Learning Large Logic Programs By Going Beyond Entailment

Andrew Cropper¹ and Sebastijan Dumančić²
¹University of Oxford
²KU Leuven
andrew.cropper@cs.ox.ac.uk, sebastijan.dumancic@cs.kuleuven.be

Abstract
A major challenge in inductive logic programming (ILP) is learning large programs. We argue that a key limitation of existing systems is that they use entailment to guide the hypothesis search. This approach is limited because entailment is a binary decision: a hypothesis either entails an example or does not, and there is no intermediate position. To address this limitation, we go beyond entailment and use example-dependent loss functions to guide the search, where a hypothesis can partially cover an example. We implement our idea in Brute, a new ILP system which uses best-first search, guided by an example-dependent loss function, to incrementally build programs. Our experiments on three diverse program synthesis domains (robot planning, string transformations, and ASCII art), show that Brute can substantially outperform existing ILP systems, both in terms of predictive accuracies and learning times, and can learn programs 20 times larger than state-of-the-art systems.

1 Introduction
A major challenge in inductive logic programming (ILP) is learning large programs [Cropper et al., 2019a]. We argue that a key limitation of existing systems is that they use entailment to guide the hypothesis search [Muggleton, 1995; Blockeel and De Raedt, 1998; Srinivasan, 2001; Law et al., 2014; Cropper and Muggleton, 2016]. This approach is limited because entailment is a binary decision: a hypothesis either entails an example or does not, and there is no intermediate position.

To illustrate this limitation, imagine learning image transformation programs from input/output examples. Figure 1 shows a scenario where the goal is to learn a program to transform the corner squares from red to white. Suppose that an entailment-guided ILP system is evaluating two hypotheses $h_1$ and $h_2$ which generate the outputs $o_1$ and $o_2$ respectively shown in Figure 2. Although $o_1$ is clearly closer to the example output than $o_2$ (because only 1 pixel needs to change compared to 7 in $o_2$), the ILP system would deem the two hypotheses equal because neither entails the example, and would thus have no reason to prefer $h_1$ to $h_2$ during the search.

To address this limitation, we take inspiration from how humans write programs. To paraphrase Ellis et al. (2019), writing code is often a process of trial and error: write code, execute it, evaluate the output, and revise the code if necessary. Our approach allows an ILP system to perform similarly: build a program, execute it on example input to generate output, compare the output with the expected output, and revise the program if necessary.

To allow an ILP system to evaluate a program (i.e. to compare the generated output with the expected output), we give it domain-specific knowledge. Specifically, we use example-dependent loss functions: loss functions that consider information about the examples, other than whether they are entailed. For instance, in the image transformation problem, we could use Hamming distance to measure how close an output is to the desired one (how many pixels differ), which would allow an ILP system to prefer $h_1$ to $h_2$.

We claim that our approach can improve learning performance, especially when the target hypothesis is large. To support this claim, we make the following contributions:

- We describe Brute, a new ILP system (Section 3) which combines techniques from search (best-first search) and answer set programming (ASP) [Lifschitz, 2008] to...
learn programs with recursion and predicate invention [Muggleton et al., 2015].

- We evaluate Brute on three diverse program synthesis domains: robot planning, real-world string transformations, and a new problem of drawing ASCII art (Section 4). Our experiments show that Brute can substantially outperform existing ILP systems both in terms of predictive accuracies and learning times, and can learn programs 20 times larger than state-of-the-art systems.

2 Related Work

The goal of program synthesis is to automatically generate computer programs from input/output examples. The topic is considered the holy grail of AI [Gulwani et al., 2017; Singh and Kohli, 2017].

Neural approaches typically need lots of training data and large training times [Balog et al., 2017; Devlin et al., 2017; Ellis et al., 2018]. For instance, REPL [Ellis et al., 2019] combines an interpreter with AlphaGo [Silver et al., 2017] approach to learn programs. However, REPL takes multiple days to train on a single domain using one P100 GPU. By contrast, Brute can learn programs in under 60 seconds using a standard single-core computer. Another disadvantage of neural approaches is that they often require hand-crafted neural architectures for each domain. For instance, REPL needs a hand-crafted grammar, interpreter, and neural architecture for each domain. By contrast, Brute uses logic programming as a uniform representation for examples, background knowledge, hypotheses, and for itself (i.e. Brute is written in Prolog), so can be applied to arbitrary domains.

Classical ILP systems, such as FOIL [Quinlan, 1990], Progol [Muggleton, 1995], TILDE [Blockeel and De Raedt, 1998], and Aleph [Srinivasan, 2001], cannot (in general) learn recursive programs and so often struggle on synthesis problems. By contrast, Brute can learn recursive programs, so can learn programs that generalise to arbitrary sized inputs.

Most ILP systems use entailment-based loss functions (also called cost or scoring functions) to guide the hypothesis search, often in combination with the hypothesis size [Quinlan, 1990; Muggleton, 1995; Blockeel and De Raedt, 1998; Srinivasan, 2001; Law et al., 2014; Cropper and Muggleton, 2016; Kaminski et al., 2018]. For instance, Aleph’s default loss function is \( p - n \), where \( p \) and \( n \) are the number of positive (\( p \)) and negative (\( n \)) examples entailed by a clause. However, as our introductory image transformation example shows, entailment-based loss functions can be uninformative because entailment is a binary decision.

Metaopt [Cropper and Muggleton, 2019] and FastLAS [Law et al., 2020] both use domain-specific loss functions to find optimal hypotheses, such as the most efficient program [Cropper and Muggleton, 2019]. Brute also uses domain-specific loss functions (specifically example-dependent), not to learn optimal programs, but to instead search efficiently.

Most authors measure the size of a logic program as either the number of literals [Law et al., 2014] or clauses [Cropper et al., 2019b] in it. What defines a large program is unclear. Given \( n \) examples, ILP systems can learn programs with \( n \) clauses by simply memorising the examples. ILP systems can also learn clauses with many literals when the variable depth is small [Muggleton, 1995]. In contrast to most ILP systems, which focus on concept learning (i.e. classification), Brute focuses on learning recursive programs that compute something, i.e. program synthesis [Flener and Yilmaz, 1999]. In this area, Metagol [Cropper and Muggleton, 2016] is a state-of-the-art system, yet struggles to learn programs with more than 8 clauses (or approximately 24 literals) [Cropper et al., 2019a]. Our experiments show that Brute can learn programs 20 times larger than Metagol, whilst still maintaining the ability to generalise.

Brute works in two stages: invent and search. ILASP [Law et al., 2014] and \( \partial \) ILP [Evans and Grefenstette, 2018] also work in two stages. Both precompute a set of clauses and then find a subset of the clauses, where ILASP uses ASP and \( \partial \) ILP uses a neural network. Brute also finds a subset of clauses, but additionally finds how to compose the clauses to build new clauses.

3 Brute

Brute is a new ILP system which we intentionally designed to be simple to clearly demonstrate our idea. Given:

- positive (\( e^+ \)) and negative (\( e^- \)) examples formed of sets of dyadic atoms \( p(x_i, y_i) \) where \( p \) is the target predicate symbol and \( x_i \) and \( y_i \) are terms denoting input and output values respectively
- background knowledge \( bk \) described as a definite logic program
- an example-dependent loss function \( \mathcal{L} : \mathcal{E} \times \mathcal{E} \rightarrow \mathbb{R} \) where \( \mathcal{E} \) is the constant signature of \( e^+ \cup e^- \cup bk \)

Brute searches for a hypothesis (a definite logic program) \( h \) such that \( \forall e \in e^+, h \cup bk \models e \) and \( \forall e \in e^-, h \cup bk \models \neg e \).

Brute works in two stages: **invent** and **search**. In the invent stage, Brute invents library predicates, which it later uses to build a hypothesis. In the search stage, Brute searches to find which library predicates to add to a hypothesis and in which sequence to execute them. Algorithm 1 sketches the Brute algorithm\(^1\). We describe the two stages in detail.

3.1 Invent

Brute takes as input background knowledge which defines primitive predicates. In the invent stage, Brute uses the primitive predicates to invent library predicates. A library predicate is set of definite clauses defined by the same predicate symbol, similar to a predicate in a Prolog library, e.g. max_list/2. We call the set of library predicates the library.

To invent library predicates, Brute uses an ASP encoding to generate programs that compose the primitive predicates, and assigns each program a unique new predicate symbol. Because the set of such programs is infinite, Brute follows common convention [Srinivasan, 2001; Law et al., 2014;]

\(^1\)Brute is implemented in Prolog.
Algorithm 1 Brute

```python
def brute(e⁺, e⁻, bk, L):
    library = invent(bk)
    prog = search(e⁺, e⁻, bk, L, library)
    return prog

def search(e⁺, e⁻, bk, L, library):
    queue = empty_priority_queue()
    initial_spec = [(x, y) | p(x, y) ∈ e⁺]
    initial_hypothesis = []
    initial_state = (initial_spec, initial_hypothesis)
    initial_loss = \sum_{(x, y)\in initial_spec} L(x, y)
    queue.push(initial_loss, initial_state)
    while not queue.empty():
        loss, state = queue.pop()
        (spec, hypothesis) = state

        if loss == 0 and consistent(hypothesis, e⁺, bk):
            return hypothesis + induce_target(hypothesis)

        for library_predicate in library:
            new_spec = apply(library_predicate, bk, spec)
            new_hypothesis = hypothesis + library_predicate
            new_loss = \sum_{(x, y)\in new_spec} L(x, y)
            new_state = (new_spec, new_hypothesis)
            q.push(new_loss, new_state)

    return []
```

Brute does not invent the following predicate because it has a recursive clause without a base clause:

\[ f_5(A, B) \leftarrow right(A, C), f_5(C, B) \]

### 3.2 Search

In the search stage, Brute tries to build a hypothesis using the library predicates. To do so, Brute performs a best-first search [Russell and Norvig, 2010] guided by a given loss function.

A specification is a set of \((x, y)\) pairs denoting input \((x)\) and output \((y)\) values. Given \(n\) positive examples \(\{p(x_1, y_1), \ldots, p(x_n, y_n)\}\), the initial specification (line 8) is \(\{(x_1, y_1), \ldots, (x_n, y_n)\}\). The initial hypothesis is an empty list, denoted as \([]\) (line 9). A state is a \((\text{specification}, \text{hypothesis})\) pair. The initial state (line 10) is a pair of the initial specification and the initial hypothesis. The initial loss is the sum of the losses of the initial specification (line 11). Brute adds the initial loss and the initial state to a priority queue (line 12) and then performs a best-first search.

**Example 3.** Suppose we have the positive examples \(f(1, 4)\) and \(f(7, 10)\), the loss function \(L(x, y) = |x - y|\), and four library predicates:

\[
\begin{align*}
    f_1(A, B) & \leftarrow \text{at}_\text{end}(A), \text{draw}_\text{white}(A, B) \\
    f_2(A, B) & \leftarrow \text{at}_\text{end}(A), \text{draw}_\text{black}(A, B) \\
    f_3(A, B) & \leftarrow \text{draw}_\text{white}(A, C), \text{right}(C, D), f_2(D, B) \\
    f_4(A, B) & \leftarrow \text{succ}(A, C), \text{succ}(C, B)
\end{align*}
\]

The initial specification \(s\) is \(\{(1, 4), (7, 10)\}\), the initial state is \((s, [])\) and the initial loss is \(|1 - 4| + |7 - 10| = 6\). Because the loss is not zero, Brute searches for a better state.

To search for a better state, Brute applies each library predicate to every pair in the current specification to generate a new specification (line 22). For instance, applying the library predicate \(f_1(A, B) \leftarrow \text{succ}(A, B)\) to the specification pair \((1, 4)\) means to call \(f_1(1, B)\) to deduce a value for \(B\) (i.e. 2) to form the new specification pair \((2, 4)\). Formally:\n
**Definition 1 (Application).** Given background knowledge \(bk\), a library \(l\), a library predicate \(p/2\), and a specification pair \((x, y)\), an application forms a new specification pair \((z, y)\) where \(z\) is the computed answer for \(B\) in an SLD-refutation of \(bk \cup l \cup \{\leftarrow p(x, B)\}\).

**Example 4.** Applying the library predicate \(f_4(A, B) \leftarrow \text{succ}(A, C), \text{succ}(C, B)\) to the specification \(\{(1, 4), (7, 10)\}\) generates the new specification \(\{(3, 4), (9, 10)\}\) from the computed answers of the SLD-refutations of \(bk \cup l \cup \{\leftarrow f_4(1, B)\}\) and \(bk \cup l \cup \{\leftarrow f_4(7, B)\}\) respectively.

After generating a new specification, Brute adds the library predicate to the hypothesis (line 23), calculates the loss of the new specification (line 24), and adds the new state with the new loss to the priority queue (line 25).

**Example 5.** Continuing our running example, the state is \(\{(1, 4), (7, 10); []\}\) and the loss is \(|1 - 4| + |7 - 10| = 6\).
6. Brute can apply the library predicate \( f4(A, B) \leftarrow succ(A, C), succ(C, B) \) to generate the new state:

\[
\{(3, 4), (9, 10), [f4(A, B) \leftarrow succ(A, C), succ(C, B)]\}
\]

The loss of this new state is \( |3 - 4| + |9 - 10| = 2 \), so Brute again searches for a better state.

The search continues until either (1) there are no more states to consider, or (2) the loss at the current state is 0 and the hypothesis does not entail any negative examples (line 18). When the search finishes, Brute induces a target-clause from the sequence of used library predicates and adds it to the hypothesis, which it returns (line 19). A target clause defines the target predicate \( p \) and is of the form \( p(S_1, S_{m+1}) \leftarrow p_1(S_1, S_2), p_2(S_2, S_3), \ldots, p_m(S_m, S_{m+1}) \) where each \( p_i \) is the \( i \)th clause in the hypothesis and each \( S_i \) is a variable.

**Example 6.** To finish our running example, our state is:

\[
\{(3, 4), (9, 10), [f4(A, B) \leftarrow succ(A, C), succ(C, B)]\}
\]

And our loss is \( |3 - 4| + |9 - 10| = 2 \). Brute can apply the predicate \( f1(A, B) \leftarrow succ(A, B) \) to generate the new state:

\[
\{(4, 4), (10, 10), [f4(A, B) \leftarrow succ(A, C), succ(C, B)),
\left\{f1(A, B) \leftarrow succ(A, B)\right\}\}
\]

The loss is now 0, so the search stops. Brute then induces a target clause from the sequence of library predicates and adds it to the hypothesis to form:

\[
\begin{align*}
& f(A, B) \leftarrow f4(A, C), f1(C, B) \\
& f4(A, B) \leftarrow succ(A, C), succ(C, B) \\
& f1(A, B) \leftarrow succ(A, B)
\end{align*}
\]

### 4 Experiments

We claim that example-dependent loss functions can improve performance. Our experiments therefore aim to answer the question:

**Q1** Can example-dependent loss functions outperform entailment-based loss functions?

To answer Q1, we compare Brute against Brute\(_m\), a variant which mimics an entailment-based approach using the loss function:

\[
\mathcal{L}(x, y) = \begin{cases} 
0 & \text{if } x = y \\ 
1 & \text{otherwise}
\end{cases}
\]

In other words, if the output from the hypothesis exactly matches the desired output then there is no loss; otherwise the loss is 1.

We also claim that Brute can outperform existing ILP systems, especially when the target hypothesis is large, because it uses example-dependent loss functions to guide the search. Our experiments therefore aim to answer the question:

**Q2** Can Brute outperform state-of-the-art ILP systems?

To answer Q2, we compare Brute against Metagol, a state-of-the-art ILP system, which can learn recursive programs.

To answer questions Q1 and Q2, we consider three diverse domains: robot planning, string transformations, and drawing ASCII art.

### Experimental Settings

In each experiment, we enforce a learning timeout of 60 seconds per task. We repeat each experiment 10 times and plot 95% confidence intervals. We use only positive training examples. We describe below the system settings used in the experiments.

**Brute** We restrict Brute to only invent library predicates with at most two clauses, where each clause has at most three variables and at most two body literals.

**Metagol** Metagol uses metarules [Muggleton et al., 2015], higher-order Horn clauses, to guide the proof search. We provide Metagol with the `ident`, `precon`, `postcon`, `chain`, and `tailrec` metarules, which are commonly used in the Metagol literature. We also force Metagol to learn functional logic programs [Lin et al., 2014], which ensures that for any induced program and for any input value there is exactly one output value. This constraint helps Metagol when learning from only positive examples because otherwise it tends to learn overly general recursive programs.

### 4.1 Experiment 1 - Robot Planning

Our first experiment is on learning robot plans, a domain often used to evaluate Metagol [Cropper, 2019; Cropper and Muggleton, 2019].

**Materials**

A robot and a ball are in a \( n^2 \) grid. An example is an atom \( f(x, y) \), where \( f \) is the target predicate and \( x \) and \( y \) are initial and final states respectively. A state describes the positions of the robot and the ball, and whether the robot is holding the ball. The task is to learn to transform the initial state to the final state. We provide as background knowledge the dyadic predicates `up`, `down`, `right`, `left`, `grab`, `drop`, and the monadic predicates `at_top`, `at_bottom`, `at_left`, `at_right`. Brute uses a Manhattan distance loss function.

**Method**

For each \( n \) in \{2, 4, 6, .., 10\}, we generate a single training example for a \( n^2 \) world, i.e. this is a one-shot learning task. As the grid size grows, so should the size of the target hypothesis. We measure the percentage of tasks solved (i.e. where an induced hypothesis entails all the positive and no negative examples) and learning times.

**Results**

Figure 3 shows that for a \( 2^2 \) grid, all three systems solve 100% of the tasks. However, as the grid size grows, the performance of Metagol quickly degrades. For a \( 6^2 \) grid, Metagol only solves 8% of the tasks. By contrast, for a \( 10^2 \) grid, Brute solves 67% of the tasks. Figure 3 also shows that Brute learns programs substantially quicker than Metagol. Figure 3 shows that Brute outperforms Brute\(_m\) for a \( 6^2 \) grid or bigger, both in terms of learning times and percentage of tasks solved. These results suggest that the answers to Q1 and Q2 are both yes.

Metagol struggles on larger grids because it must learn larger programs. The biggest program learned by Metagol has 5 clauses and 15 literals. By contrast, the biggest program learned by Brute has 18 clauses and 69 literals. Brute
starts to struggle on larger grids because of local optima. For instance, suppose that the robot starts at position 1/1, the ball at position 3/1, and the goal is to move the ball to position 1/3. In this scenario, Brute first tries to find a program to move the robot to position 1/3 to minimise the loss function (Manhattan distance). Brute then explores the space around 1/3, before eventually finding the ball at position 3/1, at which point it quickly finds the target hypothesis. Despite occasional local optima, Brute substantially outperforms both Brute$_p$ and Metagol.

4.2 Experiment 2 - String Transformations

Our second experiment is on real-world string transformations, a domain often used to evaluate program synthesis systems [Lin et al., 2014; Balog et al., 2017; Ellis et al., 2019].

Materials

We use 130 string transformation tasks from [Cropper, 2019]. Each task has 10 examples. An example is an atom $f(x, y)$ where $f$ is the task name and $x$ and $y$ are input and output states respectively. A state is a $(s, p)$ pair, where $s$ is the string and $p$ is a cursor pointing to a specific position in the string. Figure 4 shows a task with three examples, where the goal is to extract the first three letters of the month and make them uppercase.

Method

For each task and for each $n$ in $\{1, 3, 5, 7, 9\}$, we sample uniformly without replacement $n$ examples as training examples and use the other $10 - n$ examples as testing examples. We measure predictive accuracies and learning times.

Results

Figure 5 shows that Brute$_p$ slightly outperforms Brute in all cases, which contradicts the results from Experiment 1. Figure 5 also shows that Brute significantly outperforms Metagol in all cases, which again suggests that the answer to Q2 is yes.

Brute sometimes performs worse than Brute$_p$ because of local optima. For instance, when trying to learn a program that takes a string and returns the first letter made uppercase, e.g. “james” $\rightarrow$ “J”, Brute first uses an invented recursive predicate to delete all but the last character from the input to minimise the loss (edit distance), e.g. “james” $\rightarrow$ “s” (where the loss is only 1). Brute then searches in this region of the search space, but is unable to find the target hypothesis in the allocated time (60 seconds).

Brute typically outperforms Brute$_p$ and Metagol on tasks that require larger programs. For instance, consider trying to learn a program to extract the number of logical inferences per second (Lips) from the output of time/1 in Prolog, e.g. “16,079 inferences, 0.003 CPU in 0.003 seconds (95% CPU, 5842660 Lips)” $\rightarrow$ “5842660”. For this task, Brute learns the general program shown in Figure 6, which contains four invented recursive predicates. Figure 7 shows the execution trace of this program on the aforementioned example.

Method

For each task and for each $n$ in $\{1, 3, 5, 7, 9\}$, we sample uniformly without replacement $n$ examples as training examples and use the other $10 - n$ examples as testing examples. We measure predictive accuracies and learning times.

4.3 Experiment 3 - ASCII Art

Our third experiment is on a new problem of learning to draw ASCII art.
An image is the pixel representation of an ASCII string. As the string grows, so should the size of the target hypothesis. We use the text2art library with the font 3x5. Figure 8 shows an example image for the string “IJCAI”. An example is an atom \( f(x, y) \), where \( f \) is the target predicate and \( x \) and \( y \) are input and output states respectively. A state is a \((i, p)\) pair, where \( i \) is the image, represented as a list, and \( p \) is a cursor pointing to a specific pixel in the image. We provide as background knowledge the dyadic predicates up, down, right, left, draw1, draw0, and the monadic predicates at_top, at_bottom, at_left, at_right. The predicates draw0 and draw1 manipulate the image. The rest manipulate the cursor. Brute uses a Hamming distance loss function.

Figure 8: Example ASCII image.

Method
For each \( n \) in \( \{1, 2, \ldots, 5\} \), we sample uniformly at random with replacement an ASCII string of length \( n \). As the string grows, so should the size of the target hypothesis. We use the text2art library to transform a string to a pixel representation which forms our output image. We use the empty image as the input image. We measure the percentage of tasks solved and learning times.

Results
Figure 9 shows that Metagol and Brute cannot learn any solutions. By contrast, Brute learns programs for 86% of the images with 3 characters, and still manages to learn programs for 25% of the images with 5 characters. The largest program learned by Brute has 79 clauses and 294 literals. These results again suggest that the answer to questions Q1 and Q2 is yes.

Materials
An image is the pixel representation of an ASCII string according to the text2art library with the font 3x5. Figure 8 shows an example image for the string “IJCAI”. An example is an atom \( f(x, y) \), where \( f \) is the target predicate and \( x \) and \( y \) are input and output states respectively. A state is a \((i, p)\) pair, where \( i \) is the image, represented as a list, and \( p \) is a cursor pointing to a specific pixel in the image. We provide as background knowledge the dyadic predicates up, down, right, left, draw1, draw0, and the monadic predicates at_top, at_bottom, at_left, at_right. The predicates draw0 and draw1 manipulate the image. The rest manipulate the cursor. Brute uses a Hamming distance loss function.

Figure 7: Execution trace of the program from Figure 6 on the example “16,079 inferences, 0.003 CPU in 0.003 seconds (95% CPU, 5842660 Lips)” \( \downarrow f_0 \)

“PU in 0.003 seconds (95% CPU, 5842660 Lips)” \( \downarrow f_1 \)

“.003 seconds (95% CPU, 5842660 Lips)” \( \downarrow f_0 \)

“PU, 5842660 Lips)” \( \downarrow f_2 \)

“5842660 Lips)” \( \downarrow f_3 \)

“5842660 Lips” \( \downarrow f_2 \)

“5842660”

Figure 7: Execution trace of the program from Figure 6 on the example “16,079 inferences, 0.003 CPU in 0.003 seconds (95% CPU, 5842660 Lips)” \( \rightarrow \) “5842660”.

5 Conclusions and Limitations
A major challenge in ILP (and program synthesis in general) is learning large programs. To tackle this problem, we have proposed an approach inspired by how humans write programs. In our approach, an ILP system builds a program, executes it on some example input to generate output, compares the output with the expected output, and revises the program if needed. To evaluate a hypothesis, we use example-dependent loss functions. We implemented our idea in Brute, a new ILP which first invents a library of predicates, including recursive predicates, and then performs a best-first search informed by a given loss function to build a hypothesis using the library predicates.

Our experiments on three diverse program synthesis domains (robot planning, real-world string transformations, and a new problem of drawing ASCII art), show that (1) example-dependent loss functions can outperform entailment-based loss functions, and (2) Brute can outperform Metagol, a state-of-the-art ILP system. In our experiments, given only 60 seconds to learn a program, the largest program learned by Metagol had 5 clauses and 15 literals. By contrast, the largest program learned by Brute had 79 clauses and 294 literals.

Limitations and Future Work
Generality In contrast to classical ILP systems, which focus on concept learning (i.e. classification), Brute focuses on learning recursive programs from input/output examples. Brute cannot therefore currently solve some classical ILP problems, such as Mutagenesis [Srinivasan et al., 1994], because the examples in these domains are not dyadic. To address this limitation, we could represent classification tasks as program synthesis tasks, where the output is the label.

Search To clearly demonstrate our idea, we intentionally designed Brute to be simple in two ways (1) brute-force inventing library predicates, and (2) using simple best-first search. As our experiments show, our intentionally simple approach can drastically outperform existing systems. To further improve performance, we want to (1) dynamically invent library predicates during the search to reduce the branching factor, and (2) use better search techniques, such as A* or iterative budgeted exponential search [Helmert et al., 2019].
Loss functions  Brute uses example-dependent (domain-specific) loss functions to guide the search, which our experiments show are important for high performance. By contrast, most ILP systems use general entailment-based loss functions. An important direction for future work is to bridge the gap between the two approaches. An exciting idea is to learn suitable loss functions for a given problem through meta-learning [Thrun and Pratt, 2012].

Summary
To conclude, we think that Brute is an important contribution to ILP and program synthesis, and that the ideas introduced in this paper open up new and exciting research opportunities for learning large programs.

References


