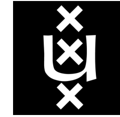
Vrije Universiteit Amsterdam

Universiteit van Amsterdam



Master Thesis

Premise Selection for Lean 4

Author: Alistair Geesing (2750928)

1st supervisor:	Jasmin Blanchette
daily supervisor:	Jannis Limperg
2nd reader:	Anne Baanen

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

June 7, 2023

Abstract

Premise selection is a key component for the Sledgehammer tool which lets the user of a proof assistant use automated theorem provers. The Isabelle proof assistant includes several premise selection algorithms: MePo, MaSh Naive Bayes, MaSh k-Nearest Neighbors, and MeSh. Lean 4 does not currently provide a Sledgehammer, nor does it have an implementation of the premise selection algorithms necessary for a Sledgehammer implementation.

By implementing premise selection algorithms, a Sledgehammer can be developed for Lean 4. Isabelle's premise selection algorithms are ported to Lean 4 and the necessary infrastructure is developed. Performance of the algorithms is evaluated and compared to their Isabelle counterparts using the original research that produced the MaSh and MeSh premise selection algorithms.

Performance of the MaSh NB, MaSh k-NN, and MeSh algorithms in Lean 4 is comparable to their Isabelle counterparts. The MePo algorithm does not achieve comparable performance, however, because of differences between the experiment and its Isabelle counterpart.

Most of the ported algorithms can be successfully used to perform premise selection in Lean 4. When a Sledgehammer is implemented for Lean 4, real-world performance of the premise selection algorithms can be measured.

Acknowledgements

I would like express my gratitude to Jasmin Blanchette for their guidance during research and writing this thesis.

I would also like to thank Jannis Limperg for their supervision and help in getting me started with Lean 4 metaprogramming.

I would also like to note my appreciation for my family's continuous support.

Contents

List of Figures v

List of Tables vi

1 Introduction 1

2 Background 3

2.1 Proof assistants . . . 3
2.2 Automated theorem proving . . . 4
2.3 Sledgehammer . . . 5
2.4 Premise selection . . . 5
2.5 Lean 4 . . . 5
2.5.1 Types . . . 6
2.5.2 Formal proofs . . . 6
2.5.3 Compilation . . . 7
2.5.4 General-purpose programming and metaprogramming . . . 7

3 Building blocks of premise selection in Lean 9

3.1 Extracting information from modules . . . 9
3.2 Going from constants to facts . . . 9
3.2.1 Theorems and axioms . . . 10
3.2.2 Definitions . . . 10
3.2.3 Constructors . . . 10
3.2.4 Boring facts and incomplete proofs . . . 11
3.3 Visibility relation . . . 11
3.3.1 Respecting the visibility relation . . . 12
3.4 Cataloging facts . . . 12
3.4.1 Building and updating the state . . . 12

3.4.1.1 Aliases . . . 14
3.4.1.2 Auxiliary definitions . . . 14
3.4.2 Disk persistence . . . 14
3.4.2.1 On-disk representation . . . 15
3.4.2.2 Implications of using the `.olean` format . . . 16
3.4.3 Non-monotonic changes . . . 16
3.4.4 Transient changes . . . 17
3.5 MeSh . . . 17

4 Implementing MePo **18**
4.1 The MePo algorithm . . . 18
4.2 MePo in Isabelle . . . 21
4.3 MePo in Lean . . . 22
4.4 Cataloging facts . . . 22

5 Implementing MaSh **24**
5.1 Features . . . 24
5.2 Naive Bayes . . . 25
5.2.1 The Naive Bayes algorithm . . . 25
5.2.2 Naive Bayes in Isabelle . . . 26
5.2.3 Naive Bayes in Lean . . . 27
5.3 k-Nearest Neighbors . . . 27
5.3.1 The k-Nearest Neighbors algorithm . . . 27
5.3.2 k-Nearest Neighbors in Isabelle . . . 28
5.3.3 k-Nearest Neighbors in Lean . . . 29
5.4 Cataloging facts . . . 29

6 Evaluation **30**
6.1 Experiment design . . . 30
6.1.1 Metrics . . . 31
6.2 Results . . . 31

7 Discussion **35**
7.1 Limitations of the experiment . . . 35
7.2 MePo performance . . . 36
7.3 MaSh performance . . . 36

8 Related work **37**
8.1 SInE . . . 37
8.2 Premise selection using a support vector machine . . . 37
8.3 DeepMath . . . 38

9 Conclusion **39**
9.1 Future work . . . 39

References **41**

List of Figures

3.1 Definition of the premise selection state 13

6.1 Average k-coverage 34

List of Tables

6.1 Statistics of the formalizations used in experiments 32
6.2 Average full recall 32
6.3 Average AUC (%) 33
6.4 Average rank of necessary dependencies 33

1

Introduction

The field of mathematics provides the underpinnings of many of the sciences. Countless examples of practical applications of mathematical concepts exist: physics relies on mathematics to model reality; economics, social sciences and medicine use statistics to infer relationships between real-world phenomena; engineering and architecture need models from physics and of material properties to design infrastructure and buildings; and so on.

Mathematics needs to be consistent to ensure that it makes sense and is reliable in practical applications. Indeed, without a rigorous proof which builds on existing mathematics a mathematical statement is only considered conjecture, not fact. Building mathematics on top of conjectures produces a precarious structure. By disproving one such conjecture the house of cards may collapse. Similarly, an incorrect proof causes other proofs that depend on it to become incorrect.

Historically, proofs have been written in the form of formulas mixed with prose. Verifying these proofs requires perfect mathematicians able to spot subtle mistakes that lead to an incorrect proof. Perfect mathematicians do not exist, of course. By writing the proof in a formal language its verification can be done mechanically by a computer. Formal languages have well defined syntax and semantics that leave no room for interpretation and can be processed by software. Assuming that the combination of hardware and software, including the verification program, performing verification is free of errors, then verified formal proofs are guaranteed to be correct.

The development of formal proofs has lead to tools such as proof assistants, e.g. Isabelle (1) and Lean (2). A proof assistant provides its users with an environment that helps them to write formal proofs. This environment includes an implementation of a formal proof language, a repository of facts, and tools to aid in writing proofs.

One such tool is Sledgehammer (3), which is implemented for the Isabelle proof assistant. Sledgehammer allows the user to invoke an automated theorem prover (ATP) on a conjecture. The ATP attempts to automatically construct a proof for the conjecture. By offloading the burden of proving (simpler) conjectures to an automated tool the user can become more productive.

Ideally, an ATP would consider all facts known to the proof assistant when attempting to construct a proof. However, ATPs do not work well with the large number of facts (tens of thousands or more) contained in a proof assistant's repository (4). When an ATP only needs to consider a smaller number of facts its performance benefits. Therefore, Sledgehammer performs a filtering step on the set of facts to reduce its size. This filtering step is known as premise selection.

It is not possible to know a priori which facts will and will not be used in a proof. Instead, a premise selection algorithm determines a relevance score which estimates the probability that a fact will be used to prove a given conjecture. A fixed number of facts with the highest relevance scores are passed to the ATP. Sledgehammer implements three premise selection algorithms: MePo, MaSh Naive Bayes, and MaSh k-Nearest Neighbors. Additionally, Sledgehammer includes MeSh (specifically MeSh-Naive Bayes and MeSh-k-NN), which combines the results of these premise selection algorithms.

The Lean proof assistant does not currently have an equivalent of Isabelle's Sledgehammer. For this thesis I have implemented and evaluated the premise selection algorithms present in Isabelle for Lean 4. Building on this work a Sledgehammer for Lean can be implemented in the future. Currently, the premise selection algorithms are accessed through tactics that print the premise selection results, e.g. the `mepo` and `mashknn` tactics found in the `Hammer.PS` module. The implementation can be found at https://github.com/aalistairr/ps-lean.

Chapter 2 will provide the necessary background for understanding the following chapters. Chapter 3 describes the building blocks of premise selection in Lean. Chapters 4 and 5 explain the MePo and MaSh premise selection algorithms and their implementation in Isabelle and Lean. Chapter 6 lines out the experiment used to measure the performance of the premise selection algorithms and presents its results. Chapter 7 discusses the experiment results. Chapter 8 gives an overview of several research papers that present alternative premise selection algorithms. Finally, chapter 9 concludes and presents possible future work.

2

Background

An implementation of premise selection algorithms in Lean serves to support a Sledgehammer implementation. This chapter gives background on aspects that are relevant to premise selection algorithms as used by Sledgehammer.

2.1 Proof assistants

By proving a conjecture using a formal proof, its validity can be checked by a computer. Instead of writing the proof of a conjecture in the traditional mix of formulas and prose, the conjecture and its proof are expressed in a formal language. Verification of traditional proofs depends on the ability of reviewers to spot errors, which itself is an error-prone task. In contrast, assuming that the trusted computing base (the combination of hardware and software, including the verification program) is correct, a formal proof verified by a computer is guaranteed to be correct.

A formal language defines syntax and semantics to allow expressing conjectures and their proofs. For example, modus ponens

$$\frac{P \to Q \qquad P}{Q}$$

might be expressed in Lean as follows:

```
variable (p q : Prop)
variable (r : p → q)
variable (h : p)
example : q := r h
```

To increase the expressivity of the language, primitives such as inductive types may be included. More complex mathematical concepts such as natural and real numbers or user

friendly ways of writing proofs can then be expressed. For example, consider the following example from the Lean tutorials (5):

```
example (a b c : ℝ) : (a * b) * c = b * (a * c) := by
  rw [mul_comm a b]
  rw [mul_assoc b a c]
```

With the `rw` tactic, commutativity and associativity of multiplication are used to prove the conjecture.

Of the concepts used in the example the core language of Lean understands only a very limited number directly. It has no understanding of what real numbers, multiplication or even the `by` and `rw` syntax are. The user friendly syntax is compiled into the simpler core language:

```
fun (a b c : ℝ) =>
  Eq.mpr
    (id (mul_comm a b ▸ Eq.refl (a * b * c = b * (a * c))))
    (Eq.mpr
      (id (mul_assoc b a c ▸ Eq.refl (b * a * c = b * (a * c))))
      (Eq.refl (b * (a * c))))
```

Clearly, such a proof would be tedious to write. However, by restricting what can be directly expressed in the core language, its scope and therefore also its complexity are reduced.

Proof assistants provide an environment that enables the user to write and verify formal proofs. To that end, a proof assistant maintains a repository of facts (e.g. axioms and theorems), has tools to aid in writing proofs (such as tactics), and implements an algorithm to verify proofs written in its formal language.

2.2 Automated theorem proving

Automated theorem provers (ATPs) are programs that are capable of proving conjectures automatically. An ATP needs to be provided with an appropriate set of axioms to be able to prove a conjecture. ATPs will fail for a number of reasons; for example when the set of provided axioms is not sufficient to construct the proof or if too many axioms are provided.

2.3 Sledgehammer

The Isabelle proof assistant includes a tool known as Sledgehammer (3), which acts as an interface between the proof assistant and external ATPs. Less interesting (and perhaps also tedious) aspects of proof construction can be automated using Sledgehammer.

Computational limits make it infeasible for ATPs to consider all possible facts when constructing a proof. Tools such as Sledgehammer must therefore perform a filtering step on facts, known as premise selection.

After premise selection, formulas are translated from higher-order logic to first-order logic. The first-order logic formulas are passed to external ATPs in order to attempt construction of a proof. Once a proof has been found, an attempt is made to restructure it using Isabelle's resolution prover Metis, otherwise a more detailed proof is used. Finally, Sledgehammer reduces the set of facts used in the proof by iteratively constructing proofs, passing only the facts used by the previously found proof.

2.4 Premise selection

Premise selection consists of making educated guesses as to which facts may be relevant for the proof of a conjecture and which facts may be dismissed. A premise selection algorithm is configured to emit a certain number of facts that are considered most likely to be used in the proof of a conjecture. The rank of a fact chosen by premise selection (its position in the results) is an indication of its relevance, relative to the other facts that were selected. Facts with a lower rank have been determined, according to the algorithm, to have a higher relevance than other facts.

Several classes of premise selection algorithms exist, based on heuristics, classical machine learning, and deep learning. The MePo and MaSh algorithms that have been implemented for this thesis use heuristics and classical machine learning, respectively.

2.5 Lean 4

The Lean 4 programming language is used for writing formal proofs. The language follows the functional programming paradigm and provides features such as dependent types. Besides being used to write formal proofs the language is suitable for general-purpose programming and metaprogramming. In fact, most of the Lean implementation, including its compiler, is written in Lean itself.

2.5.1 Types

Lean relies heavily on its type system to provide the expressivity of the language. First of all, Lean has an infinite hierarchy of types (6). The infinite hierarchy of types allows the user to treat types just as they would treat values of a conventional type, say `Bool`. As a consequence, type constructors are first-class citizens in the language. This type hierarchy is similar to Haskell's higher-kinded types. However, Lean also includes a type universe at the bottom of the type hierarchy containing propositions: `Prop`. Propositions form the basis of formal proving in Lean, which is described in the following section.

In Lean, types can depend on function parameters, known as dependent typing. For example, dependent typing allows users to write the following:

```
theorem a_le_b (a b : ℕ) (h : a < b) : a ≤ b := ...
```

Both the parameter `h` and the return type depend on parameters `a` and `b`. This property allows users to conveniently write formal proofs.

2.5.2 Formal proofs

Propositions are types that live in the `Prop` universe. Because of the unified type hierarchy, in Lean, writing a formal proof is just a special case of regular programming. Consider the following two examples:

```
def any_list : List ℕ := []
def zero_le_one : 0 ≤ 1 := Nat.le.step Nat.le.refl
```

The value of `any_list` must be a term of type `List ℕ`, a type in `Type`, i.e. `Sort 1`. Its definition should be easy to follow for anyone with some familiarity with programming. The value of `zero_le_one` must be a term of type `0 ≤ 1`, a type in `Prop`, i.e. `Sort 0`. In Lean, constructing a value for a type in `Prop` is not fundamentally different to constructing a value for a type in `Type`. The same syntax and semantics apply in both cases.

Lean includes special syntax and tactics to make it easier to find and write proofs. The `by` syntax puts Lean in tactic mode, which allows the use of tactics. Instead of writing a proof by handcrafting an expression, in tactic mode the user proves a goal in steps. Let's revisit the previously discussed example from the Lean tutorial (5):

```
example (a b c : ℝ) : (a * b) * c = b * (a * c) := by
                      -- (a * b) * c = b * (a * c)
  rw [mul_comm a b]     -- (b * a) * c = b * (a * c)
  rw [mul_assoc b a c] -- (a * b) * c = b * (a * c)
```

The current goal after running the tactic is shown at the end of each line. When writing a proof, the current goal is displayed interactively. This interactivity makes proving a conjecture more intuitive.

2.5.3 Compilation

Lean code is compiled into a representation that uses a minimal number of constructs. While a source file can use concepts such as namespaces, free variables and tactic mode, these concepts are not directly represented in the compiled module. Instead, a compiled module consists of only an array which names imported modules and an array of constants. Some higher-level concepts are directly encoded in compiled modules, but these can be considered metadata. Constants correspond to the basic constructs that make up a Lean program, such as axioms, theorems, definitions, and inductive types. Each kind of constant is structured uniformly and expressed in terms of a limited set of types that are understood by Lean's kernel. Most importantly, a constant has at least an `Expr` which describes its type.

During compilation, in a step called *elaboration*, higher level constructs such as tactic mode are converted to the basic constructs. After elaboration the compiler performs optimization and other transformations on the elaborated code. Then, the compiled modules are saved to disk in `.olean` files that can be imported by other modules and interpreted. Optionally, the compiler can ahead-of-time (AOT) compile modules to machine code by emitting C code which is compiled by the system compiler. AOT-compiled modules are useful when better performance relative to interpreted code is required, as is the case for premise selection.

2.5.4 General-purpose programming and metaprogramming

To enable general-purpose programming, Lean includes the `EIO` monad. Through the `EIO` monad it is possible to execute code that has side effects, such as interaction with files. The `EIO` monad is evaluated in two contexts: at the entry point of an executable program and when doing metaprogramming.

Lean lets the user extend the programming language through metaprogramming. Metaprogramming enables the addition of custom syntax. When a syntax extension is used it is evaluated during elaboration of the program, which allows it to access context and modify state. Premise selection uses tactics to provide its functionality. When a Sledgehammer is built for Lean, it will provide its own interface and subsume these tactics.

Tactics can perform side effects through the `EIO` monad and have access to the environment. The environment consists of all elaborated definitions/constants available at the point in the source file where the tactic was invoked. This includes the constants of all modules that were (transitively) imported, but also all constants in the current file (up to the current point in the file). For premise selection, both during cataloging and in preparation of running the algorithm, the environment is relied on to find out which facts are available.

3

Building blocks of premise selection in Lean

To be able to run premise selection algorithms in Lean, it is necessary to have specific infrastructure in place. This consists of being able to extract information from modules, such as facts and the visibility relation. From this information we can build the state which is used to perform premise selection.

3.1 Extracting information from modules

In Lean, a definition, or equivalently a constant, is declared in a module. A module can import other modules to gain access to their definitions. To determine which facts are available and to determine the visibility relation we need to extract the relevant information from modules. This information is gathered through the environment. When the environment is built, all imported modules are loaded, which provides access to the information contained in them.

3.2 Going from constants to facts

A fact corresponds to a constant. These constants can be anything which is representable in Lean, such as theorems, axioms, regular definitions, typeclass projections and constructors. A fact is identified by the name of its corresponding constant and represents the constant using an expression. The representation is used by the premise selection algorithms to predict which facts are relevant. In the case of a theorem, the facts that it uses in its proof constitute its dependencies.

Not all constants defined in a module directly correspond to facts. Therefore, a sensible mapping from constants to facts is needed. Defining an appropriate mapping is important for the performance of the premise selection algorithms.

3.2.1 Theorems and axioms

The most straightforward mapping is from theorems and axioms to facts. The types of their definitions are propositions, which directly map to facts, after all.

Not all definitions that could be considered theorems or axioms are defined using the `theorem` and `axiom` keywords. For example, instances of `Prop` type classes and some types of auxiliary definitions are compiled as regular definitions. Constants whose type is a proposition are therefore considered a theorem or axiom, depending on whether the constant has a proof (its value).

3.2.2 Definitions

For a regular definition, its corresponding fact is the equation of its name and value. For example:

```
def Nat.inc (x : Nat) : Nat := x + 1
```

is converted to a fact with the following representation

```
Nat.inc = fun x => x + 1
```

Note that type information of the constant, such as the return type, is not directly expressed in the fact's representation. When cataloging a fact, type information is inferred from the representation and so is not lost.

3.2.3 Constructors

A constructor's constant does not have a value to which its name can be equated. For example, the `Nat.succ` constructor corresponds to a constant which can be described as:

```
constr Nat.succ : Nat → Nat
```

Note that the `constr` keyword is made up syntax; in the source code, the constant is defined as part of the `Nat` inductive type. The fact corresponding to `Nat.succ` uses its type, `Nat → Nat`, as the representation.

3.2.4 Boring facts and incomplete proofs

Not all facts necessarily contribute much to the results of premise selection. Such facts, termed *boring facts* in Isabelle, are likely tautologies or too technical to be useful. Though they might in principle be useful — tautologies in particular — an ATP is likely to have a built-in understanding of them already. By including boring facts in the results of premise selection performance is degraded because they take up unnecessary space in the results, which might cause other more interesting facts to not be included. This is especially the case for facts that concern only logic. An ATP necessarily has a good grasp of logic; it won't need to be provided facts about logic. Therefore, when a fact's symbols consist only of very general logic symbols (e.g. `True`, $\wedge$, $\neg$) it is not considered during premise selection.

The user can specify an incomplete proof, which will compile, by using the `sorry` keyword. Depending on the desired user experience of Sledgehammer, premise selection should perhaps either entirely include or exclude facts with incomplete proofs or only include them when they are defined in the current module. In the current implementation, facts with an incomplete proof are always included.

3.3 Visibility relation

To determine which facts should be considered by the premise selection algorithm, the visibility relation is used. It ensures that all facts that are visible to a goal are present in the environment and also that the results of premise selection do not lead to a circular proof, i.e. a proof that is defined in terms of itself. For a given fact, only facts that do not (transitively) depend on it should be visible to it.

In this work, the visibility relation is approximated by relying on the directed acyclic graph (DAG) of module imports. In practice the approximate relation is good enough because cycles in the module DAG are disallowed, which means that a more precise visibility relation in terms of the fact dependency DAG would yield unusable results. Inter-module visibility is defined in terms of the module DAG which implicitly encodes visibility of facts: facts in a module A which is depended on by a module B are necessarily visible to facts in B because A cannot depend on B. Intra-module visibility comes for free: at the point where premise selection is invoked, of the facts defined in the same module as the goal only facts visible to the goal are present in the environment.

3.3.1 Respecting the visibility relation

The algorithm to determine which facts are visible to a goal is fairly simple. Consider a goal ϕ in module A. First, inter-module visibility of facts is considered: all facts in modules transitively imported by A are visible to ϕ. Then, intra-module visibility of facts is considered: all facts in the same module as ϕ that are present in the environment are visible to ϕ. The premise selection algorithms use only these facts when producing predictions.

This simple algorithm does have a limitation regarding inter-module visibility. Because only facts in modules (transitively) imported by A are considered, visible facts are not considered when they are defined in modules that *could* be but are not imported by A. This means that the set of facts that is considered by premise selection is smaller than the actual set of visible facts. This limitation can be fixed by considering all modules that do not (transitively) import module A; however, this fix is not implemented due to time constraints. For the evaluation in chapter 6 the limitation does not apply and all visible facts are used.

3.4 Cataloging facts

Facts and their visibility need to be maintained to achieve good runtime performance. Cataloging all facts every time premise selection is invoked wastes a lot of time. The information about facts is kept in the premise selection state. The premise selection algorithms operate on the state to produce suggestions. Whenever new facts and modules are available, the state must be updated to ensure that they are available for premise selection. Besides facts themselves, the premise selection state also contains information related to facts, such as their symbols, features and statistics, which is used by the premise selection algorithms. Exactly what information related to facts is maintained for the premise selection algorithms is explained in the MePo and MaSh chapters (chapters 4 and 5).

3.4.1 Building and updating the state

The premise selection state consists of several arrays and hash maps that store the facts, the visibility relation and information about modules. Its definition is shown in figure 3.1. The most important pattern used is the indexing of modules, symbols, features, and facts. Whenever one of these is added to the state it is associated with an index. The index

```
structure PState where
  moduleIdxs : HashMap Name UInt64
  moduleNames : Array Name
  moduleMtimes : Array IO.FS.SystemTime
  moduleImportss : Array (Array UInt64)
  moduleFactss : Array (Array UInt64)

  factIdxs : HashMap (UInt64 × Name) UInt64
  factNames : Array Name

  symbolIdxs : HashMap Sym UInt64
  symbolValues : Array Sym
  symbolOrders : Array UInt8
  symbolFreqs : Array UInt64

  featureIdxs : HashMap Feature UInt64
  featureValues : Array Feature
  featureFreqs : Array UInt64

  factDepss : Array (Array UInt64)
  factSymbolss : Array (Array UInt64)
  factFeaturess : Array (Array UInt64)
  factAsDepFreqs : Array UInt64
  factAsDepOfFeatureFreqss : Array (HashMap UInt64 UInt64)
```

Figure 3.1: Definition of the premise selection state

is stored in a map to be accessed through the name or value of the datum. Information regarding the datum is stored at this index in the relevant arrays. Whenever a datum is referenced, e.g. which facts a module contains, its index is used instead of its name or value. Though data could be referenced by their names or values instead of an index, by using arrays runtime performance and memory efficiency is increased. However, a consequence is that data cannot be removed from the state; the state will only ever grow.

As can be seen from the `factIdxs` field in the state definition, a fact is not identified by only its name. Rather, a fact is identified by both the module in which it is defined and its name. Because the namespace to which a fact belongs is encoded in the fact's name, namespaces are handled transparently.

Whenever the user runs a premise selection algorithm, a check is performed to see whether the premise selection state is up to date. If this is the case, the premise selection algorithm runs. However, if modules imported by the current module have been modified since the state was last updated, the modified modules are cataloged. Cataloging mod-

ules is done in the background. When this occurs the user is presented with a message indicating that cataloging is in progress and they should attempt to run premise selection when cataloging has finished. Currently, no indication is given as to how far background cataloging has progressed. Also, invoking the background process itself is relatively slow (on the order of seconds) because a fresh environment containing the imported modules needs to be created in case the current environment is freed during cataloging.

Note that only facts from transitively imported modules are cataloged. Lean guarantees that imported modules are compiled and up to date when the source file is elaborated. A scheme that attempts to catalog all source files in a project would be brittle. For example, the source files of two modules could exhibit a cyclic dependency and so would be invalid. Furthermore, the user could be in the process of refactoring and might expect that a certain module should not be used for learning. Therefore, the conservative approach is taken to only catalog from modules that are guaranteed to compile and have been explicitly specified by the user.

3.4.1.1 Aliases

Some constants are aliases of other constants. They have an `@[implemented_by _]` attribute or a value of the form `Expr.const _ _`. Aliases degrade the performance of the premise selection algorithms. When an alias is encountered during cataloging, it is ignored. The premise selection algorithms use unaliased constants.

3.4.1.2 Auxiliary definitions

When the Lean compiler processes a definition, it often creates auxiliary definitions. Auxiliary definitions are created when a subproof is outlined, for example. During cataloging, auxiliary definitions are treated as regular definitions as they provide useful information. However, one type of auxiliary definition, `._cstage`, represents low-level definitions that are the result of compiler transformations, such as specialization and erasure. This type of auxiliary definitions leads to runtime errors during cataloging and likely does not provide information that supplements the original definitions; they are therefore ignored.

3.4.2 Disk persistence

Building the premise selection state from scratch each time it is required would be prohibitively impractical and expensive. Therefore the state needs to be saved to disk.

In Isabelle, premise selection state is stored globally and is shared between different projects. For this work the state is not stored globally, but rather it is stored locally in the build directory of the project invoking premise selection. Storing state locally versus storing it globally has both advantages and disadvantages. With a local state it is possible to maintain a precise vibility relation which has a one-to-one correspondence with that of the libraries used by the project. In a global state this is harder to do, due to the possibility of multiple versions of the same library being cataloged in the state. However, because a global state is likely to learn facts from a greater number of libraries than a local state, the training data has a greater variety. Also, a single global state will use less disk space than all the local states of separate projects combined.

3.4.2.1 On-disk representation

Lean includes JSON (de)serialization functions that could be used to persist state. However, they do not have the appropriate performance characteristics for (de)serializing data of the magnitude of the premise selection state (on the order of 1.5GB for the experiment in chapter 6). Still, disk persistence of the premise selection state is achieved using a fairly simple scheme. By abusing Lean's machinery for saving a compiled module to a `.olean` file, we can serialize arbitrary objects to disk and efficiently deserialize them. The Mathlib library abuses this mechanism in the same way[1], so it is likely that it will remain usable or a suitable replacement is developed.

The Lean project makes no stability guarantees regarding `.olean` files. Therefore, whenever a different toolchain version is used from the one that created the premise selection state the state must be rebuilt. As Lean 4 matures and moves into stable versions, perhaps the `.olean` file format will become stable too. In the meantime, one could conceive of a scheme where breaking changes in the `.olean` are tracked, which could be mostly automated. For example, at the time of writing, the Lean code that deals with `.olean` files, `compact.cpp`, has not been changed since December 2021. An automated system might check whether the `compact.cpp` of the current toolchain differs from that of the toolchain that created the premise selection state.

Similarly, whenever the premise selection state structure changes, the on-disk representation may become incompatible. Therefore, a version number is included in the premise selection state. In the absence of code to migrate the state, whenever an incompatible version is detected the state must be rebuilt.

[1] https://github.com/leanprover-community/mathlib4/blob/b5b6fb2d54f53a9e0af36e33624d5d31920cec21/Mathlib/Util/Pickle.lean

3.4.2.2 Implications of using the `.olean` format

Lean's `.olean` format achieves good deserialization performance by (de)serializing objects in a particular way. Whenever an object is serialized it retains its in-memory structure, i.e. it is `memcpy`'d.[1] Its pointers to other objects are replaced by relative offsets to their location in the file. This serialized representation is exactly the same as the in-memory representation, except for pointers (and object locations). When the `.olean` file is loaded, only the pointers need to be calculated from the offsets to lead to a valid in-memory representation.

The entire `.olean` file is loaded into memory from a single allocation. Objects contained in it therefore do not have separate allocations but are contained in something called a `CompactedRegion`, which is associated with the allocation. An object that belongs to a `CompactedRegion` has different memory management to regular objects. Regular objects are automatically freed whenever they are no longer referenced by any other objects. However, only when a `CompactedRegion` is explicitly freed are its objects freed too, no earlier and no later.

Memory leaks are unacceptable because interactive Lean sessions can last for a long time. Therefore, the `CompactedRegion` and its objects must be freed at some point. Whenever the premise selection state is used, care must be taken that no objects originating from the state escape beyond when the state is freed. If they do, it will inevitably lead to a crash. Luckily, for the task of premise selection only the names of facts and their scores are returned by the algorithm. The fact scores are calculated at runtime so they pose no problem. When the premise selection algorithm finishes, fact names are mapped from objects inside the `CompactedRegion` to objects inside the environment. Now that no objects in the `CompactedRegion` are used anymore it is finally freed.

3.4.3 Non-monotonic changes

As in the implementation of the MaSh paper (4), non-monotonic changes are not considered beyond maintaining the visibility relation. The only monotonic change to the state is the addition of a fact. A non-monotonic change is the removal of a fact or modification of its proof. With the exception of the modification of a proof, all other changes to the set of facts can be considered a combination of the removal and addition of a fact. For example, changing a fact's name is the removal of a fact and addition of a new fact. Similarly, when

[1] https://github.com/leanprover/lean4/blob/8d4dd2311ccfa3de8dbcaba015b52e3ae1ed308d/src/runtime/compact.cpp

a module name is changed all of its facts are removed and the corresponding new facts are added.

As mentioned earlier, candidate facts for premise selection are derived from the visibility relation. Because the visibility relation is maintained even under non-monotonic changes, premise selection will never yield facts that no longer exist.

However, when a fact is removed or its proof is changed, its contribution to the various datastructures that track statistics of facts are not altered. That is, symbol frequencies (used for MePo), and feature and fact dependency frequencies (used for MaSh) do not reflect the change. The implementation in Isabelle does not appear to account for non-monotonic changes in this way either, indicating that it does not lead to significant performance degradation.

3.4.4 Transient changes

During cataloging, only modules transitively imported by the current module are considered. The module that the user is working on is ignored during cataloging. The state will therefore likely not contain all facts up to the point where premise selection was invoked. To include these facts, they are added as transient changes. The state for the current module is cleared and its facts are cataloged (up to the point where premise selection was invoked). This updated state containing the transient changes is not saved to disk.

3.5 MeSh

Isabelle includes a method for combining the results of multiple premise selection algorithms called MeSh. MeSh combines results based on the ranks of their constituent facts.[1] First, each fact in the results of an algorithm is given a score proportional to its rank. The score is weighted according to a particular weight assigned by the algorithm. All the scores of an individual fact are then combined. The combined scores are sorted and cut off at a predetermined number of facts, yielding the MeShed results.

[1] https://github.com/isabelle-prover/mirror-isabelle/blob/8116f95fd508a3d1bb9639c9fd51d20ba3e20aac/src/HOL/Tools/Sledgehammer/sledgehammer_mash.ML#L174

4

Implementing MePo

The MePo algorithm (7) iteratively estimates the relevance of facts based on the syntactic similarity of their signature to that of the goal. If a fact's relevance is high enough it becomes relevant. When a fact becomes relevant so do its symbols, in turn making more facts relevant. The justification for this heuristic is that facts that share symbols with already relevant facts may be useful in using those relevant facts.

4.1 The MePo algorithm

In MePo (7), the set of relevant facts is constructed iteratively. Initially, the set is empty. However, the conjecture's symbols are always considered relevant. In each iteration facts are added to the set based on a measure of their relevance to already relevant facts. When determining the relevance of new facts their syntactic similarity to already relevant facts is considered.

The syntactic similarity of two facts is based on the symbols that they have in common. In generic MePo, symbols are defined as predicate, function or constant symbols. In Lean's MePo, symbols correspond to a constant's name, i.e. the `declName` in `Expr.const declName _`.

Consider the following example:

Example 4.1.1.

$$\forall x : \text{list } \mathbb{N}, \text{map inc } (\text{rev } x) = \text{rev } (\text{map inc } x)$$
$$\forall x : \text{list } \mathbb{N}, \text{empty } x \to \text{rev } x = x$$

In this example the two facts have the following symbols in common: list, $\mathbb{N}$, rev and $=$. Those symbols are used to determine the measure of relevance which decides whether the fact is added to the set of relevant facts.

To determine the relevance of a fact, a *relevance score* is used. For a fact to be considered relevant its relevance score must exceed the *pass mark*. Both the relevance score and pass mark are real numbers between 0 and 1.

The pass mark ensures that each iteration becomes more restrictive in considering a new fact relevant. In a sense, favouring exploitation over exploration over time. Once the pass mark reaches 1 no new facts are considered relevant and the algorithm terminates. For iteration $k > 0$ the pass mark is defined as $p_k = p_{k-1} + (1 - p_{k-1})/c$. The algorithm is configured with a constant pass mark p_0 for the first iteration and constant convergence parameter c.

In the basic algorithm the relevance score is defined as m/n. Here m represents the number of relevant symbols in the fact; n is the total number of symbols in the fact. The second fact in example 4.1.1, $\forall x : \text{list } \mathbb{N}, \text{empty } x \rightarrow \text{rev } x = x$, has a total of five symbols of which four are relevant to the first fact, $\forall x : \text{list } \mathbb{N}, \text{map inc } (\text{rev } x) = \text{rev } (\text{map inc } x)$. Its relevance score is therefore $\frac{4}{5} = 0.8$.

Because many unrelated facts will have symbols in common, such as the equality symbol, the relevance score is extended to take symbol rarity into account. First the weight function $w(f_s)$ is defined which determines how much a symbol contributes to the relevance score. The term f_s is the frequency at which the symbol s occurs in the corpus. Meng and Paulson (7) define $w(f_s) = 1 + 2/\log(f_s + 1)$. The sum $m = \sum_{s \in R} w(f_s)$ over all relevant symbols of a fact is calculated. Then the relevance score becomes $m/(m + i)$, where i is the number of irrelevant symbols in the fact.

Using an imaginary corpus we can give examples of the relevance scores of some facts w.r.t. the following conjecture from example 4.1.1:

$$\forall x : \text{list } \mathbb{N}, \text{map inc } (\text{rev } x) = \text{rev } (\text{map inc } x)$$

The following fact may be useful in proving the conjecture and is indeed given a fairly high relevance score:

$$\forall x : \text{list } \mathbb{N}, \text{empty } x \rightarrow \text{rev } x = x$$

s	$f(s)$	$w(s)$
list	1000	1.29...
$\mathbb{N}$	4000	1.24...
rev	25	1.61...
$=$	8000	1.22...

$$m = \sum_{s \in R} w(s) \approx 5.37$$

$i = 1$ ({empty})

relevance score $= m/(m + i) \approx 0.84$

The following fact is not useful for the proof but does get a high relevance score as we choose $f(\text{inc})$ to be fairly rare. Because the symbol is used so rarely the relevance score of this irrelevant fact is far higher than desirable.

$$\forall n : \mathbb{N}, n < \text{inc } n$$

s	$f(s)$	$w(s)$
$\mathbb{N}$	4000	1.24...
inc	3	2.82...

$m = \sum_{s \in R} w(s) \approx 4.06$

$i = 1 \ (\{<\})$

relevance score $= m/(m+i) \approx 0.80$

However, the following fact which is only related by the common symbol $\mathbb{N}$ has a much lower relevance score of 0.29:

$$\forall n : \mathbb{N}, n < n + 1$$

s	$f(s)$	$w(s)$
$\mathbb{N}$	4000	1.24...

$m = \sum_{s \in R} w(s) \approx 1.24$

$i = 3 \ (\{<, +, 1\})$

relevance score $= m/(m+i) \approx 0.29$

Finally, the following fact has the same form as the fact we're trying to prove except with a quantified variable substituted for the inc symbol. Intuitively, this fact should have a fairly high relevance score. Indeed, because there are no irrelevant symbols its relevance score is 1.

$$\forall (x : \text{list } \alpha)\ (f : (\alpha \to \beta)), \text{map } f\ (\text{rev } x) = \text{rev } (\text{map } f\ x)$$

$i = 0 \ (\emptyset)$

relevance score $= m/m = 1$

4.2 MePo in Isabelle

In Isabelle's MePo implementation the algorithm described above is extended in a number of ways.[1] These extensions do not alter the basic idea of iteratively constructing a set of relevant facts according to syntactic similarity. However, usage of the pass mark and calculation of the relevance score are modified.

The pass mark has a reduced role in Isabelle's implementation. The pass mark is calculated based on an exponential decay parameter. In the original algorithm all facts with a relevance greater than the pass mark are considered relevant. In practice, a large number of facts will have a relevance greater than the pass mark. Isabelle's implementation therefore considers no facts with a relevance below the pass mark relevant and in each iteration only a fixed number of imperfect facts are considered relevant. Facts that have a perfect relevance score of 1 (or very close to it) are always considered relevant.

Calculation of the relevance score is modified in several ways. First of all, various fudge factors are defined that reward or punish a fact and its symbols based on their properties. The fudge factors allow for tuning of the algorithm to improve the performance of its results. The properties that affect the calculation of the score include higher-orderedness of a symbol's type and the *stature* of a fact. A fact's stature indicates whether the fact is, for example, global or local and whether it is, for example, a general or elimination rule.

Secondly, irrelevant symbols are weighted too. When an irrelevant symbol s is very rare, i.e. its frequency is below a certain threshold k, its weight is defined as:

$$w_{ir}(s) = \log(f_s + 1)/\log(k)$$

More common symbols have the following weight:

$$w_{ic}(s) = w(f_s)/w(k)$$

Unlike the weight for relevant symbols, these weights are multiplied with a value depending on the fudge factor σ and the order of the symbol's type o:

$$w_i(s) = \sigma^{o-1} \cdot \begin{cases} w_{ir}(s) & f(s) < k \\ w_{ic}(s) & f(s) \geq k \end{cases}$$

The weight ensures that irrelevant symbols occuring very rarely and very commonly do not affect the relevance score too much, whereas relatively common irrelevant symbols are

[1] https://github.com/isabelle-prover/mirror-isabelle/blob/8116f95fd508a3d1bb9639c9fd51d20ba3e20aac/src/HOL/Tools/Sledgehammer/sledgehammer_mepo.ML

weighted similarly to relevant symbols. Because $\sigma > 1$, irrelevant symbols with a type of higher order affect the relevance score more than those with a type of lower order.

Thirdly, the theory to which a fact belongs is considered as a symbol. The weight of a theory symbol is defined as a constant: 0.5 for a relevant theory symbol and 0.25 when it is irrelevant. Because all facts carry a theory symbol, a fact is only ever considered perfectly relevant (has a relevance score of 1) when it both includes all relevant non-theory symbols and belongs to a theory which is considered relevant. This makes it less likely for a fact to be considered perfectly relevant, which MePo is prone to do, thereby increasing MePo's performance.

Finally, in the fifth iteration, facts that have a relevance score of practically zero (< 0.001) are considered hopeless, which disqualifies them from participation in future iterations. This improves runtime performance but also prevents such hopeless facts from being eventually be considered relevant due to a fluke.

4.3 MePo in Lean

The implementation of MePo in Lean follows the original algorithm and attempts to apply most extensions of the Isabelle implementation. For example, irrelevant symbols carry a weight, a fact's theory is considered a symbol, and hopeless facts are disqualified. However, a fact is not granted a bonus according to its stature. It should be possible to implement an equivalent extension in the Lean implementation, but this is currently not the case. Also, subtleties in the Isabelle implementation may have been missed. This may cause the performance of the results of Lean's MePo to not be as good as Isabelle's MePo.

4.4 Cataloging facts

To run the MePo algorithm, the premise selection state must maintain several pieces of information about facts. Maintaining the premise selection state consists of collecting the symbols present in fact signatures and processing them appropriately.

Collecting which symbols are used in a fact's signature is done by traversing the structure of the corresponding expression. Whenever an `Expr.const _ _` is encountered during traversal, its name is considered a symbol. Symbols that are not yet present in the premise selection state are indexed. The indexes of the symbols that were encountered during traversal of the fact signature are then stored (the `factSymbolss` field in figure 3.1).

To be able to compute the weight of a symbol, the frequency of each symbol in the fact's signature is incremented (the `symbolFreqss` field in figure 3.1).

The names corresponding to the fact's module and namespaces (all namespaces up to the root namespace) are also considered a symbol. To distinguish these symbols from regular symbols a suffix is added to their names.

The higher-orderedness of a symbol's type is stored alongside the symbol's value (the `symbolOrders` field in figure 3.1).

5

Implementing MaSh

Blanchette et al. (4) apply classical machine learning techniques to the task of premise selection, dubbed MaSh. Two premise selection algorithms are presented: a Naive Bayes classifier and a k-Nearest Neighbors classifier. Both algorithms are agnostic to the proof assistant being used and the Isabelle implementation can therefore be straightforwardly ported to Lean. Aspects that are specific to the proof assistant are limited to the learning phase and are abstracted away through features and generic datastructures.

5.1 Features

In MaSh, features are strings that represent information about a fact. A fact tends to have many features. Feature are derived from the subterms of the fact's signature.

One way to derive a feature from a subterm is to use the name of the constant at the root of the subterm. This method would be equivalent to how symbols are extracted for MePo. However, because a feature is a string it is not limited to only using a constant's name as its value, we can express more information in a feature. When deriving a feature from a (sub)term, its subterms are considered up to a depth of 2 (including the root). For example, the term `1 + 2` yields the feature `+(1, 2)` and `(1 + 2) + 3` yields `+(+, 3)`. Because subterms are considered at depth 1 as well, both terms also yield the feature `+`. When a variable is encountered in a term it yields a feature corresponding to its type. For example, the term `a + 1` yields the feature `+(Nat, 1)`.

The amount of information expressed by a fact's features can be increased by considering more aspects of subterms. To that end, the type of a subterm is considered too, up to a depth of 1. This leads to the examples above also yielding the feature `Nat`.

Additionally, the parents of a structure or typeclass are included as features. When a term of type A which extends B is encountered it yields both features `A` and `B`. By considering the parents of a type performance is increased when facts defined on a parent need to be selected.

Finally, the theory (module and namespaces in Lean) and whether a fact is local are considered features.

The MaSh paper shows the following example of the derivation of features:

$$\text{transpose (map (map } f\text{) } xss\text{) = map (map } f\text{) (transpose } xss\text{)}$$

map	map(list_list)	fun
map(fun)	map(map,list_list)	list
map(map)	transpose	list_list
map(transpose)	transpose(map)	List
map(map,transpose)	transpose(list_list)	

5.2 Naive Bayes

MaSh's Naive Bayes classifier is based on the principles of a generic Naive Bayes classifier. The premise selection Naive Bayes classifier learns from proofs to be able to determine an approximate probability of a fact with some particular features being used in the proof of a goal with some particular features.

5.2.1 The Naive Bayes algorithm

The probability of a fact ϕ being used in the proof of a fact γ is as follows (4):

$$\begin{aligned}
&P(\phi \text{ is used in } \psi\text{'s proof}) \\
&\cdot \prod_{f \in F(\gamma) \cap \bar{F}(\phi)} P(\psi \text{ has feature } f \mid \phi \text{ is used in } \psi\text{'s proof}) \\
&\cdot \prod_{f \in F(\gamma) - \bar{F}(\phi)} P(\psi \text{ has feature } f \mid \phi \text{ is not used in } \psi\text{'s proof}) \\
&\cdot \prod_{f \in \bar{F}(\phi) - F(\gamma)} P(\psi \text{ does not have feature } f \mid \phi \text{ is used in } \psi\text{'s proof})
\end{aligned}$$

The probability operates on features of ϕ and γ. The expression $F(\gamma)$ denotes the set of features of the fact γ. The expression $\bar{F}(\phi)$ denotes the set of extended features of the fact ϕ. The set of extended features of ϕ is defined as the union of ϕ's features and the features of all facts that use ϕ in their proof.

The probability is rewritten to (4):

$$
\begin{aligned}
&\sigma_1 \ln t(\phi) \\
&+ \sum_{f \in F(\gamma) \cap \bar{F}(\phi)} w(f) \ln \frac{\sigma_2 s(\phi, f)}{t(\phi)} \\
&+ \sigma_3 \sum_{f \in \bar{F}(\phi) - F(\gamma)} w(f) \ln \left(1 - \frac{s(\phi, f) - 1}{t(\phi)}\right) \\
&+ \sigma_4 \sum_{f \in F(\gamma) - \bar{F}(\phi)} w(f)
\end{aligned}
$$

The individual probabilities are expressed in terms of s and t. The expression $s(\phi, f)$ denotes the frequency of the fact ϕ being used in the proof of a fact which has feature f. The expression $t(\phi)$ denotes the frequency of the fact ϕ being used in a proof. The values of $s(\phi, f)$ and $t(\phi)$ are determined by learning from the proof of facts. The rewritten expression is not a direct translation of the original probability but adapts it to be more suitable for practical use.

5.2.2 Naive Bayes in Isabelle

The Isabelle implementation of the Naive Bayes premise selection algorithm is a straightforward implementation of the probability expression described above.[1] The implementation does not need to separately keep track of the extended features of a fact, $\bar{F}(\phi)$, during learning. Instead, the extended features are derived from the datastructure used for the term $s(\phi, f)$. The expression $s(\phi)$ is a mapping from features of the facts that use ϕ in their proof to the corresponding frequency. The set of keys in this mapping happens to be the same as the extended features of ϕ.

The Isabelle implementation of the Naive Bayes premise selection algorithm makes a distinction between features from regular, chained, and extra facts. A fact becomes chained when it is referenced in a proof using keywords such as `from` and `hence` (4). Extra facts are those facts that are defined in the same theory as the goal. Each feature is associated with an additional weight which is multiplied by the IDF feature weight. For regular features this additional weight is 1.0, causing them to be treated normally. Chained and extra features have an additional weight determined by a fudge factor which causes them to be weighted less heavily.

[1] https://github.com/isabelle-prover/mirror-isabelle/blob/8116f95fd508a3d1bb9639c9fd51d20ba3e20aac/src/HOL/Tools/Sledgehammer/sledgehammer_mash.ML

5.2.3 Naive Bayes in Lean

The Lean implementation of Naive Bayes is a direct port of the Isabelle implementation. However, in the Lean implementation no distinction is made between features from regular, chained, and extra facts. For chained facts, the keywords that would correspond to Isabelle's `from` and `hence` are not preserved in constants, which are used to extract information about a fact's proof. Extra facts can be determined, however this was not been done because experimental results indicated similar performance to Isabelle's MaSh.

The fudge factors σ are the same as those in the Isabelle implementation and MaSh paper. This is under the assumption that the characteristics for which the fudge factors are appropriate do not differ significantly between the corpus of facts in Isabelle and Lean.

5.3 k-Nearest Neighbors

MaSh's k-Nearest Neighbors premise selection is based on generic k-Nearest Neighbors classification (4). However, the implementation extends the basic classifier to make it more suitable for use in premise selection. One of the extensions causes dependencies of a fact to be considered relevant for a proof. The other extension uses a dynamic value for k instead of the conventional fixed k. Like Naive Bayes, the k-NN classifier learns from proofs to be able to determine an approximate probability of a fact with some particular features being used in the proof of a goal with some particular features.

5.3.1 The k-Nearest Neighbors algorithm

The k-Nearest Neighbors algorithm selects the facts that are nearest (according to some metric) to the goal. How near a fact is considered to be to the goal, its distance, is called the nearness or overlap metric. Using the nearness, the relevance of the facts is calculated.

For a given fact ϕ and goal γ their distance is defined as (4):

$$n(\phi, \gamma) = \sum_{f \in F(\phi) \cap F(\gamma)} w(f)^{\tau_1}$$

where $w(f)$ is the feature weight and τ_1 is a fudge factor. The distances of all facts to the goal are calculated, after which their relevance is calculated.

In the standard k-Nearest Neighbors algorithm the distance of an object is the output of the algorithm. However, MaSh's k-Nearest Neighbors algorithm also takes the dependencies of a fact into account when calculating the relevance of facts. The estimated relevance

of a fact ϕ is as follows (4):

$$\left(\tau_2 \sum_{\chi \in N, \phi \in \Pi(\chi)} \frac{n(\chi, \gamma)}{|\Pi(\chi)|}\right) + \begin{cases} n(\phi, \gamma) & \text{if } \phi \in N \\ 0 & \text{otherwise} \end{cases}$$

where N is the set of the k-nearest neighbors for a given k, and $\Pi(\chi)$ are the dependencies of the fact χ. Note that the relevance score of a fact ϕ is only contributed to by facts in the k-nearest neighbors that have ϕ as a dependency or when ϕ is a k-nearest neighbor itself. The right part of the expression is the standard k-nearest neighbors, and the left part is the extension which takes into account dependencies of the nearest neighbors.

Because the k-nearest neighbors contribute to the relevance of their dependencies too, a fixed value of k is not suitable. A k that is too small might cause too few facts to be considered relevant, whereas a k that is too large leads to degraded results. The algorithm starts with $k = 1$ and keeps incrementing it until enough facts have been found.

5.3.2 k-Nearest Neighbors in Isabelle

The Isabelle implementation of MaSh's k-Nearest Neighbors mostly follows the algorithm described above. Calculation of the nearness of a fact to the goal is the same. However, the heuristic that determines at which k the algorithm should terminate causes the calculation of the relevance to somewhat deviate from the definition above.

Initially, all facts have a relevance of zero. The algorithm iterates over k, starting with $k = 1$. In each iteration the k-th nearest neighbor ϕ and its dependencies are considered. When considering ϕ, its distance is added to its relevance score. When considering a dependency of ϕ, the value $\tau_2 \frac{n(\phi,\gamma)}{|\Pi(\phi)|}$ is added to its relevance score. So far the algorithm follows the definition from 5.3.1. However, these values are added to the fact's relevance score only if the fact already has a relevance score greater than zero, i.e. it has already been recommended at least once. If this is not the case then the current *age* is added to the relevance score instead and the number of facts that have been recommended at least once is increased. The age is initialized at 500000000 and is decreased by 10000 after each iteration following the initial iteration. The algorithm terminates when k exceeds the total number of facts or when the number of facts that have been recommended at least once reaches the number of facts that the algorithm should return.

The use of the age causes facts to receive an initial relevance when they are named by the k-th nearest neighbor, which diminishes as k grows. The distance of a fact to the goal only contributes to the relevance score once the fact and its dependencies have already been recommended. Being recommended early, either because the fact is fairly near to the

goal or it is a dependency of such a fact, and then being recommended often, preferably by facts that are near to the goal, determines how relevant a fact is.

5.3.3 k-Nearest Neighbors in Lean

The Lean implementation is a direct port of the Isabelle implementation. However, as with Naive Bayes, in the Lean implementation no distinction is made between features from regular, chained, and extra facts.

Again, the fudge factors τ are the same as those in the Isabelle implementation and MaSh paper, under the assumption that the characteristics for which the fudge factors are appropriate do not differ significantly between the corpus of facts in Isabelle and Lean.

Profiling has shown that most of the time spent in the algorithm was in the function that sorts facts according to their distances to the goal. As a runtime performance optimization, the number of facts to be sorted is decreased by only sorting those facts that have features in common with the goal. This leads to a modest runtime performance improvement over sorting all facts. Storing the distances of facts in a sorted map lead to performance degradation.

5.4 Cataloging facts

The following information used by the MaSh algorithms is stored in the premise selection state (for its definition see figure 3.1):

- A fact's features are stored in the `factFeaturess` field.
- The IDF of a feature, which is used to calculate its weight, is stored in the `featureFreqss` field.
- The $s(\phi, f)$ and $t(\phi)$ values are stored in the `factAsDepOfFeatureFreqss` and `factAsDepFreqs` fields, respectively.
- Dependencies of a fact are stored in the `factDepss` field.

6

Evaluation

To evaluate the performance of the premise selection algorithms in Lean an experiment was performed. The experiment follows the same principle and produces the same metrics as one of the experiments performed by Blanchette et al. (4), allowing us to compare the results of the two experiments.

6.1 Experiment design

The experiment consists of performing premise selection on the goals in the leaf modules of the Mathlib library and deriving metrics by comparing the results of premise selection with the actual proof of the goals. The premise selection algorithms are configured to return 1024 results. In the case of MePo, the algorithm may return more than 1024 results.

The experiment in the MaSh paper tests the results of premise selection on multiple formalizations. However, in this experiment only the Mathlib library is tested. The Lean community website has a list of formalizations that use Lean which shows many formalizations in Lean 3, but none in Lean 4. Lean 3 is incompatible with Lean 4, and formalizations for it can not be used to test the Lean 4 implementation of premise selection. Therefore, this experiment only tests the work-in-progress Mathlib library available for Lean 4.

Because of the size of the Mathlib library, running premise selection on all goals in the Mathlib library is prohibitively expensive in terms of runtime. Instead, premise selection is performed for all goals in the leaf modules in the library. The set of leaf modules consists of the modules that are not (transitively) imported by any other modules in the library, with the exception of modules that consist only of imports. Because the leaf modules are not imported by any other module containing definitions, all facts in the library are visible

to the goal (with the exception of those necessarily not visible to the goal in the module being tested).

For a given goal, the algorithms are trained only on the facts that are visible to it.

6.1.1 Metrics

The counterpart of this experiment done by Blanchette et al. evaluates the results of premise selection using a number of metrics: full recall, area under ROC curve (AUC), and k-coverage (4). These metrics are defined in terms of a number of values. The set of facts that make up the proof of a goal is denoted by Π. The set of results of the premise selection algorithm is denoted by $\Phi = \phi_1, ..., \phi_n$. The rank of a fact $\phi_i \in \Phi$ is taken to be $\text{rank}(\phi_i) = i$, whereas the rank of a fact $\phi \notin \Phi$ is taken to be $\text{rank}(\phi) = n + 1$. Facts with a lower rank are expected to have more relevance than facts with a higher rank.

Full recall gives an indication of how well an algorithm performs by indicating at which position in its results all facts in Π are present. Of course, the results will not return all necessary facts for all goals, in which case the value $n + 1$ is used.

The AUC metric indicates the probability that a fact returned by premise selection which is a dependency of the goal has a better rank than one which is not a dependency. AUC is defined as (4)

$$\frac{|\{(\phi, \chi) \in \Pi \times (\Phi - \Pi) \mid \text{rank}(\phi) < \text{rank}(\chi)\}|}{|\Pi| \cdot |\Phi - \Pi|}$$

k-coverage indicates which fraction of required facts is present in the results of the premise selection algorithm, cut off at k results. k-coverage is defined as (4)

$$\frac{|\{\phi_1, ..., \phi_k\} \cap \Pi|}{\min\{k, |\Pi|\}}$$

6.2 Results

The statistics of the formalizations used in both experiments are found in table 6.1. The number of goals in the leaf modules of Mathlib exceeds that of any of the formalizations used in the MaSh experiment. The average number of features per fact is similar, but the total number of facts and total number of features are significantly higher than in the Isabelle formalizations. Note that the total number of facts and features includes those of a formalization's dependencies.

The average number of dependencies per goal is far higher than in the Isabelle formalizations. In Isabelle, each step of a tactic proof results in a separate goal, meaning that

Formalization	Number of goals	Avg. deps. per goal	Avg. feats. per fact	Total number of facts ('000)	Total number of feats. ('000)
MaSh paper (4)					
Auth	739	5.3	42	15	17
IsaFoR	571	7.4	30	65	139
Jinja	749	5.8	33	17	27
List	863	6.7	14	13	15
Nominal2	432	5.9	19	15	19
Probability	1542	7.2	31	24	32
Lean PS					
Mathlib (leaf modules)	8996	29.3	28	319	994

Table 6.1: Statistics of the formalizations used in experiments

the dependencies of the proof are divided between the separate goals. In this experiment only the proposition is taken to be the goal, leading to the higher average number of dependencies per goal. Because it is more difficult to correctly predict more facts, results for the Isabelle and Lean formalizations are not directly comparable; however, they can provide some insight into the performance of the premise selection algorithms.

Formalization	MePo	MaSh		MeSh	
		NB	k-NN	NB	k-NN
MaSh paper (4)					
Auth	647	104	143	**96**	112
IsaFoR	1332	**513**	604	517	570
Jinja	839	244	306	**229**	256
List	1083	**234**	263	259	271
Nominal2	1045	**220**	276	229	264
Probability	1324	434	422	**393**	395
Lean PS					
Mathlib (leaf modules)	1870	542	**432**	626	542

Table 6.2: Average full recall

Table 6.2 shows the average full recall of the premise selection algorithms. According to this metric performance of the MaSh premise selection algorithms in Lean is comparable to those in Isabelle, whereas MePo and MeSh perform worse. Perhaps surprisingly, the MaSh k-NN algorithm outperforms the other algorithms on full recall, which is not the

case for any of the Isabelle formalizations that were tested.

Formalization	MePo	MaSh		MeSh	
		NB	k-NN	NB	k-NN
MaSh paper (4)					
Auth	86.8	98.1	97.3	**98.4**	98.0
IsaFoR	64.0	**93.7**	92.1	92.9	92.1
Jinja	77.6	96.9	95.7	**97.0**	96.5
List	71.6	**96.8**	96.5	96.5	96.4
Nominal2	72.1	**97.0**	96.2	96.9	96.4
Probability	65.4	94.1	**94.8**	**94.8**	**94.8**
Lean PS					
Mathlib (leaf modules)	23.3	91.7	**93.5**	86.7	88.8

Table 6.3: Average AUC (%)

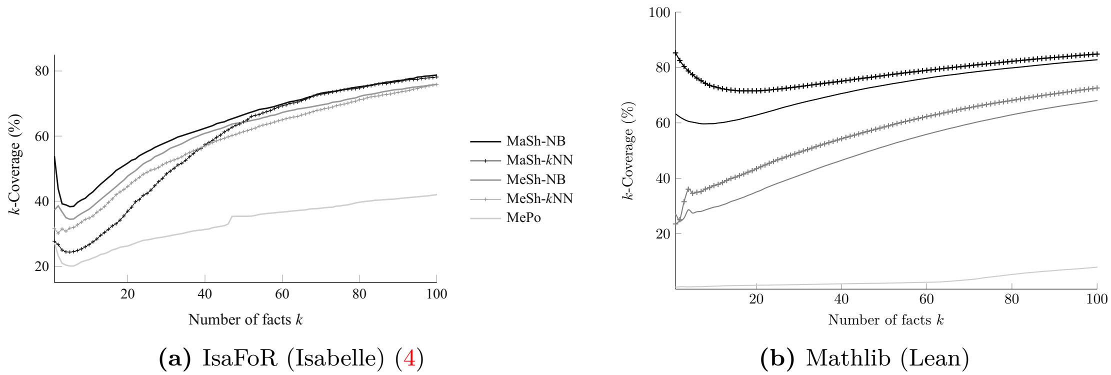
Table 6.3 shows the average AUC of the premise selection algorithms. The AUC of the premise selection algorithms in Lean is notably worse than in Isabelle. With the exception of MaSh k-NN, all algorithms have worse performance with regards to AUC in Lean than in Isabelle. Especially MePo has dramatically bad performance in Lean compared to Isabelle. Part of the reason why comparable AUC is not achieved may be the significantly higher average number of dependencies per goal and total number of facts for Mathlib. Because MeSh determines the relevance of facts based on their rank, MePo's poor AUC will contribute to relatively worse performance for MeSh-NB and MeSh-KNN.

Formalization	MePo	MaSh		MeSh	
		NB	k-NN	NB	k-NN
MaSh paper (4)					
Auth	267	41	58	**35**	42
IsaFoR	735	**132**	164	149	165
Jinja	439	66	90	**63**	73
List	578	**67**	75	74	77
Nominal2	571	**63**	80	67	77
Probability	708	123	**109**	**109**	**109**
Lean PS					
Mathlib (leaf modules)	1703	132	**126**	204	197

Table 6.4: Average rank of necessary dependencies

The average rank of necessary dependencies is shown in table 6.4. Here, the algorithms

in Lean, except for MePo, have performance that is comparable to their performance in Isabelle.



(a) IsaFoR (Isabelle) (4) **(b)** Mathlib (Lean)

Figure 6.1: Average k-coverage

Figure 6.1 shows the average k-coverage for the IsaFoR (Isabelle) and Mathlib (Lean) formalizations. Here, MePo again shows significantly worse performance in Lean than in Isabelle. The MaSh and MeSh algorithms show comparable performance, albeit showing a flatter curve in Lean. The flatter curve may be a result of the larger corpus of facts on which the algorithms were trained, which improves results for a smaller k.

These results show that both the MaSh and MeSh algorithms have similar performance in Lean and Isabelle, while MePo performs significantly worse in Lean.

7

Discussion

The work presented in this thesis is a Lean implementation of the premise selection algorithms found in Isabelle. What the experiment measures is somewhat restricted and not representative of real-word performance. The results of the experiment show that MePo performs worse than what was measured for the Isabelle formalizations. The MaSh premise selection algorithms have comparable performance, however.

7.1 Limitations of the experiment

The experiment measures the performance for theorems in the leaf modules in Mathlib. For the MaSh algorithms this means that the proofs of nearly all facts have been learned. The symbol and feature frequencies over nearly all facts are also known. For theorems that have a smaller set of visible facts, the algorithms have not had the opportunity to learn as much, which might degrade their performance. However, with a smaller set of visible facts the algorithms also have fewer facts that they might predict, which might improve their performance. This balance influences the overall performance of the algorithms. Because testing all theorems in Mathlib takes too long, the experiment does not take it into account. However, the experiment is indicative of the kind of performance an end-user of Mathlib might expect.

Secondly, only the Mathlib formalization was used for the experiment. Other formalizations may have different characteristics which impact performance. Additional formalizations will therefore need to be tested when they become available.

Finally, the experiment on the Isabelle formalizations tests the subgoals of a proof, whereas the experiment in this work considers only the theorem as the goal, leading to a higher average number of necessary dependencies. Also, a given theorem can have more

than one possible proof. A premise selection algorithm might suggest a set of facts that can be used to successfully construct a proof, but which differs from the proof in the formalization. The metrics used by the experiment cannot account for this possibility and will show poor performance. So while the results give an impression of the performance of the algorithms, only when Sledgehammer is implemented in Lean can the actual performance of the algorithms as used in practice be measured. At that point an experiment that replicates the second experiment of the MaSh paper can be done, to be able to compare the real-world performance of the algorithms and their Isabelle counterparts.

7.2 MePo performance

MePo's poor performance compared to that of the experiment performed by Blanchette et al. is likely because of differences in the experiments. In the experiment in this work a goal directly corresponds to a theorem, whereas for the Isabelle formalizations goals are derived from the intermediate steps in a theorem's proof. This means that the number of dependencies that need to be found by MePo is a lot higher. Having to correctly predict a larger number of facts certainly is more challenging. While the MaSh algorithms manage to do this, MePo appears to have a harder time in succeeding at the task. A more realistic experiment using Sledgehammer will provide more insight into MePo's performance in Lean.

7.3 MaSh performance

Unlike MePo, the MaSh premise selection algorithms are able to cope with the higher number of necessary dependencies. The information that is gained from learning from the proofs of previous facts is sufficient to tackle the higher number of dependencies. Of course, the possibility that the trained MaSh algorithms are overfitted to Mathlib, leading to their good results, cannot be discounted based on this experiment alone. An experiment on formalizations that depend on Mathlib will give an indication of how overfitting has influenced this experiment.

8

Related work

Beside the MePo and MaSh algorithms described previously, several other premise selection algorithms have been developed. Among these algorithms are those based on heuristics (like MePo), classical machine learning (like MaSh), and deep learning.

8.1 SInE

The SInE algorithm (8), like MePo, constructs a set of relevant facts iteratively based on syntactic similarity through facts' symbols. The algorithm uses the concept of a *trigger relation* that determines which facts are *triggered* (become relevant) by a relevant symbol. When a fact becomes relevant so do its symbols. The trigger relation which specifies the trigger symbols of a fact is defined as follows (8):

> Let us denote by $occ(s)$ the number of axioms in which the symbol s appears. Then we define the relation *trigger* as follows: $trigger(s, A)$ iff for all symbols s' occurring in A we have $occ(s) \leq occ(s')$. In other words, an axiom is only triggered by the least common symbols occurring in it.

Hoder and Voronkov (8) define two variations of the trigger relations to allow for configurable tolerance and generality. By default these variations are not used.

8.2 Premise selection using a support vector machine

Alama et al. (9) present a kernel-based support vector machine for premise selection.

The specifics of the task of premise selection have been abstracted away by generating features from facts and defining an expected loss function which the support vector machine can operate on. The expected loss function is expressed in terms of a proof

matrix and allows the classifier to learn from proofs. The proof matrix is created at the learning stage and shows whether a fact was used in the proof of a conjecture. A general kernel-based support vector machine can operate on these structures.

The algorithm presented by Alama et al. is non-linear. They claim that the non-linearity leads to better performance of the algorithm when it is used on a different dataset than the one used during training (9).

Experimental results show that the kernel-based support vector machine by Alama et al. outperform both SInE and SNoW's Naive Bayes classifier (9). In one of the experiments, the support vector machine was able to solve 45.1% more problems than SInE and 10.1% more problems than SNoW (9).

8.3 DeepMath

DeepMath employs neural networks to perform premise selection (10). Instead of directly encoding heuristics in the classifier, certain heuristics emerge in the model during training.

The neural network consists of two separate embedder networks that interpret the conjecture and an axiom, respectively. The outputs of the embeddings are concatenated and passed through a logistic layer which outputs the relevance of the axiom. Unlike the previously discussed premise selection algorithms, facts encoded in the formal language are not deconstructed according to their features, but are rather passed to the neural network verbatim.

Experimental results show that DeepMath outperforms k-NN by a few percentage points (10). However, Irving et al. note that "Despite these encouraging results, our models are relatively shallow networks with inherent limitations to representational power and are incapable of capturing high level properties of mathematical statements" (10).

9

Conclusion

For this thesis I have ported Isabelle's premise selection algorithms to Lean 4 and evaluated the performance of the ports.

Concepts used for premise selection in Isabelle, such as the visibility relation and feature extraction, are applicable to Lean without the need for adaptation. Of course, the implementation of these concepts differs between Isabelle and Lean. Due to their relative simplicity, the Lean implementations of the MaSh algorithms closely match their Isabelle counterparts with the exception of a single extension used by Isabelle. The implementation of MePo in Lean follows the spirit of its Isabelle counterpart and carries over many of the extensions used by Isabelle's MePo.

The performance of the MaSh algorithms is comparable for Lean, as measured by testing on Mathlib, and Isabelle, as was measured by Blanchette et al. (4) by testing a number of formalizations. The metrics used to measure performance of the algorithms are all very similar for both experiments. Only one metric shows slightly worse performance by a few percentage points.

However, the performance of MePo is dramatically worse in Lean than in Isabelle. None of the metrics for Mathlib come close to what is achieved for the Isabelle formalizations. This is most likely the result of differences between the Isabelle and Lean experiments regarding the definition of a goal.

9.1 Future work

To assess the real-world performance of the premise selection algorithms in Lean, an experiment should be performed using a Sledgehammer for Lean.

When more Lean 4 formalizations become available the performance of the premise selection algorithms on the formalizations could be performed. This additional data would provide more insight into how well the algorithms perform compared to Isabelle.

Building on the premise selection algorithms, a Sledgehammer for Lean can be implemented. The Sledgehammer will enable users to leverage ATPs to automatically prove theorems in Lean. As an alternative to using ATPs, a Sledgehammer-like mechanism that leverages proof search tactics such as Aesop (11) could be implemented.

References

[1] Tobias Nipkow, Lawrence Paulson, Makarius Wenzel, Gerwin Klein, Florian Haftmann, Tjark Weber, Johannes Hölzl, and the Isabelle community. **Isabelle Home Page**. 1
https://isabelle.in.tum.de.

[2] Leo de Moura, Sebastian Ullrich, Gabriel Ebner, and the Lean community. **Lean Home Page**. 1
https://leanprover.github.io.

[3] Jasmin Christian Blanchette, Lukas Bulwahn, and Tobias Nipkow. **Automatic Proof and Disproof in Isabelle/HOL**. In *Frontiers of Combining Systems: 8th International Symposium, FroCoS 2011, Saarbrücken, Germany, October 5-7, 2011. Proceedings 8*, pages 12–27. Springer, 2011. 2, 5

[4] Jasmin Christian Blanchette, David Greenaway, Cezary Kaliszyk, Daniel Kühlwein, and Josef Urban. **A Learning-based Fact Selector for Isabelle/HOL**. *Journal of Automated Reasoning*, **57**:219–244, 2016. 2, 16, 24, 25, 26, 27, 28, 30, 31, 32, 33, 34, 39

[5] The Lean community. **Lean tutorials: equality rewriting**. 4, 6
https://github.com/leanprover-community/tutorials/blob/88cb01f76462a0331a69bc53cbf302176edc1fc0/src/solutions/01_equality_rewriting.lean#L29.

[6] Jeremy Avigad, Leonardo de Moura, Soonho Kong, Sebastian Ullrich, and the Lean community. **Theorem Proving in Lean 4: Dependent Type Theory**. 6
https://leanprover.github.io/theorem_proving_in_lean4/dependent_type_theory.html.

[7] Jia Meng and Lawrence C Paulson. **Lightweight Relevance Filtering for Machine-generated Resolution Problems**. *Journal of Applied Logic*, **7**(1):41–57, 2009. 18, 19

[8] Kryštof Hoder and Andrei Voronkov. **Sine Qua Non for Large Theory Reasoning**. In *International Conference on Automated Deduction*, pages 299–314. Springer, 2011. 37

[9] Jesse Alama, Tom Heskes, Daniel Kühlwein, Evgeni Tsivtsivadze, and Josef Urban. **Premise Selection for Mathematics by Corpus Analysis and Kernel Methods**. *Journal of Automated Reasoning*, **52**(2):191–213, 2014. 37, 38

[10] Geoffrey Irving, Christian Szegedy, Alexander A. Alemi, Niklas Eén, François Chollet, and Josef Urban. **DeepMath - Deep Sequence Models for Premise Selection**. In Daniel D. Lee, Masashi Sugiyama, Ulrike von Luxburg, Isabelle Guyon, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 29: Annual Conference on Neural Information Processing Systems 2016, December 5-10, 2016, Barcelona, Spain*, pages 2235–2243, 2016. 38

[11] Jannis Limperg and Asta Halkjær From. **Aesop: White-Box Best-First Proof Search for Lean**. In *Proceedings of the 12th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 253–266, 2023. 40