Faster Smarter Proof by Induction in Isabelle/HOL

Yutaka Nagashima
Yale-NUS College, National University of Singapore
University of Innsbruck
Czech Institute of Informatics, Robotics, and Cybernetics, Czech Technical University in Prague
yutaka@yale-nus.edu.sg

Abstract

We present sem_ind, a recommendation tool for proof by induction in Isabelle/HOL. Given an inductive problem, sem_ind produces candidate arguments for proof by induction, and selects promising ones using heuristics. Our evaluation based on 1,095 inductive problems from 22 source files shows that sem_ind improves the accuracy of recommendation from 20.1% to 38.2% for the most promising candidates within 5.0 seconds of timeout compared to its predecessor while decreasing the median value of execution time from 2.79 seconds to 1.06 seconds.

1 Introduction

As our society grew reliant on software systems, the trustworthiness of such systems became essential. One approach to develop trustworthy systems is complete formal verification using proof assistants. In a complete formal verification, we specify the desired properties of our systems and prove that our implementations are correct in terms of the specifications using software tools, called proof assistants.

In many verification projects, proof by induction plays a critical role. To facilitate proof by induction, modern proof assistants offer sub-tools, called tactics. For example, Isabelle [Nipkow et al., 2002] comes with the induct tactic. Using the induct tactic, human proof authors can apply proof by induction simply by passing appropriate arguments instead of manually developing induction principles. When choosing such arguments, proof engineers have to answer the following three questions:

• On which terms do they apply induction?
• Which variables do they pass to the arbitrary field to generalise them?
• Which induction rule do they pass to the rule field?

For example, Program 1 defines the append function (@) and two reverse functions (rev1 and rev2) and presents two ways to prove their equivalence by applying the induct tactic. Note that [], #, and [x] represent the empty list, the list constructor, and the syntactic sugar for x # [], respectively.

Program 1 Equivalence of two reverse functions
@ :: α list ⇒ α list ⇒ α list
[] @ ys = ys
| (x # xs) @ ys = x # (xs @ ys)

rev1 :: α list ⇒ α list
rev1 [] = []
| rev1 (x # xs) = rev1 xs @ [x]

rev2 :: α list ⇒ α list ⇒ α list
rev2 [] ys = ys
| rev2 (x # xs) ys = rev2 xs (x # ys)

theorem rev2 xs ys = rev1 xs @ ys
apply(induct xs ys rule: rev2.induct)
by auto

Theorem rev2 xs ys = rev1 xs @ ys
apply(induct xs arbitrary: ys) by auto

The first proof script applies computation induction by passing rev2.induct to the rule field. rev2.induct is a customised induction rule, which Isabelle automatically derives from the definition of rev2. The subsequent application of auto discharges all sub-goals produced by this induction.

The second proof script applies structural induction on xs while generalising ys. This application of structural induction results in the following base case and induction step:

base case: rev2 [] ys = rev1 [] @ ys
induction step:
(∀ys. rev2 xs ys = rev1 xs @ ys) --->
rev2 (a # xs) ys = rev1 (a # xs) @ ys

where ∀ and ---> represent the universal quantifier and implication, respectively. Using the associative property of @, the subsequent application of auto firstly transformed the induction step to the following intermediate goal internally:

(∀ys. rev2 xs ys = rev1 xs @ ys) --->
rev2 xs (a # ys) = rev1 xs @ (a # ys)

Since ys was generalised in the induction hypothesis, auto proved rev2 xs (a # ys) = rev1 (xs @ (a # ys))

1981
by considering it as a concrete case of the induction hypothesis. If we remove \( ys \) from the arbitrary field, the subsequent application of \( \text{auto} \) leaves the induction step as follows:

\[
\text{rev2 \ xs \ ys = rev1 \ xs \ @ \ ys} \rightarrow \\
\text{rev2 \ xs \ (a \ # \ ys) = rev1 \ xs \ @ \ (a \ # \ ys)}
\]

In other words, \( \text{auto} \) cannot make use of the induction hypothesis since the conclusion of induction step share the same \( ys \). Experienced human researchers can judge that this application of the \textit{induct} tactic was not appropriate. However, it is also true that this induction step is still provable. For this reason, counter-example finders, such as Nitpick [Blanchette and Nipkow, 2010] and Quickcheck [Bulwahn, 2012], cannot detect that this \textit{induct} tactic without generalisation is not appropriate for this problem. This is why engineers still have to carefully examine inductive problems to answer the aforementioned three questions when using the \textit{induct} tactic.

This issue is not specific to Isabelle: other proof assistants, such as Coq [The Coq development team, 2021], HOL4 [Slind and Norrish, 2008], and HOL Light [Harrison, 1996], offer similar tactics for inductive theorem proving, and it is human engineers who have to specify the arguments for such tactics. This issue is not trivial either: in a summary paper from 2005, Gramlich listed generalisation as one of the main problems and challenges of inductive theorem proving while predicting that substantial progress in inductive theorem proving will take time due to the enormous problems and the inherent difficulty of inductive theorem proving [Gramlich, 2005].

Previously, we built \texttt{smart.induct}, which suggests arguments of the \textit{induct} tactic in Isabelle/HOL. Our evaluation showed that \texttt{smart.induct} predicts on which variables Isabelle experts apply the \textit{induct} tactic for some inductive problems. Unfortunately, \texttt{smart.induct} has the following limitations:

L1. It tends to take too long to produce recommendations.
L2. It cannot recommend induction on compound terms or induction on multiple occurrences of the same variable.
L3. It is bad at predicting variable generalisations.
L4. Its evaluation is based on a small dataset with 109 inductive problems.

We overcame these problems with \texttt{sem.ind}, a new recommendation tool for the \textit{induct} tactic. Similarly to \texttt{smart.induct}, \texttt{sem.ind} suggests what arguments to pass to the \textit{induct} tactic for a given inductive problem. Our overall contribution is that:

we built a system that predicts how one should apply proof by induction in Isabelle/HOL both quickly and accurately.

Even though we built \texttt{sem.ind} for Isabelle/HOL, our approach is transferable to other proof assistants based on tactics: no matter what proof assistants we use, we need an architecture that aggressively removes less promising candidates to address L1 (presented in Section 2), a procedure to construct promising induction candidates without missing out too many good ones to address L2 (presented in Section 3), and domain-agnostic heuristics that analyse not only the syntactic structures of inductive problems but the definitions of relevant constants to address L3 (presented in Section 4). Finally, Section 5 justifies our claims through extensive evaluations based on 1095 inductive problems, addressing L4.

2 The Overall Architecture

Figure 1 illustrates the overall architecture of \texttt{sem.ind}, consisting of 5 steps to produce and select candidate tactics as follows.

Step 1. \texttt{sem.ind} produces a set of sequences of induction terms and induction rules for the \textit{induct} tactic from a given inductive problem. The aim of this step is to produce a small number of candidates intelligently, so that it covers most promising sequences of induction terms and induction rules while avoiding a combinatorial blowup. We expound the algorithm to achieve this goal in Section 3.

Step 2. \texttt{sem.ind} applies the \textit{induct} tactic with the sequences of arguments produced in Step 1 to discard less promising candidates. \texttt{sem.ind} decides a sequence of arguments is unpromising if the sequence satisfies any of the following conditions:

- the \textit{induct} tactic fails with an error message, or
- one of the resulting intermediate goals is identical to the original goal itself.

Step 3. \texttt{sem.ind} applies 36 pre-defined heuristics written in SelFiE [Nagashima, 2020a], which we explain in Section 4 using our running example. These heuristics judge the validity of induction terms and induction rules with respect to the proof goal and the relevant definitions. Each heuristic is implemented as an assertion on inductive problems and arguments of the \textit{induct} tactic, and each assertion is tagged with a value. If an assertion returns True for a sequence of arguments, \texttt{sem.ind} gives the tagged value to the sequence. \texttt{sem.ind} sums up such points from the 36 heuristics to compute the score for each sequence. Based on these scores,
sem_ind sorts sequences of arguments from Step 2 and selects the five most promising sequences for further processing.

**Step 4.** After deciding induction terms and induction rules for the induct tactic in Step 3, sem_ind adds arguments for the arbitrary field to the sequences of arguments passed from Step 3. Firstly, sem_ind collects free variables in the proof goal that are not induction terms for each sequence from Step 3. Then, it constructs the powerset of such free variables and uses each set in the powerset as the arguments to the arbitrary field of the induct tactic. For example, if sem_ind receives (induct xs) from Step 3 for our running example of list reversal, it produces \{\{\}, \{ys\}\} as the powerset because xs and ys are the only free variables in the goal and xs appears as the induction term. This powerset leads to the following two induct tactics: (induct xs), and (induct xs arbitrary:ys).

**Step 5.** For each remaining sequence, sem_ind applies 8 pre-defined SeLFiE heuristics to judge the validity of generalisation. Again, each heuristic is tagged with a value, which is used to compute the final score for each candidate: for each sequence, sem_ind adds the score from the generalisation heuristics to the score from Step 3 to decide the final score for each sequence. Based on these final scores, sem_ind sorts sequences of arguments from Step 5 and presents the 10 most promising sequences in the Output panel of Isabelle/jEdit, the default proof editor for Isabelle/HOL [Wenzel, 2012].

We developed sem_ind entirely within the Isabelle ecosystem without any dependency to external tools. This allows for the easy installation process of sem_ind: to use sem_ind, users only have to download the relevant Isabelle files from our public GitHub repository\(^1\) and install sem_ind using the standard Isabelle command. The seamless integration into the Isabelle proof language, Isar [Wenzel, 2002], lets users invoke sem_ind within their ongoing proof development and copy a recommended induct tactic to the right location with one click as shown in Figure 2.

\(^1\)https://github.com/data61/PSL

On the other hand, it is often necessary to pass multiple induction terms to the induct tactic, and the order of such induction terms is important to apply the induct tactic effectively. Moreover, it is sometimes indispensable to pass the same induction term multiple times to the induct tactic, so that each of them corresponds to a distinct occurrence of the same term in the proof goal. What is worse, induction terms do not have to be variables: they can be compound terms such as function applications.

Enumerating all possible sequences of induction terms leads to a combinatorial explosion. To avoid such combinatorial explosion, sem_ind produces sequences of induction terms and induction rules taking a syntax-directed approach, which traverses the syntax tree representing the proof goal while collecting plausible sequences of induction terms and rules as follows.

**Step 1-A.** The collection starts at the root node of the syntax tree with an empty set of sequences of induction arguments.

**Step 1-B.** If the current node is a function application, sem_ind takes arguments to the function, produces a set of lists of such arguments while preserving their order. This set of lists represents candidates for induction terms. If the function in this function application is a constant with a relevant induction rule stored in the proof context, sem_ind produces candidate induct tactics with and without this rule for the rule field. For example, if the current node is rev2 xs ys, Step 1-B produces (induct xs) and (induct xs ys rule:rev2.induct), as well as other candidates such as (induct xs ys) and (induct xs ys rule:rev2.induct).

**Step 1-C.** If any sub-terms of the current node is a compound term, sem_ind moves down to such sub-terms in the syntax tree and repeats S1-b to collect more candidates for induction arguments.

**Step 1-D.** sem_ind finishes Step 1 when it reaches the leaf nodes in all branches of the syntax tree.

This syntax-directed argument construction avoids a combinatorial explosion at the cost of missing out some effective sequences of induction arguments. One notable example is the omission of simultaneous induction, which is essential to tackle inductive problems with mutually recursive functions. Our evaluation results in Section 5 show that despite the omission of such cases sem_ind manages to recommend correct induction arguments for most of the cases that appear in day-to-day theorem proving.

In principle, this smart construction of candidate induct tactics ignores handcrafted induction rules: in Step 1-B sem_ind collects induction rules derived automatically by Isabelle when defining relevant functions. For example, sem_ind picks up rev2.induct when seeing rev2 in our running example since rev2.induct is an induction principle Isabelle automatically derived when defining rev2. This constraint is unavoidable since we cannot predict what induction rules proof engineers will manually develop in the future for problem domains that may not even exist yet.

The notable exceptions to this principle are induction rules manually developed in Isabelle’s standard library: in Step 1-B

\[\text{lemma \ "rev2 \ xs \ ys = rev1 \ xs \ @ \ ys"}\
\[\begin{array}{l}
1\text{st candidate is } \text{apply (induct \ "xs" arbitrary:ys)}\\
\text{(* The score is 103.0 out of 104.0. *)}\\
2\text{nd candidate is } \text{apply (induct \ "xs")}\\
\text{(* The score is 102.0 out of 104.0. *)}
\end{array}\]

Figure 2: The user-interface of sem_ind.
the smart construction algorithm collects some manually developed induction rules from the standard library if the rules seem to be relevant to the inductive problem at hand. This optimisation is reasonable: some concepts in the standard library, such as lists and natural numbers, are used in many projects and have useful induction rules handcrafted by Isabelle experts.

4 Induction and Generalisation Heuristics

We now have a closer look at heuristics used in Step 3 and Step 5. To produce accurate recommendations quickly, heuristics for sem_ind have to satisfy the following two criteria.

C1: The heuristics should be applicable to a wide range of problem domains, some of which do not exist yet.

C2: They should be able to analyse not only the syntactic structures of the inductive problems at hand but also the definitions of relevant constants in terms of how such constants are used within the inductive problems.

To satisfy the above criteria, we choose SeLFiE [Nagashima, 2020a] as our implementation language to encode heuristics. SeLFiE is a meta-language to encode heuristics for inductive theorem proving as assertions. A SeLFiE assertion takes a pair of arguments to the induct tactic and an inductive problem with relevant definitions. The assertion should return True if the choice of argument of the induct tactic is compatible with the heuristic.

The exact definitions of our 44 heuristics are not informative or possible due to the page limit. Therefore, instead of presenting each heuristic, this section introduces one simple generalisation heuristic written in SeLFiE to demonstrate how we address the above two criteria using SeLFiE.

Program 2 and 3 define the following generalisation heuristic introduced by Nipkow et al. [Nipkow et al., 2002]:

(Variable generalisation) should not be applied blindly. It is not always required, and the additional quantifiers can complicate matters in some cases. The variables that need to be quantified are typically those that change in recursive calls.

Figure 3 illustrates how this heuristic justifies the generalisation of ys in our running example as an informal dialogue between Program 2 and Program 3. In this dialogue, Program 2 analyses the syntactic structure of the proof goal in terms of the arguments of the induct tactic, whereas Program 3 analyses the definition of rev2 in terms of how rev2 is used in the goal. Note that Program 2 and Program 3 realise this dialogue through a definition-aware generalisation heuristic as a dialogue.

Figure 3: Definition-aware generalisation heuristic as a dialogue.
• rev2 for f_term,
• the only occurrence of rev2 in the proof goal for f_occ,
• the occurrence of ys on the left-hand side of the equation in the proof goal for arb_occ,
• 2 for generalise_nth,

we can satisfy the first conjunct for all arb_terms because in the proof goal ys appears as the second argument of the only occurrence of rev2.

In the second conjunct, Program 2 uses \( \exists_{\text{def}} \) to ask Program 3 if there exists a clause defining rev2 that satisfies the condition specified in Program 3. Formally speaking, Program 3 examines if there is a term occurrence, nth_param_on_lhs, such that nth_param_on_lhs is the generalise_nth argument on the left-hand side in the equation defining f_term, but an occurrence of f_term on the right-hand side has a different term for its second argument.

In the second clause defining rev2, the second argument of rev2 is ys on the left-hand side, while the second argument of rev2 is x \# ys on the right-hand side. Therefore, Program 3 returns True to \( \exists_{\text{def}} \) in Program 2, with which Program 2 confirms that the candidate arguments of the induct tactic satisfy Nipkow's heuristic.

Attentive readers may have noticed that Program 2 and Program 3 satisfy the aforementioned two criteria. They satisfy C1 because they refer to problem specific constants and arguments, such as rev2 and ys, abstractly using quantifiers, so that they can be applicable to other inductive problems. They also satisfy C2 because Program 3 analyses the definitions of relevant constants, such as rev2, while Program 2 analyses the syntactic structures of the problem. This is why \texttt{sem_ind} achieves higher coincidence rates compared to its predecessor, \texttt{smart_induct} [Nagashima, 2020b], reported in Section 5.

In total, we implemented 44 heuristics in SeLFiE. 36 of them are induction heuristics and 8 of them are generalisation heuristics. We adopted some heuristics from \texttt{smart_induct}, and we newly implemented others based on literature and our expertise. As discussed above, SeLFiE allows us to encode heuristics that transcend problem domains using quantifiers. At the same time, however, some basic concepts, such as lists, sets and natural numbers, appear in a wide range of verification projects. Therefore, we developed 20 SeLFiE heuristics that explicitly refer to concrete constants or manually derived induction rules from the standard library to improve the accuracy of recommendations for problems involving such commonly used concepts. Unlike other parts of this paper, these 20 heuristics involve expertise specific to Isabelle/HOL.

### 5 Evaluation

We evaluated \texttt{sem_ind} against \texttt{smart_induct}. Our focus is to measure the accuracy of recommendations and execution time necessary to produce recommendations. All evaluations are conducted on a MacBook Pro (15-inch, 2019) with 2.6 GHz Intel Core i7 6-core memory 32 GB 2400 MHz DDR4.

Unfortunately, it is, in general, not possible to decide whether a given application of the induct tactic is right for a given problem. In particular, even if we can finish a proof search after applying the induct tactic, this does not guarantee that the arguments passed to the induct tactic are a good choice. For example, it is possible to prove our motivating example by applying (induct ys); however, the necessary proof script following this application of the induct tactic becomes unnecessarily lengthy.

Therefore, we adopt coincidence rates as the surrogate for success rates to approximate the accuracy of \texttt{sem_ind}'s recommendations: we measure how often recommendations of \texttt{sem_ind} coincide with the choice of human engineers. Since there are often multiple equally valid sequences of induction arguments for a given inductive problem, we should regard coincidence rates as conservative estimates of true success rates.

As our evaluation target, we use 22 Isabelle theories with 1,095 applications of the induct tactic from the Archive of Formal Proofs (AFP) [Klein et al., 2004]. The AFP is an online repository of formal proofs in Isabelle/HOL. Each entry in the AFP is peer-reviewed by Isabelle experts prior to acceptance, which ensures the quality of our target theory files. Therefore, if \texttt{sem_ind} achieves higher coincidence rates for our target theory files, we can say that \texttt{sem_ind} produces good recommendations for many problems. To the best of our knowledge, this is the most diverse dataset used to measure recommendation tools for proof by induction. For example, when Nagashima evaluated \texttt{smart_induct} they used only 109 invocations of the induct tactic from 5 theory files, all of which are included in our dataset.

#### 5.1 Coincidence Rates within 5.0 Seconds

Table 1 shows coincidence rates of both \texttt{sem_ind} and \texttt{smart_induct} within 5.0 seconds of timeout.

<table>
<thead>
<tr>
<th>tool</th>
<th>top 1</th>
<th>top 3</th>
<th>top 5</th>
<th>top 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{sem_ind}</td>
<td>38.2%</td>
<td>59.3%</td>
<td>64.5%</td>
<td>72.7%</td>
</tr>
<tr>
<td>\texttt{smart_induct}</td>
<td>20.1%</td>
<td>42.8%</td>
<td>48.5%</td>
<td>55.3%</td>
</tr>
</tbody>
</table>

Table 1: Overall coincidence rates within 5.0 seconds of timeout.

<table>
<thead>
<tr>
<th>tool</th>
<th>top 1</th>
<th>top 3</th>
<th>top 5</th>
<th>top 10</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{sem_ind}</td>
<td>54.5%</td>
<td>63.6%</td>
<td>72.7%</td>
<td>72.7%</td>
</tr>
<tr>
<td>\texttt{smart_induct}</td>
<td>0.0%</td>
<td>0.0%</td>
<td>0.0%</td>
<td>9.1%</td>
</tr>
</tbody>
</table>

Table 2: Coincidence rates for Nearest_Neighbors.thy.
Table 3 shows that for all theory files \textit{sem-ind} produces more recommendations than \textit{smart-induct} does for all timeouts specified in this evaluation, proving the superiority of \textit{sem-ind} over \textit{smart-induct} in terms of the execution time necessary to produce recommendations. In fact, the median values of execution time for these 1,095 problems are 1.06 seconds for \textit{sem-ind} and 2.79 seconds for \textit{smart-induct}. That is to say, \textit{sem-ind} achieved 62\% of reduction in the median value of execution time.

5.2 Return Rates for 5 Timeouts

\textit{sem-ind} achieves the higher accuracy by analysing not only the syntactic structures of inductive problems but also the definitions of constants relevant to the problems. Inevitably, this requires larger computational resources: the SeLFiE interpreter has to examine not only the syntax tree representing proof goals but also the syntax trees representing the definitions of relevant constants. However, thanks to the syntax-directed candidate construction algorithm presented in Section 3 and aggressive pruning strategy presented in Section 2, \textit{sem-ind} provides recommendations faster than \textit{smart-induct} does.

This performance improvement is presented in Table 3, which shows how often \textit{sem-ind} and \textit{smart-induct} return recommendations within certain timeouts. For example, the return rate of \textit{sem-ind} is 8.8\% for 0.2 seconds. This means that \textit{sem-ind} returns recommendations for 8.8\% of proofs by induction within 0.2 seconds. On the other hand, the return rate of \textit{smart-induct} is 0.0\% for 0.2 seconds.

Table 3 shows that for all theory files \textit{sem-ind} produces more recommendations than \textit{smart-induct} does for all timeouts specified in this evaluation, proving the superiority of \textit{sem-ind} over \textit{smart-induct} in terms of the execution time necessary to produce recommendations. In fact, the median values of execution time for these 1,095 problems are 1.06 seconds for \textit{sem-ind} and 2.79 seconds for \textit{smart-induct}. That is to say, \textit{sem-ind} achieved 62\% of reduction in the median value of execution time.

6 Related Work

Boyer and Moore invented the waterfall model [Moore, 1973] for inductive theorem proving for a first-order logic on Common Lisp [Jr., 1982]. In the original waterfall model, a prover tries to apply any of the following six techniques: simplification, destructor elimination, cross-fertilization, generalisation, elimination of irrelevance, and induction to emerging sub-goals until it solves all sub-goals.

The most well-known prover based on the waterfall model is ACL2 [Moore, 1998]. To decide how to apply induction, ACL2 computes a score, called hitting ratio, based on a fixed formula [Boyer and Moore, 1979; Moore and Wirth, 2013] to estimate how good each induction scheme is. Instead of computing a hitting ratio, we use SeLFiE to encode our induction heuristics as assertions. While ACL2 produces many induction schemes and computes the corresponding hitting ratios, \textit{sem-ind} produces a small number of promising sequences of induction terms and rules.

For Isabelle/HOL, we developed a proof strategy language, PSL [Nagashima and Kumar, 2017]. PSL’s interpreter discharges easy induction problems by conducting expensive proof searches, and its extension to abductive reasoning tries to identify auxiliary lemmas useful to prove inductive problems [Nagashima and Parsert, 2018]. While our abductive reasoning mechanism took a top-down approach, Johansson \textit{et al.} took a bottom-up approach [Johansson \textit{et al.}, 2014] based on the idea of theory exploration.

There are ongoing attempts to extend saturation-based superposition provers with induction: Cruanes presented an extension of typed superposition that can perform structural induction [Cruanes, 2017], while Reger \textit{et al.} incorporated lightweight automated induction [Reger and Voronkov, 2019] to the Vampire prover [Kovács and Voronkov, 2013] and Hajdú \textit{et al.} extended it to cover induction with generalisation [Hajdú \textit{et al.}, 2020]. A straightforward comparison to their approaches is difficult as their provers are based on less expressive logics and different proof calculi. However, we argue that one advantage of \textit{sem-ind} over their approaches is that \textit{sem-ind} never introduces axioms that risk the consistency of Isabelle/HOL. Furthermore, our evaluation consists of a wider range of problem domains written by experienced Isabelle users based on their diverse interests: Hajdú’s evaluation involved a number of inductive problems, but the problem domains were limited to lists, natural numbers, and trees.

Similarly to \textit{sem-ind}, TacticToe [Gauthier \textit{et al.}, 2017; Gauthier \textit{et al.}, 2021] for HOL4, Tactician [Blaauwbroek \textit{et al.}, 2020] for Coq, and PaMpeR [Nagashima and He, 2018] for Isabelle are meta-tactic tools seamlessly integrated in proof assistants’ ecosystems; however, none of them logically analyse inductive problems or predict arguments of the \textit{induct} tactic accurately. Unlike these tools, \textit{sem-ind} presents accurate recommendations without relying on statistical machine learning.

Despite the growing interest in deep learning for theorem proving, [Kaliszyn \textit{et al.}, 2017; Bansal \textit{et al.}, 2019; Yang and Deng, 2019; Jakubuv \textit{et al.}, 2020; Paliwal \textit{et al.}, 2020; Chvalovský, 2019; Sekiyama and Suenaga, 2018; Potrowski and Urban, 2020; Loos \textit{et al.}, 2017; Li \textit{et al.}, 2021], we mindfully avoided deep learning since we have only a limited number of inductive problems available. Instead of deep learning, we used SeLFiE’s quantifiers to encode our heuristics in a domain-agnostic style. To the best of our knowledge, no project based on deep learning has managed to predict arguments to the \textit{induct} tactic accurately.
7 Conclusion

We presented sem_ind, a recommendation tool for proof by induction. sem_ind constructs candidate induct tactics for a given inductive problem while avoiding combinatorial explosion, and it selects promising candidates by filtering out unpromising candidates and scoring remaining ones. To give scores to each remaining candidate, we encoded 36 heuristics in SeLFiE to decide on which terms and with which rules we should apply the induct tactic, as well as 8 SeLFiE heuristics to decide which variables to generalise.

Our evaluation based on 1,095 inductive problems from 22 theory files showed that compared to the existing tool, smart_ind, sem_ind achieves a 90.0% increase of the coincidence rate from 20.1% to 38.2% for the most promising candidate, while achieving a 62.0% decrease of the median value of execution time. In particular, sem_ind surpassed the accuracy of the existing tool by a wide margin for inductive problems involving variable generalisation.

Currently, sem_ind uses manually specified weights for heuristics. It remains as our future work to optimise such weights using evolutionary computation [Nagashima, 2019b] and to integrate sem_ind into a larger AI tool for Isabelle/HOL [Nagashima, 2020c].

Acknowledgements

We thank the anonymous reviewers for useful feedback both at TACAS2020 and IJCAI2021. This work has been supported by the following grants:

• the grant of Singapore NRF National Satellite of Excellence in Trustworthy Software Systems (NSoE-TSS),

• the European Regional Development Fund under the project AI & Reasoning (reg.no.CZ.02.1.01/0.0/0.0/15_003/0000466),

• the ERC starting grant no. 714034, and

• NII under NII-Internship Program 2019-2nd call.

References


