration of the granularity parameter can be approached in two different ways:

- *top-down*: this approach suggests first using a small value for the granularity to obtain a general proof pattern, and then refine that pattern increasing the granularity value.
- *bottom-up*: in this approach, a high value for the granularity is used to see what are the most similar lemmas and, then, decrease the granularity value to see more general – and potentially less trivial – patterns.

Example 4 (Proof Granularity). In our case, we use the top-down approach starting with the default granularity value of 3. Using this value ML4PG obtains 15 suggestions which are related to lemmas about summations including Lemmas 3 and 1. Increasing the granularity level to 4, ML4PG discovers Lemmas 3 and 1, and also the following lemma.

Lemma 4 Let M be a nilpotent matrix, then there exists a matrix N such that $N \times (1 - M) = 1$.

At first sight, the proof of this lemma does not seem to fit Proof Strategy 1, since the statement of the lemma does not involve summations. However, inspecting its proof, we can see that it uses $\sum\limits_{i=0}^{n-1} M^i$ as witness for N and then follows Proof Strategy 1. In fact, if we use the highest granularity level, this is the only suggestion given by ML4PG to prove Lemma 2, since, apart from the step to provide the witness, the proofs of both lemmas are practically the same.

Therefore, ML4PG can capture non-trivial mathematical proof patterns arising across different data types, a variety of lemma shapes and libraries. Moreover, it allows to change machine-learning settings to obtain clusters of varied precision.

4 Proof Patterns for Importing Proof Methods

There is a trend in ITPs to develop general purpose methodologies to aid in the formalisation of a family of related proofs. However, although the application of a methodology is straightforward for its developers, it is usually difficult for an external user to decipher the key results to import such a methodology into a new development. Therefore, tools which can capture methods and suggest appropriate lemmas based on proof patterns would be valuable. Here, we show how ML4PG can be useful in this context with an example coming from the formal proof of the correctness of a Computer Algebra algorithm.

Most algorithms in modern Computer Algebra systems are designed to be efficient, and this usually means that their verification is not an easy task. In order to overcome this problem, a methodology based on the idea of *refinements* was presented in [7], and was implemented as a new library, built on top of the SSReflect libraries, called *CoqEAL*. Roughly speaking, the approach to formalise efficient algorithms followed in [7] can be split into three steps:

S1. define the algorithm relying on rich dependent types, this will make the proof of its correctness easier;
S2. refine such a definition to an efficient algorithm described on high-level data structures; and,
S3. implement it on data structures which are closer to machine representations.

The CoqEAL methodology is clear and the authors have shown that it can be extrapolated to different problems. Nevertheless, this library contains approximately 400 definitions and 700 lemmas; then, the search of proof strategies inside this library is not a simple task which could be undertaken manually.

In order to illustrate this, let us consider the formalisation of a fast algorithm to compute the inverse of triangular matrices over a field with 1s in the diagonal using the CoqEAL methodology. Our interest in the inverse of this kind of matrices comes from its application in the context of Discrete Morse Theory [19], where they are used to speed-up the computation of homology groups. SSReflect already implements the matrix inverse relying on rich dependent types using the `invmx` function; then, we only need to focus on the second and third steps of the CoqEAL methodology.

Using an algorithm specially designed to efficiently compute the inverse of triangular matrices with 1s in the diagonal, we define a function called `fast_invmx` using high-level data structures.

Algorithm 1 Let M be a square triangular matrix of size n with 1s in the diagonal; then `fast_invmx(M)` is recursively defined as follows.

- If $n = 0$, then `fast_invmx(M)=1%M` (where `1%M` is the notation for the identity matrix in SSReflect).
- Otherwise, we can decompose M in a matrix with four components: the top-left element, which is 1; the top-right line vector, which is null; the bottom-left column vector C; and the bottom-right $(n-1) \times (n-1)$ matrix N; that is, $M = \left(\begin{array}{c|c} 1 & 0 \\ \hline C & N \end{array}\right)$. Then, we define `fast_invmx(M)` as:

  $$\texttt{fast\_invmx(M)} = \left(\begin{array}{c|c} 1 & 0 \\ \hline -\texttt{fast\_invmx(N) *m C} & \texttt{fast\_invmx(N)} \end{array}\right)$$

  where `*m` is the notation for matrix multiplication in SSReflect.

Subsequently, we should prove the equivalence between the functions `invmx` and `fast_invmx`. Proving this result is not trivial due to the different nature of the algorithms: the former is a general algorithm to compute the inverse of matrices – using adjugate matrices and determinants; on the contrary, the latter is an ad-hoc efficient algorithm for a special case of triangular matrices, which takes advantage of the shape of those matrices to obtain their inverse.

In the CoqEAL library, there are just three lemmas devoted to prove the equivalence between a matrix algorithm and its efficient version. Namely, those lemmas are related to the multiplication, the rank and the determinant of matrices. However, the strategies followed to prove those equivalence lemmas are ad-hoc for the concrete algorithms, and the only common step which could be reused from them in our concrete case is the application of induction on the size of the matrix. Therefore, in this situation, it makes sense to ask ML4PG for some hint that could help us to tackle the proof before trying to prove it by brute force.

We configure ML4PG as follows. First of all, we consider both the matrix library of SSReflect and the CoqEAL library for clustering – they involve approximately 1000 lemmas. This time, we connect ML4PG to the *Gaussian* clustering algorithm in Matlab. Moreover, we use a bottom-up approach to configure the granularity parameter.Finally, in order to configure the frequencies parameter, our experience shows that the analysis of frequencies may give two opposite effects.

- On the one hand, high frequencies suggest that the proofs found in clusters have a high correlation, and that is a desirable property.
- On the other hand, proofs with too high correlation may be too trivial for providing interesting proof hints. Therefore, it is sometimes useful to look for proof clusters with lower frequencies – as they may potentially contain those non-trivial analogies.

Example 5 (Proof frequencies). For our CoqEAL experiments, we start using 5 as granularity parameter and 3 as frequencies parameter (see Table 3); with these settings, ML4PG will provide similar lemmas with a high correlation among them. However, ML4PG does not find any proof cluster for our proof. Therefore, we decrease both granularity and frequencies parameters to 3 and 2 in order to obtain more general clusters and with lower correlation. Using these settings, ML4PG suggests 10 lemmas. Three of them are the ones about efficient multiplication, rank and determinant; but, as we have previously said, they do not provide any hint to finish our proof. Among the most interesting suggestions was the following unicity lemma.

Lemma 5 Let M_1 and M_2 be two square matrices such that $M_1 \times M_2 = 1$ (where 1 is the identity matrix); then, M_2 is the inverse of M_1.

Then, to prove the equivalence between `invmx` and `fast_invmx`, it is enough to prove that given a triangular matrix `M`, then `M *m fast_invmx(M)= 1%M`. This result is easy to prove.

Proof. Apply induction on the size of the matrix.

- The base case is trivial.
- In the inductive case, `M *m fast_invmx(M)` is equal to:

$$\left(\begin{array}{c|c} 1 & 0 \\ \hline \texttt{C} & \texttt{N} \end{array}\right) \texttt{*m} \left(\begin{array}{c|c} 1 & 0 \\ \hline -\texttt{fast\_invmx(N) *m C} & \texttt{fast\_invmx(N)} \end{array}\right) \quad =$$

$$\left(\begin{array}{c|c} 1 & 0 \\ \hline \texttt{C - N *m fast\_invmx(N) *m C} & \texttt{N *m fast\_invmx(N)} \end{array}\right)$$

Applying the inductive hypothesis, the result is proven.

□

Therefore, ML4PG does not provide here any proof pattern and the correlation of our current proof with the suggested lemmas is not too high, but it has helped us to *formulate* an auxiliary lemma to finish our original proof.

Once we have proven the equivalence between the two matrix inverse algorithms, we can focus on the third step of the CoqEAL methodology. It is worth mentioning that neither `invmx` nor `fast_invmx` can be used to actually compute the inverse of matrices. These functions cannot be executed since the definition of matrices is locked in SSReflect to avoid the trigger of heavy computations during deduction steps. Using step **S3.** of the CoqEAL methodology we can overcome this pitfall. In our case, we implement the function `cfast_invmx` using lists of lists as the low level data type for representing matrices.

CoqEAL provides the executable counterpart of most of the matrix operations included in the matrix library of SSReflect. Moreover, the correctness of those functions is proved through *translation* lemmas, which state the equivalence between the executable version and the abstract version. Then, the implementation of `cfast_invmx` is almost a direct translation of `fast_invmx` using the executable counterparts of matrix operations provided by CoqEAL; and, the proof of correctness can be achieved applying the translations lemmas of the operations involved in the definition of the algorithm (see [7]). If ML4PG is called, it finds that all the translation lemmas form a unique cluster.

Therefore, ML4PG can help to import proof methods into new developments in two different ways. First of all, ML4PG can detect typical proof patterns which are followed in a methodology (the proof pattern of translation lemmas). Moreover, even in the cases where there is no common proof strategy, ML4PG can suggest lemmas that help in the formulation of auxiliary results, making the proof development easier.

5 Proof Patterns in Industrial Proofs

ITPs have been successfully used in industry to verify the correctness of hardware and software systems. In this context, proofs usually have a certain regularity involving several routing cases and similar lemmas. However, the proofs are often developed by a team, where users have their own list of definitions and lemmas in different notations. Thus, in team-based developments, it would be extremely helpful to use a tool that could detect proof patterns across different users, notations and libraries.

We examine the suitability of ML4PG for this task with the formalisation of a simple model of the Java Virtual Machine in Coq/SSReflect. The Java Virtual Machine (JVM) [17] is a stack-based abstract machine which can execute Java byte code. We have modelled an interpreter for JVM programs in Coq. From now on, we refer to our machine as "CJVM" (for Coq JVM).

Given a specific Java method, we can translate it to Java byte code using a tool such as `javac` of Sun Microsystems. Such a byte code can be executed in CJVM provided a schedule, and the result will be the state of the JVM at the end of the schedule. Moreover, we can prove theorems about the CJVM model behaviour when interpreting that byte code. The byte code associated with the factorial program can be seen in Figure 1.

```
static int factorial(int n)
{
  int a = 1;
  while (n != 0){
    a = a * n;
    n = n-1;
    }
  return a;
}
```

```
 0 : iconst 1
 1 : istore 1
 2 : iload 0
 3 : ifeq 13
 4 : iload 1
 5 : iload 0
 6 : imul
 7 : istore 1
 8 : iload 0
 9 : iconst 1
10 : isub
11 : istore 0
12 : goto 2
13 : iload 1
14 : ireturn
```

```
Fixpoint helper_fact (n a) :=
match n with
| 0 => a
| S p => helper_fact p (n * a)
end.

Definition fn_fact (n : nat) :=
     helper_fact n 1.
```

Fig. 1. ***Factorial function***. ***Left:*** *Java program for computing the factorial of natural numbers.* ***Centre:*** *Java byte code associated with the Java program.* ***Right:*** *tail recursive version of the factorial function in Coq.*

The state of the CJVM consists of 4 fields: a *program counter* (a natural number), a set of registers called *locals* (implemented as a list of natural numbers), an operand *stack* (a list of natural numbers), and the byte code *program* of the method being evaluated.

Java byte code, like the one presented in Figure 1, can be executed within CJVM. However, more interesting than mere executing Java byte code, we can prove the correctness of the implementation of the Java byte code programs using Coq. For instance, in the case of the factorial program, we can prove the following theorem, which states the correctness of the `factorial` byte code.

Lemma 6 $\forall n \in \mathbb{N}$, CJVM produces a state which contains $n!$ on top of the stack running the byte code associated with the factorial program with n as input.

The proof of theorems like the one above always follows the same methodology exported from ACL2 proofs about Java Virtual Machines [12] and which consists of the following three steps.

(1) Write the specification of the function, write the algorithm, and prove that the algorithm satisfies the specification.
(2) Write the JVM program within Coq, define the function that schedules the program (this function will make CJVM run the program to completion as a function of the input to the program), and prove that the resulting code implements this algorithm.
(3) Prove total correctness of the Java byte code.

Using this methodology, we have proven the correctness of a dozen of programs related to arithmetic (multiplication of natural numbers, exponentiation of natural numbers and so on). The proof of each theorem was done independently from others to model a distributed proof development.

Therefore, we simulated the following scenario. Suppose a new developer tackles for the first time the proof of Lemma 6, and she knows the general methodology to prove it and has access to the library of programs previously proven by other users. This situation is similar to the one presented in Section 4 with an additional problem: the different notation employed by different users obscure some common features. ML4PG would be a good alternative to the manual search for proof patterns.

The settings of ML4PG allowing to find interesting proof patterns are the same for all three steps listed above. We consider all the libraries related to the proofs of Java byte code programs (they involve approximately 150 lemmas). We connect ML4PG to the *k-means* library in *Matlab*. Moreover, we use the default value of 3 for the granularity parameter and the frequency value of 3.

Let us focus on the first step of the methodology – that is, the proof of the equivalence between the specification of the factorial function (which is already defined in SSReflect) and the algorithm. The Java factorial function is an iterative function; then, the algorithm is written in Coq as a tail recursive function, see the right side of Figure 1. In general, all the tail recursive functions are defined using an auxiliary function, called the *helper*, and a wrapper for such a function. The suggestions provided by ML4PG in this case are the proofs of step

(1) for three iterative programs: the multiplication, the exponentiation and the power of natural numbers. All of them follow the same proof strategy which can be also applied in the case of factorial:

Proof Strategy 2 *Prove an auxiliary lemma about the helper considering the most general case. For example, if the helper function is defined with formal parameters n, m, and a and the wrapper calls the helper initializing a to 0, the helper theorem must be about* `(helper n m a)`*, not just about the special case* `(helper n m 0)`*. Subsequently, instantiate the lemma for the concrete case.*

In order to prove that the Java byte code implements the factorial algorithm, ML4PG suggests 4 lemmas which are used to that aim in other cases. All of those programs are iterative and involve a loop. The strategy which is followed in those proofs is the following one.

Proof Strategy 3 *Prove that the loop implements the helper using an auxiliary lemma. Such a lemma about the loop must consider the general case as in the case of Proof Strategy 2. Subsequently, instantiate the result to the concrete case.*

Finally, ML4PG finds that all the lemmas involved in the proof of the total correctness of the programs for different functions are similar and follow the same proof pattern which consists in applying the lemmas obtained from steps (1) and (2). Following these guidelines, Lemma 6 can be formalised in Coq by analogy with other similar proofs, obtaining as a result the proof of the correctness of the factorial Java byte code, see [11] for the full proof.

6 Conclusions and Further work

In this paper, we have presented three examples, of very different nature, to test the capabilities of statistical proof-pattern recognition. Various technical details of ML4PG implementation may be subject to change in the future: e.g. the feature extraction mechanism or the clustering algorithms may be further tuned. Our experiments convince us that a tool like ML4PG can be a practical addition to interactive proof development; its fully functional version can be downloaded from [11]. Among the methods that are crucial for ML4PG success are the *proof trace* and the *proof-patch* methods, as well as the model of interactive and modular interfacing between Proof General and machine-learning engines.

The *proof trace method* is general enough to be applied to other ITPs, such as Isabelle/HOL or HOL4, without any special hindrance. In addition, Proof General provides a common interface for several ITPs. Therefore, it is appealing to use machine-learning techniques to automatically find and import proof patterns across various ITPs in the future.

A more technical research line of future work is the development of ML4PG as a distributed tool. As we have shown in Section 5, ML4PG can be helpful for team-based developments. However, current implementation of ML4PG is centralised; this means that the user can obtain proof clusters of the libraries available on her computer. Then, we think that a client-server architecture, where the proof information is shared among several users could also be useful.

References

1. D. Basin, A. Bundy, D. Hutter, and A. Ireland. *Rippling: Meta-level Guidance for Mathematical Reasoning.* Cambridge University Press, 2005.
2. N. Benton. Machine Obstructed Proof: How many months can it take to verify 30 assembly instructions?, 2006.
3. Y. Bertot and P. Castéran. *Coq'Art: the Calculus of Constructions.* Springer-Verlag, 2004.
4. C. Bishop. *Pattern Recognition and Machine Learning.* Springer, 2006.
5. R. Brualdi and H. Ryser. *Combinatorial Matrix theory.* Cambridge University Press, 1991.
6. A. Bundy, D. Hutter, C. Jones, and J S. Moore. AI meets Formal Software Development (Dagstuhl Seminar 12271). *Dagstuhl Reports*, 2(7):1–29, 2012.
7. M. Dénès, A. Mörtberg, and V. Siles. A Refinement Based Approach to Computational Algebra in Coq. In *Proceedings 3rd International Conference on Interactive Theorem Proving 2012 (ITP'12)*, volume 7406 of *LNCS*, pages 83–98, 2012.
8. Hazel Duncan. *The use of Data-Mining for the Automatic Formation of Tactics.* PhD thesis, University of Edinburgh, 2002.
9. G. Gonthier and A. Mahboubi. An introduction to small scale reflection. *Journal of Formalized Reasoning*, 3(2):95–152, 2010.
10. J. Heras, T. Coquand, A. Mörtberg, and V. Siles. Computing Persistent Homology within Coq/SSReflect, 2012. arXiv:1209.1905.
11. J. Heras and E. Komendantskaya. ML4PG: downloadable programs, manual, examples, 2012–2013. www.computing.dundee.ac.uk/staff/katya/ML4PG/.
12. J S. Moore. *Models, Algebras and Logic of Engineering Software*, chapter Proving Theorems about Java and the JVM with ACL2, pages 227–290. IOS Press, 2004.
13. M. Johansson, L. Dixon, and A. Bundy. Conjecture synthesis for inductive theories. *Journal of Automated Reasoning*, 47(3):251–289, 2011.
14. C. Kaliszyk and J. Urban. Learning-assisted Automated Reasoning with Flyspeck, 2012. arXiv:1211.7012.
15. Donald E. Knuth. *The Art of Computer Programming, Volume I: Fundamental Algorithms.* Addison-Wesley, 1968.
16. E. Komendantskaya, J. Heras, and G. Grov. Machine Learning for Proof General: interfacing interfaces, 2012. 25 pages. Submitted to post-proceedings of User Interfaces for Interactive Theorem Proving (UITP), arXiv:1212.3618.
17. T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. The Java Virtual Machine Specification: Java SE 7 Edition, 2012.
18. Mathematical components team. Formalization of the Odd Order theorem. Technical report, 2012. `http://www.msr-inria.inria.fr/Projects/math-components`.
19. A. Romero and F. Sergeraert. Discrete Vector Fields and Fundamental Algebraic Topology, 2010. arXiv:1005.5685v1.
20. R. Xu and D. Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, 2005.