Smart Induction for Isabelle/HOL (System Description) ⋆

Yutaka Nagashima[1,2][0000−0001−6693−5325]

[1] Czech Technical University in Prague, Prague, Czech Republic
Yutaka.Nagashima@cvut.cz
[2] University of Innsbruck, Innsbruck, Austria

Abstract. Proof assistants offer tactics to facilitate inductive proofs. However, it still requires human ingenuity to decide what arguments to pass to these tactics. To automate this process, we present `smart_induct` for Isabelle/HOL. Given an inductive problem in any problem domain, `smart_induct` lists promising arguments for the `induct` tactic without relying on a search. Our evaluation demonstrated `smart_induct` produces valuable recommendations across problem domains.

Keywords: proof by induction · Isabelle · logical feature extraction

1 Induction

Given the following two simple reverse functions defined in Isabelle/HOL [19], how do you prove their equivalence [18] ?

```
primrec rev::"'a list =>'a list" where
  "rev  []      = []"
| "rev (x # xs) = rev xs @ [x]"
fun itrev::"'a list =>'a list =>'a list" where
  "itrev  []    ys = ys"
| "itrev (x#xs) ys = itrev xs (x#ys)"
lemma "itrev xs ys = rev xs @ ys"
```

where `#` is the list constructor, and `@` appends two lists. Using the `induct` tactic of Isabelle/HOL, we can prove this inductive problem in multiple ways:

```
lemma prf1: "itrev xs ys = rev xs @ ys"
  apply(induct xs arbitrary: ys) by auto
lemma prf2: "itrev xs ys = rev xs @ ys"
  apply(induct xs ys rule:itrev.induct) by auto
```

⋆ This work was supported by the European Regional Development Fund under the project AI & Reasoning (reg. no.CZ.02.1.01/0.0/0.0/15_003/0000466).

prf1 applies structural induction on xs while generalizing ys before applying induction by passing ys to the arbitrary field. On the other hand, prf2 applies functional induction on itrev by passing an auxiliary lemma, itrev.induct, to the rule field.

There are other lesser-known techniques to handle difficult inductive problems using the induct tactic, and sometimes users have to develop useful auxiliary lemmas manually; However, for most cases the problem of how to apply induction boils down to the the following three questions:

- On which terms do we apply induction?
- Which variables do we generalize using the arbitrary field?
- Which rule do we use for functional induction using the rule field?

To answer these questions automatically, we developed a proof strategy language, PSL [16]. Given an inductive problem, PSL produces various combinations of induction arguments for the induct tactic and conducts an extensive proof search based on a given strategy. If PSL completes a proof search, it identifies the appropriate combination of arguments for the problem and presents the combination to the user; However, when the search space becomes enormous, PSL cannot complete a search within a realistic timeout and fails to provide any recommendation, even if PSL produces the right combination of induction arguments. For further automation of proof by induction, we need a tool that satisfies the following two criteria:

- The tool suggests right induction arguments without resorting to a search.
- The tool suggests right induction arguments for any inductive problems.

In this paper we present smart_induct, a recommendation tool that addresses these criteria. smart_induct is available at GitHub [11] together with our running example and the evaluation files discussed in Section 3. The seamless integration of smart_induct into Isabelle's ecosystem made smart_induct easy to install and easy to use. The implementation of smart_induct is specific to Isabelle/HOL; However, the underlying concept is transferable to other tactic based proof assistants including HOL4 [22], Coq [23], and Lean [10].

To the best of our knowledge smart_induct is the first recommendation tool that analyzes the syntactic structures of proof goals across problem domains and advise how to apply the induct tactic without resorting to a search.

2 smart_induct: the System Description

Fig. 2 illustrates the following internal workflow of smart_induct. When invoked by a user, the first step produces many variants of the induct tactic with different combinations of arguments. Secondly, the multi-stage screening step filters out less promising combinations induction arguments. Thirdly, the scoring step evaluates each combination to a natural number using logical feature extractors implemented in LiFtEr [13] and reorder the combinations based on their scores. Lastly, the short-listing step takes the best 10 candidates and print them in the Output panel of Isabelle/jEdit. In this section, we explore details of Step 1 to Step 3.

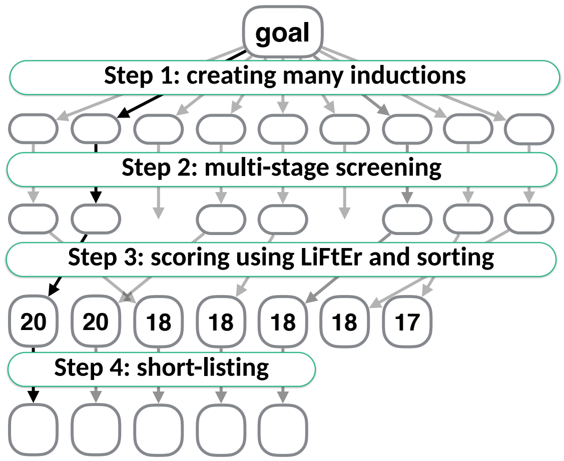

Fig. 1: The Workflow of `smart_induct`.

2.1 Step 1: Creation of Many Induction Tactics.

`smart_induct` first inspects the given proof goal and produces a number of combinations of arguments for the `induct` tactic taking the following procedure: `smart_induct` collects variables and constants appearing in the goal. If a constant has an associated induction rule, `smart_induct` also collects that rule from the underlying proof context, Then, from these variables and induction rules, `smart_induct` produces a power set of combinations of arguments for the `induct` tactic. In our example, `smart_induct` produces 40 combinations of induction arguments.

If the size of this power set is enormous, we cannot store all the produced induction tactics in our machines. Therefore, `smart_induct` produces this set using a lazy sequence and takes only the first 10000 of them for further processing.

2.2 Step 2: Multi-Stage Screening.

10000 is still a large number, and feature extractors used in Step 3 often often involve nested traversals of nodes in the syntax tree representing a proof goal, leading to high computational costs. Fortunately, the application of the `induct` tactic itself is not computationally expensive in many cases: We can apply the `induct` tactic to a proof goal and have intermediate sub-goals at a low cost. Therefore, in Step 2, `smart_induct` applies the `induct` tactic to the given proof goal using the various combinations of arguments from Step 1 and filter out some of them through the following two stages.

Stage 1. In the first stage, `smart_induct` filters out those combinations of induction arguments, with which Isabelle/HOL does not produce an intermediate goal. Since we have no known theoretical upper bound for the computational cost for the `induct` tactic, we also filter out those combinations of arguments, with which the `induct` tactic does not return a result within a pre-defined timeout. In our running example, this stage filters out 8 combinations out of 40.

Stage 2. Taking the results from the previous stage, Stage 2 scans both the original goal and the newly introduced intermediate sub-goals at the same time to further filter out less promising combinations. More concretely, this stage filters out all combinations of arguments if they satisfy any of the following conditions:

- Some of newly introduced sub-goals are identical to each other.
- All newly introduced sub-goals contain the original first sub-goal as their sub-term even though there was no locally introduced assumptions.
- A newly introduced sub-goal contains a schematic variable even though the original first sub-goal did not contain a schematic variable.

In our example, Stage 2 filters out 4 combinations out of 32. Note that these tests on the original goal and resulting sub-goals do not involve nested traversals of nodes in the syntax tree representing goals. For this reason, the computational cost of this stage is often smaller than that of Step 3.

2.3 Step 3: Scoring Induction Arguments using `LiFtEr`.

Step 3 carefully investigates the remaining candidates using heuristics implemented in `LiFtEr` [13]. `LiFtEr` is a domain-specific language to encode induction heuristics in a style independent of problem domains. Given a proof goal and combination of induction arguments, the `LiFtEr` interpreter mechanically checks if the combination is appropriate for the goal in terms of a heuristic written in `LiFtEr`. The interpreter returns `True` if the combination is compatible with the heuristic and `False` if not. We illustrated the details of `LiFtEr` in our previous work [13] with many examples. In this paper, we focus on the essence of `LiFtEr` and show one example heuristic used in `smart_induct`.

`LiFtEr` supports four types of variables: natural numbers, induction rules, terms, and term occurrences. An induction rule is an auxiliary lemma passed to the `rule` field of the `induct` tactic. The domain of terms is the set of all sub-terms appearing in a given goal. The logical connectives ($\vee$, $\wedge$, $\rightarrow$, and $\neg$) correspond to the connectives in the classical logic. `LiFtEr` offers atomic assertions, such as `is_rule_of`, to examine the property of each atomic term. Quantifiers bring the the power of abstraction to `LiFtEr`, which allows `LiFtEr` users to encode induction heuristics that can transcend problem domains. Quantification over `term` can be restricted to the induction terms used in the `induct` tactic.

We encoded 20 heuristics in `LiFtEr` for `smart_induct`. Some of them examine a combination of induction arguments in terms of functional induction, whereas others check the combination for structural induction or rule inversion. Program 1, for example, encodes a heuristic for functional induction. In English this heuristic reads as follows:

> if there exists a rule, $r1$, in the `rule` field of the `induct` tactic, then there exists a term $t1$ with an occurrence $to1$, such that $r1$ is derived by

Program 1 A `LiFtEr` Heuristic used in `smart_induct`.

```
 ∃ r1 : rule. True
→
 ∃ r1 : rule.
   ∃ t1 : term.
     ∃ to1 : term_occurrence ∈ t1 : term.
         r1 is_rule_of to1
       ∧
         ∀ t2 : term ∈ induction_term.
           ∃ to2 : term_occurrence ∈ t2 : term.
             ∃ n : number.
                 is_nth_argument_of (to2, n, to1)
               ∧
                 t2 is_nth_induction_term n
```

Isabelle when defining $t1$, and for all induction terms $t2$, there exists an occurrence $to2$ of $t2$ such that, there exists a number n, such that $to2$ is the nth argument of $to1$ and that $t2$ is the nth induction terms passed to the `induct` tactic.

If we apply this heuristic to our running example, `prf2`, the `LiFtEr` interpreter returns `True`: there is an argument, `itrev.induct`, in the `rule` field, and the occurrence of its related term, `itrev`, in the proof goal takes all the induction terms, `xs` and `ys`, as its arguments in the same order.

Attentive readers may have noticed that Program 1 is independent of any types or constants specific to `prf2`. In stead of handling specific constructs explicitly, Program 1 analyzes the structure of the goal with respect to the arguments passed to the `induct` tactic in an abstract way using quantified variables and logical connectives. This power of abstraction let `smart_induct` evaluate whether a given combination of arguments to the `induct` tactic is appropriate for a user-defined proof goal consisting of user-defined types and constants, even though such constructs are not available to the `smart_induct` developers. In fact, none of the 20 heuristics relies on constructs specific to any problem domain.

In Step 3, `smart_induct` applies these 20 heuristics to the results from Step 2. For each heuristic, `smart_induct` gives one point to each combination of `induct` arguments if the `LiFtEr` interpreter returns `True` for that combination. Then, `smart_induct` reorder these combinations based on their scores to present the most promising combinations to the user in Step 4.

3 Evaluation

In general it is not possible to measure if a combination of induction arguments is correct for a goal. Therefore, we evaluated trustworthiness of `smart_induct`'s recommendations using *coincident rates*: We counted how often its recommendation coincides with the choices of Isabelle experts.

Table 1: Coincidence Rates of `smart_induct`.

theory	total	top_1	top_3	top_5	top_10
`DFS`	10	6 (60%)	9 (90%)	9 (90%)	9 (90%)
`Nearest_Neighbors`	16	3 (19%)	4 (25%)	7 (44%)	12 (75%)
`RST_RBT`	24	24 (100%)	24 (100%)	24 (100%)	24 (100%)
sum	50	33 (65%)	37 (74%)	40 (80%)	45 (90%)

Table 1 shows the coincidence rates of `smart_induct` for three theory files about different problem domains written by different researchers: `DFS` is a part of the formalisation of depth-first search [20], `Nearest_Neighbors` is from the foramlisation of multi-dimensional binary search trees [21], and `RST_RBT` is from that of priority search tree [6].

The column named "total" shows the total number of proofs by induction in each file. The four columns titled with "top_n" show for how many proofs by induction in each file the proof authors' choice of induction arguments coincides with one of the n most promising recommendations from `smart_induct`. For example, top_3 for `DFS` indicates if `smart_induct` recommends 3 most promising combinations for each goal, the proof author used one of the three recommended combinations for 90% of proofs by induction in the file. Note that we often have multiple equally valid combinations of induction arguments for a given proof goal: Our running example has two proofs, `prf1` and `prf2`, and both of them are appropriate to prove this equivalence theorem. Therefore, we should regard a coincidence rate as a conservative estimate of true success rate.

A quick glance over Table 1 would give the impression that `smart_induct`'s performance depends heavily on problem domains: `smart_induct` demonstrated the perfect result for `RST_RBT`, whereas for `Nearest_Neighbor` the coincidence rate remains at 44% for top_5.

However, a closer investigation of the results reveal that the difference of performance comes from the style of induction rather than domain specific items such as the types or constructs appearing in goals: In `RST_RBT`, all 24 proofs by induction are functional inductions, whereas `Nearest_Neighbor` has only 5 functional inductions out of 16. As Table 5 in Appendix shows if we focus on proofs by functional induction the coincidence rate rises to 60% for "top_1" and 80% for "top_3" for `Nearest_Neighbor`.

Furthermore, Table 5 also suggests that `smart_induct` has relatively low coincidence rates for structural induction because `smart_induct` is not able to predict which variables to generalize using the `arbitrary` field: Since structural induction tends to involve generalization of variables more often than functional induction does, `smart_induct` struggles to predict the choice of experts for structural induction.

In Table 2 we computed the coincidence rates for `Nearest_Neighbor` again based on a different criterion: This time we ignored the `rule` and `arbitrary`

Table 2: Coincidence Rates of `smart_induct` Based Only on Induction Terms.

theory	total	top_1	top_3	top_5	top_10
`Nearest_Neighbors`	16	5 (31%)	12 (75%)	15 (94%)	15 (94%)

fields and took only induction terms into consideration. The large discrepancies between the numbers in Table 1 and those in Table 2 indicate that even for structural inductions `smart_induct` is often able to predict on which variables experts apply induction but fails to predict which variables to generalize.

The limited performance in predicting experts' use of the `arbitrary` field stems from `LiFtEr`'s limited capability to examine semantic information of proof goals. Even though `LiFtEr` offers quantifiers, logical connectives, and atomic assertions to analyze the syntactic structure of a goal in an abstract way, `LiFtEr` does not offer enough supports to analyze the semantics of the goal. For more accurate prediction of variable generalization, `smart_induct` needs a language to analyze not only the structure of a goal itself but also the structure of the definitions of types and constants appearing in the goal in an abstract way.

4 Conclusion

We presented `smart_induct`, a recommendation tool for proof by induction in Isabelle/HOL. Our evaluation showed `smart_induct`'s excellent performance in recommending how to apply functional induction and identifying induction terms for structural induction, even though recommendation of variable generalization remains as a challenging task. It is still an open question how far we can improve `smart_induct` by combining it with search based systems [16,17] and approaches based on evolutionary computation [14] or statistical machine learning [12].

Related Work. The most well-known approach for inductive problems is called the Boyer-Moore waterfall model [7]. This approach was invented for a first-order logic on Common Lisp. ACL2 [8] is a commonly used waterfall model based prover. When deciding how to apply induction, ACL2 computes a score, called *hitting ratio*, to estimate how good each induction scheme is for the term which it accounts for and proceeds with the induction scheme with the highest hitting ratio [2, 9].

Compared to the hitting ratio used in ACL2, `smart_induct` analyzes the structures of proof goals directly using `LiFtEr`. While ACL2 produces many induction schemes and computes their hitting ratios, `smart_induct` does not directly produce induction schemes but analyzes the given proof goal, the arguments passed to the `induct` method, and the emerging sub-goals.

Jiang *et al.* ran multiple waterfalls [5] in HOL Light [4]. However, when deciding induction variables, they naively picked the first free variable with recursive type and left the selection of appropriate induction variables as future work.

Machine learning applications to tactic-based provers [1, 3, 15] focus on selections of tactics rather than selections of terms as arguments to tactics, even though the choice of induction arguments is essential for inductive problems.

References

1. Bansal, K., Loos, S.M., Rabe, M.N., Szegedy, C., Wilcox, S.: HOList: An environment for machine learning of higher order logic theorem proving. In: Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA. pp. 454–463 (2019), `http://proceedings.mlr.press/v97/bansal19a.html`
2. Boyer, R.S., Moore, J.S.: A computational logic handbook, Perspectives in computing, vol. 23. Academic Press (1979)
3. Gauthier, T., Kaliszyk, C., Urban, J.: TacticToe: Learning to reason with HOL4 tactics. In: Eiter, T., Sands, D. (eds.) LPAR-21, 21st International Conference on Logic for Programming, Artificial Intelligence and Reasoning, Maun, Botswana, May 7-12, 2017. EPiC Series in Computing, vol. 46, pp. 125–143. EasyChair (2017), `http://www.easychair.org/publications/paper/340355`
4. Harrison, J.: HOL light: A tutorial introduction. In: Formal Methods in Computer-Aided Design, First International Conference, FMCAD '96, Palo Alto, California, USA, November 6-8, 1996, Proceedings. pp. 265–269 (1996), `https://doi.org/10.1007/BFb0031814`
5. Jiang, Y., Papapanagiotou, P., Fleuriot, J.D.: Machine learning for inductive theorem proving. In: Artificial Intelligence and Symbolic Computation - 13th International Conference, AISC 2018, Suzhou, China, September 16-19, 2018, Proceedings. pp. 87–103 (2018), `https://doi.org/10.1007/978-3-319-99957-9_6`
6. Lammich, P., Nipkow, T.: Priority search trees. Archive of Formal Proofs (Jun 2019), `http://isa-afp.org/entries/Priority_Search_Trees.html`, Formal proof development
7. Moore, J.S.: Computational logic : structure sharing and proof of program properties. Ph.D. thesis, University of Edinburgh, UK (1973), `http://hdl.handle.net/1842/2245`
8. Moore, J.S.: Symbolic simulation: An ACL2 approach. In: Formal Methods in Computer-Aided Design, Second International Conference, FMCAD '98, Palo Alto, California, USA, November 4-6, 1998, Proceedings. pp. 334–350 (1998), `https://doi.org/10.1007/3-540-49519-3_22`
9. Moore, J.S., Wirth, C.: Automation of mathematical induction as part of the history of logic. CoRR **abs/1309.6226** (2013), `http://arxiv.org/abs/1309.6226`
10. de Moura, L.M., Kong, S., Avigad, J., van Doorn, F., von Raumer, J.: The Lean Theorem Prover (System Description). In: Automated Deduction - CADE-25 - 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings. pp. 378–388 (2015). https://doi.org/10.1007/978-3-319-21401-6_26, `https://doi.org/10.1007/978-3-319-21401-6_26`
11. Nagashima, Y.: data61/psl, `https://github.com/data61/PSL/releases/tag/v0.1.5-alpha`
12. Nagashima, Y.: Towards machine learning mathematical induction. CoRR **abs/1812.04088** (2018), `http://arxiv.org/abs/1812.04088`
13. Nagashima, Y.: LiFtEr: Language to encode induction heuristics for Isabelle/HOL. In: Programming Languages and Systems - 17th Asian Symposium, APLAS 2019, Nusa Dua, Bali, Indonesia, December 1-4, 2019, Proceedings. pp. 266–287 (2019). https://doi.org/10.1007/978-3-030-34175-6_14, `https://doi.org/10.1007/978-3-030-34175-6_14`
14. Nagashima, Y.: Towards evolutionary theorem proving for isabelle/hol. In: Proceedings of the Genetic and Evolutionary Computation Conference Companion, GECCO 2019, Prague, Czech Republic, July 13-17, 2019. pp.

419–420 (2019). https://doi.org/10.1145/3319619.3321921, https://doi.org/10.1145/3319619.3321921
15. Nagashima, Y., He, Y.: PaMpeR: proof method recommendation system for isabelle/hol. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering, ASE 2018, Montpellier, France, September 3-7, 2018. pp. 362–372 (2018), https://doi.org/10.1145/3238147.3238210
16. Nagashima, Y., Kumar, R.: A proof strategy language and proof script generation for Isabelle/HOL. In: de Moura, L. (ed.) Automated Deduction - CADE 26 - 26th International Conference on Automated Deduction, Gothenburg, Sweden, August 6-11, 2017, Proceedings. Lecture Notes in Computer Science, vol. 10395, pp. 528–545. Springer (2017), https://doi.org/10.1007/978-3-319-63046-5_32
17. Nagashima, Y., Parsert, J.: Goal-oriented conjecturing for isabelle/hol. In: Intelligent Computer Mathematics - 11th International Conference, CICM 2018, Hagenberg, Austria, August 13-17, 2018, Proceedings. pp. 225–231 (2018), https://doi.org/10.1007/978-3-319-96812-4_19
18. Nipkow, T., Klein, G.: Concrete Semantics - With Isabelle/HOL. Springer (2014), https://doi.org/10.1007/978-3-319-10542-0
19. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL - a proof assistant for higher-order logic, Lecture Notes in Computer Science, vol. 2283. Springer (2002)
20. Nishihara, T., Minamide, Y.: Depth first search. Archive of Formal Proofs (Jun 2004), http://isa-afp.org/entries/Depth-First-Search.html, Formal proof development
21. Rau, M.: Multidimensional binary search trees. Archive of Formal Proofs (May 2019), http://isa-afp.org/entries/KD_Tree.html, Formal proof development
22. Slind, K., Norrish, M.: A brief overview of HOL4. In: Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings. pp. 28–32 (2008). https://doi.org/10.1007/978-3-540-71067-7_6, https://doi.org/10.1007/978-3-540-71067-7_6
23. The Coq development team: The Coq proof assistant, https://coq.inria.fr

Appendix A

The three tables in this Appendix give the raw information of the evaluation presented in Section 3. The first column "line" is the line number in the respective file. The numbers in the column labelled as "total" represent how many combinations of arguments `smart_induct` produced for each inductive problem. The numbers in the column labelled as "1st" show how many of the created combinations passed the first screening stage explained in Section 2.2. The numbers in the column labelled as "2nd-b" show how many of the created combinations passed the second screening stage explained in Section 2.2. The numbers in the column labelled as "nth" show the ranks `smart_induct` gave to the combination of induction arguments used by the proof author. For example, if the number of "nth" is 2, this means `smart_induct` recommended the combination of induction arguments used by the proof author as the second most promising combination. The numbers in the column labelled as "score" represent the score `smart_induct` gave to the combination of arguments used by the proof author. For example, if the number of "score" is 18, this means `smart_induct` gave 18 points to the combination of argument used by the proof author, indicating that two feature extractors wrongly judged that the combination is not appropriate for the proof goal under consideration, assuming that the human proof author is always right. The values in the column labelled as `rule` show whether the proof author passed an argument to the `rule` field. The values in the column labelled as `arb` show whether the proof author passed an argument to the `arbitrary` field.

Table 3: Evaluation of `smart_induct` on `DFS.thy`

line	theorem name	total	1st	2nd-a	2nd-b	nth	score	`rule`	`arb`
27	`nexts_set`	128	16	12	12	2	20	no	no
42	-	20	8	4	4	2	20	no	no
87	`df2_invariant`	256	40	32	32	-	-	yes	no
126	`dfs_app`	3210	590	576	576	1	20	yes	no
131	-	384	144	136	136	2	20	yes	no
137	`visit_subset_dfs`	384	144	136	136	1	20	yes	no
140	`next_subset_dfs`	384	144	136	136	1	20	yes	no
161	`nextss_closed_dfs'`	384	144	136	136	1	20	yes	no
176	`Image_closed_trancl`	20	8	8	8	1	18	no	no
206	`dfs_subset_reachable...`	384	144	136	136	1	20	yes	no

Table 4: Evaluation of smart_induct on RST_RBT.thy

line	theorem name	total	1st	2nd-a	2nd-b	nth	score	rule	arb
170	inorder_combine	60	44	40	40	1	20	yes	no
175	inorder_upd	640	80	72	64	1	20	yes	no
185	inorder_del	100	36	32	28	1	20	yes	no
227	inv_baliL	512	136	128	128	1	20	yes	no
232	invc_baliR	512	136	128	128	1	20	yes	no
242	bheight_baliL	384	112	104	104	1	20	yes	no
247	bheight_baliR	384	112	104	104	1	20	yes	no
257	invh_mkNode	512	136	128	128	1	20	yes	no
262	invh_baliR	512	136	128	128	1	20	yes	no
269	invc_upd	640	96	96	96	1	20	yes	no
276	invh_upd	384	64	56	56	1	20	yes	no
291	invpst_upd	384	64	56	56	1	20	yes	no
309	invh_baldL_invc	768	184	176	176	1	20	yes	no
316	invh_baldL_Black	640	160	160	160	1	20	yes	no
320	invc_baldL	640	160	160	160	1	20	yes	no
324	invc2_baldL	512	136	128	128	1	20	yes	no
331	invh_baldR_invc	768	184	176	176	1	20	yes	no
336	inv_baldR	640	160	160	160	1	20	yes	no
340	inv2_baldR	512	136	128	128	1	20	yes	no
347	invh_combine	80	56	52	52	1	20	yes	no
356	inv_combine	100	68	68	68	1	20	yes	no
366	del_inv_invh	140	60	60	60	1	20	yes	no
398	invpst_combine	60	44	40	40	1	20	yes	no
403	invpst_del	60	28	24	24	1	20	yes	no

Table 5: Evaluation of smart_induct on Nearest_Neighbors.thy

line	theorem name	total	1st	2nd-a	2nd-b	nth	score	rule	arb
66	sqed_ge_0	40	32	27	27	1	20	yes	no
71	sqed_eq_0	40	32	27	27	1	20	yes	no
76	sqed_eq_0_rev	40	32	27	27	1	20	yes	no
81	sqed_com	40	32	27	27	2	20	yes	no
147	minimize_sqed	256	72	62	62	-	-	yes	yes
228	sorted_insort_sqed	128	32	32	32	8	18	no	no
285	sorted_sqed_last_take_mono	256	48	44	44	5	20	no	yes
292	sorted_sqed_last_insort_eq	128	32	32	32	4	20	no	no
321	mnn_length	3210	192	192	192	-	-	no	yes
328	mnn_length_gt_0	2080	160	160	160	7	20	no	yes
335	mnn_length_gt_eq_m	2080	119	119	119	7	20	no	yes
344	mnn_sorted	2080	160	160	160	5	20	no	yes
350	mnn_set	3210	192	192	192	-	-	no	yes
359	mnn_distinct	2080	160	160	160	7	20	no	yes
388	mnn_le_last_ms	3120	192	184	184	7	20	no	yes
431	mnn_sqed	4160	204	188	188	7	20	no	yes

Appendix B

Figure 2 illustrates the user-interface of `smart_induct`.

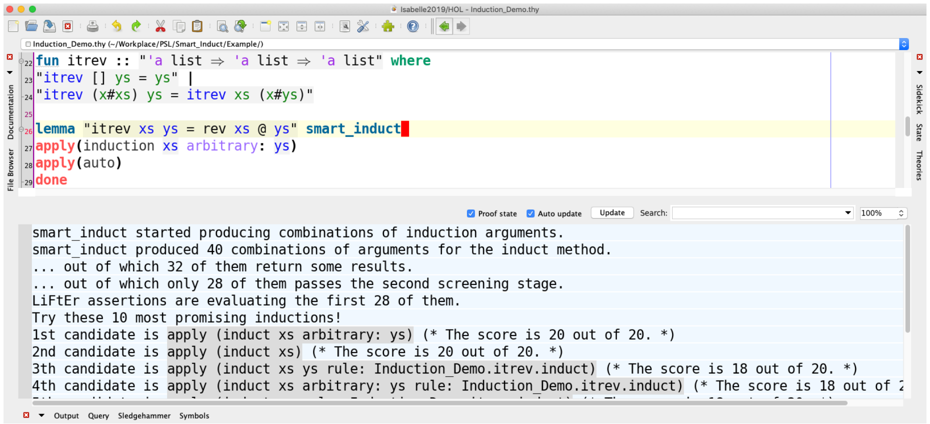

Fig. 2: A Screenshot of Isabelle/jEdit with `smart_induct`.