Learning Logic Programs Through Divide, Constrain, and Conquer

Andrew Cropper

University of Oxford
andrew.cropper@cs.ox.ac.uk

Abstract

We introduce an inductive logic programming approach that combines classical divide-and-conquer search with modern constraint-driven search. Our anytime approach can learn optimal, recursive, and large programs and supports predicate invention. Our experiments on three domains (classification, inductive general game playing, and program synthesis) show that our approach can increase predictive accuracies and reduce learning times.

1 Introduction

Inductive logic programming (ILP) (Muggleton 1991) is a form of machine learning. Given positive and negative examples and background knowledge (BK), the ILP problem is to find a set of rules (a *hypothesis*) which with the BK entails all the positive and none of the negative examples.

The fundamental challenge in ILP is to efficiently search a large hypothesis space (the set of all hypotheses) for a *solution* (a hypothesis that correctly generalises the examples). Divide-and-conquer (D&C) approaches, such as TILDE (Blockeel and Raedt 1998), divide the examples into disjoint sets and then search for a hypothesis for each set, similar to decision tree learners (Quinlan 1986). Separate-and-conquer (S&C) approaches, such as PROGOL (Muggleton 1995) and ALEPH (Srinivasan 2001), search for a hypothesis that generalises a subset of the examples, separate these examples, and then search for more rules to add to the hypothesis to generalise the remaining examples.

Although powerful, D&C and S&C approaches struggle to perform predicate invention (Stahl 1995) and learn recursive and optimal programs (Cropper et al. 2022), partly because they only consider subsets of the examples. For instance, PROGOL and ALEPH need examples of both the base and inductive cases (in that order) to learn a recursive program. To overcome these limitations, many modern systems learn from all examples simultaneously. Modern systems also use powerful constraint solvers, such as answer set programming (ASP) solvers (Gebser et al. 2014), to search for a hypothesis. For instance, ASPAL (Corapi, Russo, and Lupu 2011) precomputes the set of all possible rules that may appear in a hypothesis and uses an ASP solver to find a subset that generalises the examples. Many modern systems can learn optimal and recursive programs but struggle to learn large programs. For instance, METAGOL (Muggleton, Lin, and Tamaddoni-Nezhad 2015) struggles to learn programs with more than six rules and ILASP3 (Law 2018) struggles to learn rules with more than a few body literals.

To address these limitations, we introduce an approach that combines classical D&C search with modern constraint-driven search. As with D&C approaches, we divide the examples into disjoint subsets and induce separate hypotheses for them. Specifically, we first learn a hypothesis for each positive example. Each hypothesis will likely be too specific. We therefore search again for a hypothesis that generalises pairs of examples. These new hypotheses will again likely be too specific but should generalise better than previous ones. We repeat this process each time increasing the partition size until it matches or exceeds the number of positive examples.

The aforementioned approach on its own is pointless as it simply involves repeated search. The key idea of our approach is to *reuse* knowledge discovered when solving smaller tasks to help solve larger tasks. For instance, when searching for a hypothesis for a single positive example, if we discover a hypothesis that incorrectly entails a negative example then any more general hypothesis will also entail it. We can therefore ignore all generalisations of the hypothesis in subsequent iterations.

To realise our idea, we build on the constraint-driven *learning from failures* (LFF) (Cropper and Morel 2021a,b) ILP approach. The goal of LFF is to accumulate constraints to restrict the hypothesis space. In our approach, we accumulate constraints during both the divide and conquer steps. In other words, we reuse constraints learned during one iteration in subsequent iterations. We call our approach *divide*, *constrain*, and *conquer* (DCC).

To illustrate our approach, suppose we want to learn the following program h_1 to find odd elements in a list:

$$f(A,B) \leftarrow head(A,B), odd(B)$$
$$f(A,B) \leftarrow head(A,B), even(B), tail(A,C), f(C,B)$$

Suppose we have two positive examples $e_1 = f([4,3,4,6],3)$ and $e_2 = f([2,2,9,4,8,10],9)$ which correspond to odd elements in the second and third positions of a list respectively. Also assume we have suitable negative examples and that we restrict hypotheses to definite programs (Lloyd 2012). Given

these examples, our approach first learns a hypothesis (in this case a single rule) for each example:

$r_1 = f(A,B) \leftarrow tail(A,C), head(C,B), odd(B)$
$r_2 = f(A,B) \leftarrow tail(A,C), tail(C,D), head(D,B), odd(B)$

These rules are too specific, i.e. they do not generalise. Rather than stopping at this point, our approach searches again for a hypothesis h_2 that entails the larger chunk $\{e_1, e_2\}$. Rather than blindly searching again, our approach reuses knowledge from the first iteration to restrict the hypothesis space. For instance, let $|h|$ denote the number of literals in the hypothesis h and assume that r_1 and r_2 are the smallest (optimal) solutions for e_1 and e_2 respectively. Since r_1 and r_2 are optimal, the minimum size of h_2 is $max(|\{r_1\}|, |\{r_2\}|)$. Likewise, since $\{r_1, r_2\} \models \{e_1, e_2\}$, we can bound the maximum size of h_2 as $|\{r_1, r_2\}|$. In this scenario of finding odd elements in a list, we can bound the size of h_2 as $5 \leq |h_2| < 9$, greatly reducing the hypothesis space. We can also restrict the hypothesis space using other knowledge. For instance, before searching for h_2, we can try r_1 and r_2 on $\{e_1, e_2\}$. As neither r_1 nor r_2 alone generalises both examples, h_2 cannot be a specialisation of r_1 nor r_2, so we can prune all specialisations of both rules. As we experimentally show, this reuse of knowledge, i.e. the constrain step, is important for good learning performance. Given this constrained space, our approach searches again and finds the optimal solution h_1.

Our motivation for using a D&C approach is to reduce search complexity by decomposing a learning task into smaller tasks that can be solved separately. For instance, suppose we have 10 positive examples and that 6 require a hypothesis with 7 literals; 2 require 8 literals; and 2 require 9 literals, i.e. the solution for all the examples has 24 literals. For simplicity, consider a generate-and-test approach that enumerates all hypotheses of increasing size. With such an approach, the search complexity $sc(h)$ of finding the hypothesis h is $sc(h) = c^{|h|}$ where c denotes the number of possible literals allowed in a hypothesis. In this scenario, the complexity of finding the whole solution is c^{24}. By contrast, with a D&C approach, we can find the individual hypotheses with the much lower cost of $c^7 + c^8 + c^9$. Thus, our main claim is that DCC can reduce search complexity and thus improve learning performance.

Overall, our contributions are:

- We introduce a divide, constrain, and conquer (DCC) ILP approach. This anytime approach can learn optimal, recursive, and large programs and perform predicate invention.

- We experimentally show on three domains (classification, inductive general game playing, and program synthesis) that (i) our approach can substantially improve predictive accuracies and reduce learning times, (ii) reusing learned knowledge is vital for good learning performance, and (iii) DCC can outperform other ILP systems.

2 Related Work

TILDE is a D&C approach. TILDE behaves similarly to the decision tree learning algorithm C4.5 (Quinlan 1993). To find a hypothesis TILDE employs a D&C strategy by recursively dividing the examples into disjoint subsets. TILDE differs from C4.5 by how it generates candidate splits to partition the examples. C4.5 generates candidates as attribute-value pairs. By contrast, TILDE uses conjunctions of literals (i.e. clauses/rules). TILDE can learn large programs and can scale to large datasets. However, it has several limitations, notably an inability to learn recursive programs, no predicate invention, and difficulty learning from small numbers of examples.

Progol is a S&C approach that has inspired many other approaches (Ray 2009; Ahlgren and Yuen 2013), notably ALEPH. Starting with an empty hypothesis, Progol picks an uncovered positive example to generalise. To generalise an example, Progol uses mode declarations to build the *bottom clause* (Muggleton 1995), the logically most-specific clause that explains the example. The bottom clause bounds the search from below (the bottom clause) and above (the empty set). Progol then uses an A* algorithm to generalise the bottom clause in a top-down (general-to-specific) manner and uses the other examples to guide the search. Progol struggles to learn recursive and optimal programs and does not support predicate invention. Note that Progol variants, such as ALEPH and ATOM (Ahlgren and Yuen 2013), have the same limitations.

Many recent ILP systems are *meta-level* systems (Cropper et al. 2022). These approaches encode the ILP problem as a meta-level logic program, i.e. a program that reasons about programs. Meta-level approaches often delegate the search for a solution to an off-the-shelf solver (Corapi, Russo, and Lupu 2011; Cropper and Muggleton 2016; Law 2018; Kaminski, Eiter, and Inoue 2018; Evans and Grefenstette 2018; Evans et al. 2021) after which the meta-level solution is translated back to a standard solution for the ILP task. For instance, ASPAL translates an ILP task into a meta-level ASP program that describes every example and every possible rule in the hypothesis space. ASPAL then delegates the search to an ASP system to find a subset of the rules that covers all the positive but none of the negative examples. Meta-level approaches can more easily learn recursive programs and optimal programs.

A major issue with meta-level approaches is scalability. For instance, ASPAL, HEXMIL (Kaminski, Eiter, and Inoue 2018), and ILASP3 (Law 2018) all first pre-compute every possible rule in the hypothesis space which they pass to an ASP solver. This approach scales well when solutions require many rules with few body literals. However, this approach does not scale well when solutions require rules with many body literals (Cropper and Morel 2021a), since there are exponentially more rules given more body literals.

To improve the scalability of meta-level systems, POPPER (Cropper and Morel 2021a,b) does not precompute every possible rule in the hypothesis space. Instead, POPPER lazily generates rules. The key idea of POPPER is to discover constraints from smaller rules (and hypotheses) to rule out larger rules. POPPER can learn optimal and recursive programs and perform predicate invention. However, POPPER searches for a single solution for all the examples and struggles to learn solutions with many literals. As we experimentally demonstrate, our DCC approach can substantially outperform POPPER and other systems.

We have said that some ILP approaches struggle to learn

large programs. However, what constitutes a large program is unclear. Most authors measure the size of a logic program as either the number of literals (Law 2018) or rules (Muggleton, Lin, and Tamaddoni-Nezhad 2015) in it. However, these two metrics are too simple. For instance, many approaches can easily learn programs with lots of clauses by simply memorising the examples. Likewise, approaches based on *inverse entailment* (Muggleton 1995) can easily learn programs with lots of literals by simply returning the *bottom clause*. In this paper, we do not formally define what constitutes a large program. By *large*, we informally mean programs with reasonably large numbers of rules, variables, and literals.

3 Problem Setting

Our problem setting is the *learning from failures* (LFF) (Cropper and Morel 2021a) setting. LFF uses *hypothesis constraints* to restrict the hypothesis space. Let $\mathcal{L}$ be a language that defines hypotheses, i.e. a meta-language. For instance, consider a meta-language formed of two literals *h_lit/4* and *b_lit/4* which represent *head* and *body* literals respectively. With this language, we can denote the clause *last(A,B) ← tail(A,C), head(C,B)* as the set of literals {*h_lit(0,last,2,(0,1)), b_lit(0,tail,2,(0,2)), b_lit(0,head,2,(2,1))*}. The first argument of each literal is the clause index, the second is the predicate symbol, the third is the arity, and the fourth is the literal variables, where *0* represents *A*, *1* represents *B*, etc. A *hypothesis constraint* is a constraint (a headless rule) expressed in $\mathcal{L}$. Let C be a set of hypothesis constraints written in a language $\mathcal{L}$. A set of definite clauses H is *consistent* with C if, when written in $\mathcal{L}$, H does not violate any constraint in C. For instance, the constraint ← *h_lit(0,last,2,(0,1)), b_lit(0,last,2,(1,0))* would be violated by the definite clause *last(A,B) ← last(B,A)*. We denote as $\mathcal{H}_C$ the subset of the hypothesis space $\mathcal{H}$ which does not violate any constraint in C.

We define the LFF problem:

Definition 1 (**LFF input**). The *LFF* input is a tuple $(E^+, E^-, B, \mathcal{H}, C)$ where E^+ and E^- are sets of ground atoms denoting positive and negative examples respectively; B is a definite program denoting background knowledge; $\mathcal{H}$ is a hypothesis space, and C is a set of hypothesis constraints.

We define a LFF solution:

Definition 2 (**LFF solution**). Given an input tuple $(E^+, E^-, B, \mathcal{H}, C)$, a hypothesis $H \in \mathcal{H}_C$ is a *solution* when H is *complete* ($\forall e \in E^+, \ B \cup H \models e$) and *consistent* ($\forall e \in E^-, \ B \cup H \not\models e$).

If a hypothesis is not a solution then it is a *failure*. A hypothesis is *incomplete* when $\exists e \in E^+, \ H \cup B \not\models e$. A hypothesis is *inconsistent* when $\exists e \in E^-, \ H \cup B \models e$. A hypothesis is *totally incomplete* when $\forall e \in E^+, \ H \cup B \not\models e$.

Let $cost : \mathcal{H} \mapsto R$ be an arbitrary cost function that measures the cost of a hypothesis. We define an *optimal* solution:

Definition 3 (**Optimal solution**). Given an input tuple $(E^+, E^-, B, \mathcal{H}, C)$, a hypothesis $H \in \mathcal{H}_C$ is *optimal* when (i) H is a solution, and (ii) $\forall H' \in \mathcal{H}_C$, where H' is a solution, $cost(H) \leq cost(H')$.

In this paper, our cost function is the number of literals in the hypothesis H.

Constraints. The goal of an LFF learner is to learn hypothesis constraints from failed hypotheses. Cropper and Morel (2021a,b) introduce hypothesis constraints based on subsumption (Plotkin 1971). A clause C_1 *subsumes* a clause C_2 ($C_1 \preceq C_2$) if and only if there exists a substitution θ such that $C_1\theta \subseteq C_2$. A clausal theory T_1 subsumes a clausal theory T_2 ($T_1 \preceq T_2$) if and only if $\forall C_2 \in T_2, \exists C_1 \in T_1$ such that C_1 subsumes C_2. A clausal theory T_1 is a *specialisation* of a clausal theory T_2 if and only if $T_2 \preceq T_1$. A clausal theory T_1 is a *generalisation* of a clausal theory T_2 if and only if $T_1 \preceq T_2$. If a hypothesis H is incomplete, a *specialisation* constraint prunes specialisations of H, as they are guaranteed to also be incomplete. If a hypothesis H is inconsistent, a *generalisation* constraint prunes generalisations of H, as they are guaranteed to be inconsistent as well. If a hypothesis H is totally incomplete, a *redundancy* constraint prunes hypotheses that contain a specialisation of H as a subset.

4 DCC Algorithm

We now describe our DCC algorithm. We first briefly describe POPPER, which we use as our underlying search algorithm.

4.1 POPPER

Algorithm 1 shows the POPPER algorithm, which solves the LFF problem (Definition 1). POPPER takes as input background knowledge (bk), positive (pos) and negative (neg) examples, a set of hypothesis constraints (in_cons), and lower (min_m) and upper (max_m) bounds on hypothesis sizes. POPPER uses a *generate*, *test*, and *constrain* loop to find a solution.

POPPER starts with a ASP program $\mathcal{P}$ (hidden in the generate function) whose models correspond to hypotheses (definite programs). POPPER augments $\mathcal{P}$ with ASP constraints to eliminate models and thus prune hypotheses. In the generate stage (line 5), POPPER uses Clingo (Gebser et al. 2014), an ASP system, to search for a model (a hypothesis) of $\mathcal{P}$. A constraint ensures that the hypothesis has exactly m literals. If a model is found, POPPER converts it to a hypothesis (h). Otherwise; POPPER increments the hypothesis size (line 7) and loops again.

If there is a hypothesis then in the test stage (line 9), POPPER tests it on the given training examples. If a hypothesis fails, i.e. is *incomplete* or *inconsistent*, then in the constrain stage (line 12), POPPER deduces hypothesis constraints (represented as ASP constraints) from the failure which it adds to the set of constraints, which are in turn added to $\mathcal{P}$ to prune models and thus restrict the hypothesis space. For instance, if a hypothesis is incomplete, i.e. does not entail all the positive examples, then POPPER builds a specialisation constraint to prune hypotheses that are logically more specific.

To find an optimal solution (i.e. one with the minimum number of literals), POPPER progressively increases the number of literals allowed in a hypothesis when the hypothesis space is empty at a certain size (e.g. when $\mathcal{P}$ has no more

Algorithm 1: POPPER

```
1  def popper(bk, pos, neg, in_cons, min_m, max_m):
2    cons = in_cons
3    m = min_m
4    while m ≤ max_m:
5      h = generate(cons, m)
6      if h == UNSAT:
7        m += 1
8      else:
9        outcome = test(pos, neg, bk, h)
10       if outcome == (COMPLETE, CONSISTENT)
11         return h, cons
12       cons += constrain(h, outcome)
13   return {}, cons
```

Algorithm 2: DCC

```
1  def dcc(bk, pos, neg):
2    cons = {}
3    k = 1
4    best_h, score = None, 0
5    all_chunks = {{x} | x in pos}
6    exs_h = {}
7    while k ≤ |all_chunks|:
8      iteration_hs = {}
9      for chunk in divide(all_chunks, k):
10       h = lazy_check(bk, chunk, iteration_hs)
11       if h == None:
12         chunk_cons = filter_c(cons, chunk)
13         min_m, max_m = calc_bounds(exs_h, chunk)
14         h, new_cons = popper(bk, chunk, neg,
                          chunk_cons, min_m, max_m)
15         cons += new_cons
16       if h != None:
17         iteration_hs += h
18         exs_hs = update(exs_hs, h, chunk)
19     iteration_h = union(iteration_hs)
20     h_score = test(pos, neg, bk, iteration_h)
21     if h_score > best_score:
22       best_h, best_score = iteration_h, h_score
23     all_chunks = compress(iteration_hs,
                      all_chunks)
24     k += k
25   return best_h
```

models). This loop repeats until either (i) POPPER finds an optimal solution, or (ii) there are no more hypotheses to test.

4.2 DCC

Algorithm 2 shows the DCC algorithm. Before describing it in detail, we describe it at a high-level. The idea is to divide the positive examples into *chunks* of size k. In the first iteration $k = 1$, each example is in its own chunk. DCC enumerates the chunks and calls POPPER to find a hypothesis for the chunk examples and *all* the negative examples. After enumerating all the chunks, DCC forms an *iteration* hypothesis as the union of all the chunk hypotheses. DCC then doubles the chunk size and repeats the process until the chunk size exceeds the number of examples. Without optimisations, DCC performs $\ln n$ iterations where n is the number of positive examples.

We now describe DCC in detail. DCC takes as input background knowledge (bk), and positive (pos) and negative (neg) examples. It maintains a set of constraints (cons) that is initially empty (line 2). DCC divides the positive examples into *chunks*. In the first iteration, each positive example is in its own chunk (line 5). The while loop in Algorithm 2 builds a hypothesis for a given chunk size. Line 8 creates an empty hypothesis (iteration_hs) for the current chunk size. DCC divides the chunks into smaller chunks (chunk) of size k and enumerates them.

In line 10 DCC calls the function lazy_check. We delay discussion of this function until we have described the main DCC loop but at a high-level this function tries to find an already discovered hypothesis that covers the current chunk. The purpose is to reduce the number of calls to POPPER.

Ignoring the lazy check, DCC tries to find a hypothesis for this chunk. The first step (line 12) selects only the relevant constraints for the examples in the chunk (chunk_cons). All generalisation constraints are selected. Specialisation constraints are only selected if they hold for at least one example in the chunk. Redundancy constraints are selected only if they hold for all examples in the chunk. The second step (line 13) deduces hypothesis size bounds for the subsequent POPPER search. The minimum size is the largest best solution for each chunk example. The maximum size is the size of the union.

DCC calls POPPER with the filtered constraints and hypothesis size bounds. POPPER returns a hypothesis (h) that

entails the chunk examples (if one exists) and a set of new constraints (new_cons). DCC updates its constraints (line 15) with the new ones. If a hypothesis is found, DCC adds it to the iteration hypothesis (line 17) and updates a hash table that maintains the last hypothesis for each example (line 18).

After passing through all the chunks, DCC forms a single iteration hypothesis as the union of all the hypotheses (line 19). DCC calculates the score of this iteration hypothesis on *all* the examples. We calculate the score as the number of correctly generalised examples (true positives + true negatives). In future work, we will explore alternative scoring functions, such as those that minimise description length (Rissanen 1978). If the score improves on the best score, DCC updates the best hypothesis (line 22). Line 22 *compresses* the chunks. We delay discussion of this function until we have described the main DCC loop but at a high-level this function tries to merge examples to reduce the number of iterations. After enumerating all the chunks, DCC doubles the chunk size (line 24) and repeats the process. Once the loop has finished, DCC returns the best hypothesis.

Laziness. The goal of *laziness* is to reduce the number of calls to POPPER. Suppose we have n chunks of examples $e_1, e_2, \ldots, e_n$ and that during the for loop in Algorithm 2 we find the solution h_1 for e_1. We now want to find a solution h_2 for e_2. Suppose that h_1 is a solution for e_2. Then do we need to search for h_2? On the one hand, h_1 may be sub-optimal in that there may be a smaller hypothesis that entails e_2. On the other hand, since we need h_1 (or a generalisation of it) to entail e_1, we may as well reuse h_1 as it requires adding no more literals to our iteration hypothesis. Laziness generalises

to all previously found hypotheses. Without laziness, for a chunk with n examples, DCC requires in the best- and worst-cases n calls to POPPER. Laziness reduces the best-case to 1. Moreover, we can deduce additional constraints from these lazy checks. Suppose that h_1 does not entail e_2. We can therefore rule out all specialisations of h_1 when searching for h_2 to further restrict the hypothesis space. In Section 5, we experimentally evaluate the impact of laziness on learning performance.

Compression. Given n positive examples $e_1, e_2, \ldots, e_n$, Algorithm 2 searches for solutions $h_1, h_2, \ldots, h_n$ respectively. It then increases the chunk size and searches for solutions $h_1', h_2', \ldots, h_{n/2}'$ for $\{e_1, e_2\}, \{e_3, e_4\}, \ldots, \{e_{n-1}, e_n\}$ and so on. This approach requires in the best- and -worst-cases $\log n$ iterations and, without laziness, $n \log n$ calls to POPPER. Suppose that after the first iteration we know that h_1 entails $\{e_1\}$, $\{e_6\}$, and $\{e_9\}$; h_2 entails $\{e_3\}$ and $\{e_4\}$; and h_3 entails $\{e_2\}$, $\{e_5\}$, and $\{e_{13}\}$. Then in the second iteration Algorithm 2 will search for solutions $h_1', h_2', \ldots, h_{n/2'}$ etc. The two searches for h_1' to cover $\{e_1, e_2\}$ and h_3' to cover $\{e_5, e_6\}$ are basically the same. To reduce the number of iterations, we can *compress* the chunks by the hypotheses that entail them. In other words, we can merge two chunks if they are covered by the same hypothesis. In the above case, we create a h_1 bucket with $\{e_1, e_6, e_9\}$, a h_2 bucket with $\{e_3, e_4\}$, and a h_3 bucket with $\{e_2, e_5, e_{13}\}$. With this compression approach in the second iteration, we now search for h_1' for the chunk $\{e_1, e_6, e_9, e_3, e_4\}$ (buckets 1 and 2) and h_2' for the chunk $\{e_2, e_5, e_{13}\}$. Compression reduces the best-case number of iterations from $\log n$ to 1 and the best-case number of calls (without laziness) to POPPER from $n \log n$ to n. In Section 5, we experimentally evaluate the impact of compression on learning performance.

Anytime. DCC is an anytime algorithm. If at any point a user stops the search or the search duration exceeds a timeout, DCC returns the best hypothesis thus far.

5 Experiments

We claim that DCC can reduce search complexity and thus improve learning performance. To evaluate this claim, our experiments aim to answer the question:

Q1 Can DCC improve predictive accuracies and reduce learning times?

To answer **Q1**, we compare DCC against POPPER. This comparison allows us to answer **Q1** as DCC uses POPPER as its underlying search algorithm and both systems use identical biases. Comparing against other systems will not allow us to answer **Q1**.

DCC has various optimisations that we claim improve learning performance, notably reusing learned constraints; *laziness*, and *compression*. To evaluate these features, our experiments aim to answer three questions:

Q2 Can reusing constraints reduce learning times?

Q3 Can laziness reduce learning times?

Q4 Can compression reduce learning times?

Task	Num. rules	Num. literals	Max rule size
trains1	1	6	6
trains2	2	11	7
trains3	3	17	7
trains4	4	26	7

Table 1: Trains tasks. The values are based on the optimal solution size. The BK contains 27k facts and 20 relations.

To answer **Q2-Q4**, we compare the performance of DCC with and without these optimisations.

Comparing DCC against other systems besides POPPER cannot help us answer questions **Q1-Q4**. However, many researchers desire comparisons against 'state-of-the-art'. To appease such a researcher, our experiments try to answer the question:

Q5 How does DCC compare against other approaches?

To answer **Q5** we compare DCC against POPPER, METAGOL, ALEPH, and ILASP3. We describe these systems in Section 5.2. Note that we do not claim that DCC is *better* than other systems. All systems have strengths and weaknesses. Moreover, as most systems use different biases, a fair comparison is difficult. Indeed, there will always exist a set of settings whereby system x outperforms system y. Therefore, the reader should not use our experimental results to conclude that system x is better than system y.

5.1 Experimental Domains

We consider three domains.

Michalski Trains. Michalski trains (Larson and Michalski 1977) is a classical problem. The task is to find a hypothesis that distinguishes eastbound trains from westbound trains. Figure 1 shows an example hypothesis that says *a train is eastbound if it has a long carriage with two wheels and another long carriage with three wheels*. We use this domain because we can easily generate progressively more difficult tasks to test the scalability of the approaches as the solution size grows. Table 1 shows information about the four tasks we consider. There are 1000 examples but the distribution of positive and negative examples is different for each task. We randomly sample the examples and split them into 80/20 train/test partitions

eastbound(A) ← has_car(A,B), long(B), two_wheels(B),
has_car(A,C), three_wheels(C)

Figure 1: Target solution for the *trains1* task.

IGGP. In *inductive general game playing* (IGGP) (Cropper, Evans, and Law 2020) agents are given game traces from the general game playing competition (Genesereth and Björnsson 2013). The task is to induce a set of rules that could have produced these traces. We use four IGGP games: *minimal decay* (md), *rock, paper, scissors* (rps), *buttons*, and *coins*. We learn the *next* relation in each game, which is the most difficult to learn (Cropper, Evans, and Law 2020).

Program synthesis. Inducing complex recursive programs has long been considered a difficult problem (Muggleton et al. 2012) and most ILP systems cannot learn recursive programs. We use the program synthesis dataset introduced by Cropper and Morel (2021a).

5.2 Systems

To answer **Q5**, we compare DCC against POPPER, METAGOL, ALEPH, and ILASP3.

METAGOL METAGOL is one of the few systems that can learn recursive Prolog programs. METAGOL uses user-provided *metarules* (program templates) to guide the search for a solution. We use the approximate universal set of metarules described by Cropper and Tourret (2020).

ALEPH ALEPH excels at learning many large non-recursive rules and should excel at the trains and IGGP tasks. Although ALEPH can learn recursive programs, it struggles to do so. DCC and ALEPH use similar biases so the comparison can be considered reasonably fair.

ILASP3 We tried to use ILASP3. However, ILASP3 first pre-computes every possible rule in a hypothesis space. This approach is infeasible for our datasets. For instance, on the trains tasks, ILASP3 took 2 seconds to pre-compute rules with three body literals; 20 seconds for rules with four body literals; and 12 minutes for rules with five body literals. Since the simplest train task requires rules with six body literals, ILASP3 is unusable. In addition, ILASP3 cannot learn Prolog programs so is unusable in the synthesis tasks.

5.3 Experimental Results

We measure predictive accuracy and learning time. We enforce a timeout of five minutes per task. We repeat all the experiments[1] 20 times and measure the mean and standard deviation. We use a 3.8 GHz 8-Core Intel Core i7 with 32GB of ram. All the systems use a single CPU.

Q1. Can DCC improve predictive accuracies and reduce learning times? Table 2 shows the predictive accuracies of DCC and POPPER. The results show that DCC outperforms POPPER, especially on the trains and IGGP tasks. A McNemar's test confirms the significance of the difference between DCC and POPPER at the $p < 0.01$ level. Where POPPER has low predictive accuracy, it is because it struggled to find a solution in the given time limit and thus returns an empty hypothesis[2].

Table 3 shows that learning times of DCC and POPPER. A paired t-test confirms that the learning times of DCC and

---

[1]The experimental code and data are available at https://github.com/logic-and-learning-lab/aaai22-dcc.

[2]DCC and POPPER (and METAGOL) are all guaranteed to find an optimal solution, if one exists. They key difference is how long they take to find it. In this experiment, we use a 5 minute timeout. However, increasing the timeout does not change the results. For instance, we repeated one trial of the *buttons* experiment with **4 hour** timeout. Even with this large timeout, POPPER and METAGOL could still not learn *any* solution. By contrast, DCC learns an almost perfect solution in under 5 minutes.

Task	DCC	POPPER	ALEPH	METAGOL
trains1	100 ± 0	100 ± 0	100 ± 0	27 ± 0
trains2	98 ± 0	98 ± 0	100 ± 0	19 ± 0
trains3	98 ± 0	81 ± 1	100 ± 0	79 ± 0
trains4	100 ± 0	42 ± 5	39 ± 4	32 ± 0
md	99 ± 0	100 ± 0	94 ± 0	11 ± 0
buttons	98 ± 0	19 ± 0	87 ± 0	19 ± 0
rps	97 ± 0	18 ± 0	100 ± 0	18 ± 0
coins	86 ± 0	17 ± 0	17 ± 0	17 ± 0
dropk	99 ± 0	100 ± 0	52 ± 2	50 ± 0
droplast	100 ± 0	100 ± 0	50 ± 0	50 ± 0
evens	100 ± 0	100 ± 0	51 ± 0	50 ± 0
finddup	98 ± 0	98 ± 0	50 ± 0	50 ± 0
last	100 ± 0	100 ± 0	49 ± 0	55 ± 3
len	100 ± 0	100 ± 0	50 ± 0	50 ± 0
sorted	94 ± 2	96 ± 1	70 ± 1	50 ± 0
sumlist	100 ± 0	100 ± 0	50 ± 0	62 ± 4

Table 2: Predictive accuracies. We round accuracies to integer values. The error is standard deviation.

Task	DCC	POPPER	ALEPH	METAGOL
trains1	8 ± 2	2 ± 0	4 ± 0.2	300 ± 0
trains2	41 ± 12	7 ± 0.9	1 ± 0.1	300 ± 0
trains3	106 ± 17	295 ± 3	35 ± 0.9	300 ± 0
trains4	268 ± 9	295 ± 2	297 ± 1	300 ± 0
md	172 ± 27	52 ± 1	3 ± 0	300 ± 0
buttons	300 ± 0	299 ± 0	86 ± 1	300 ± 0
rps	282 ± 12	285 ± 14	4 ± 0.1	0.3 ± 0
coins	291 ± 4	299 ± 0	300 ± 0	0.4 ± 0
dropk	3 ± 0.2	2 ± 0.2	3 ± 0.3	0.3 ± 0
droplast	2 ± 0.2	3 ± 0.1	300 ± 0	300 ± 0
evens	5 ± 0.4	4 ± 0.1	1 ± 0	217 ± 26
finddup	47 ± 6	13 ± 0.3	1 ± 0.1	300 ± 0
last	2 ± 0.4	2 ± 0.1	1 ± 0	270 ± 20
len	16 ± 2	5 ± 0.1	1 ± 0	300 ± 0
sorted	29 ± 3	19 ± 1	1 ± 0	288 ± 11
sumlist	18 ± 0.3	19 ± 0.6	0.6 ± 0	225 ± 29

Table 3: Learning times. We round times over one second to the nearest second. The error is standard deviation.

POPPER are *not* significantly different. This result may surprise a reader. How can DCC achieve higher predictive accuracy than POPPER yet have the same learning time given that it calls POPPER? The reason is that DCC is an anytime algorithm. On some tasks, DCC finds the optimal solution early in its search but is unable to prove that it is optimal so continues to search until it reaches the maximum learning time.

Q2. Can reusing constraints reduce learning times? Tables 4 and 5 show that reusing constraints is important for high predictive accuracy and low learning times. A McNemar's test and a paired t-test confirmed the significance of the accuracy and time results respectively at the $p < 0.01$ level. For instance, without constraint reuse, DCC takes 107s to learn a *last* solution with 82% predictive accuracy. By con-

Task	DCC	Without constraints	Without laziness	Without compression
trains1	100 ± 0	95 ± 3	100 ± 0	100 ± 0
trains2	98 ± 0	90 ± 1	96 ± 1	98 ± 0
trains3	98 ± 0	70 ± 4	97 ± 0	98 ± 0
trains4	100 ± 0	77 ± 2	97 ± 0	100 ± 0
md	99 ± 0	92 ± 0	95 ± 1	98 ± 0
buttons	98 ± 0	84 ± 0	96 ± 0	98 ± 0
rps	97 ± 0	86 ± 0	95 ± 0	97 ± 0
coins	86 ± 0	77 ± 4	71 ± 6	81 ± 3
dropk	99 ± 0	90 ± 4	99 ± 0	99 ± 0
droplast	100 ± 0	100 ± 0	100 ± 0	100 ± 0
evens	100 ± 0	83 ± 5	100 ± 0	100 ± 0
finddup	98 ± 0	56 ± 3	99 ± 0	97 ± 2
last	100 ± 0	82 ± 5	100 ± 0	100 ± 0
len	100 ± 0	85 ± 5	98 ± 1	100 ± 0
sorted	94 ± 2	60 ± 2	96 ± 1	96 ± 1
sumlist	100 ± 0	95 ± 3	100 ± 0	100 ± 0

Table 4: Predictive accuracies. We round accuracies to integer values. The error is standard deviation.

Task	DCC	Without constraints	Without laziness	Without compression
trains1	8 ± 2	38 ± 20	300 ± 0	12 ± 3
trains2	41 ± 12	285 ± 14	219 ± 20	151 ± 25
trains3	106 ± 17	300 ± 0	300 ± 0	300 ± 0
trains4	268 ± 9	300 ± 0	300 ± 0	300 ± 0
md	172 ± 27	300 ± 0	251 ± 20	256 ± 22
buttons	300 ± 0	300 ± 0	300 ± 0	300 ± 0
rps	282 ± 12	300 ± 0	300 ± 0	300 ± 0
coins	291 ± 4	300 ± 0	300 ± 0	300 ± 0
dropk	3 ± 0.2	62 ± 27	20 ± 2	4 ± 0.5
droplast	2 ± 0.2	4 ± 0.2	101 ± 2	4 ± 0.2
evens	5 ± 0.4	109 ± 32	75 ± 3	11 ± 3
finddup	47 ± 6	272 ± 19	178 ± 15	102 ± 19
last	2 ± 0.4	107 ± 32	46 ± 3	4 ± 1
len	16 ± 2	95 ± 30	143 ± 14	36 ± 14
sorted	29 ± 3	286 ± 14	74 ± 7	109 ± 21
sumlist	18 ± 0.3	50 ± 19	300 ± 0	22 ± 0.5

Table 5: Learning times. We round times over one second to the nearest second. The error is standard deviation.

trast, with constraint reuse DCC takes 2s to learn a solution with 100% accuracy.

Q3. Can laziness reduce learning times? The results show that laziness can drastically reduce learning times. A paired t-test confirms the significance at the $p < 0.01$ level. For instance, for the *trains1* task, laziness reduces the learning time from 300s (timeout) to 8s.

Q4. Can compression reduce learning times? The results show that compression can drastically reduce learning times. A paired t-test confirms the significance at the $p < 0.01$ level. For instance, for the *sorted* task, compression reduces the learning time from 109s to 29s.

Q5. How does DCC compare against other ILP systems? The results show that DCC generally outperforms the other systems in terms of predictive accuracy. ALEPH performs well on the trains and IGGP tasks but struggles on the program synthesis tasks. POPPER performs well on the trains and program synthesis tasks but struggles on the IGGP tasks. METAGOL struggles on most tasks because it uses metarules with at most two body literals. To learn rules with more than two body literals, METAGOL must invent new predicates, thus increasing the hypothesis size and hypothesis space. The most notable difference in accuracies is in the *coins* task. DCC achieves 86% accuracy. By contrast, the other systems could only achieve the default accuracy of 17%

6 Conclusions and Limitations

The fundamental challenge in ILP is to efficiently search a large hypothesis space. To address this challenge, we have introduced an approach called *divide*, *constrain*, and *conquer* (DCC). Our approach combines classical divide-and-conquer search with modern constraint-driven search. Our anytime approach can learn optimal, recursive, and large programs and perform predicate invention. Our experiments on three domains (classification, inductive general game playing, and program synthesis) show that (i) our approach can drastically improve predictive accuracies and reduce learning times, (ii) reusing learned knowledge is vital for good learning performance, and (iii) our approach can outperform other ILP systems.

6.1 Limitations and Future Work

Noise. In contrast to many other systems, such as TILDE and ALEPH, DCC cannot (explicitly) handle misclassified examples. However, due to the D&C approach, DCC naturally provides a method to handle misclassified positive examples. In future work we want to extend the approach to handle misclassified negative examples.

Expressivity. As DCC uses POPPER as its underlying search algorithm, it inherits some of the limitations of POPPER, such as no explicit support for non-observational predicate learning (Muggleton and Bryant 2000), where a hypothesis must define rules for predicates not seen in the training examples[3], and a restriction to definite programs. Future work should address these limitations.

Parallelisation. Although multi-core machines are ubiquitous, most ILP approaches are single-core learners, including all the systems mentioned in this paper. However, our DCC algorithm is trivially parallelisable as we can independently search for a solution for each chunk. In future work, we want to explore parallel DCC approaches.

Acknowledgments

---

[3]Because POPPER supports predicate invention, the distinction between OPL and non-OPL is unclear as by definition an invented predicate symbol is not given in the examples.

References

Ahlgren, J.; and Yuen, S. Y. 2013. Efficient program synthesis using constraint satisfaction in inductive logic programming. *J. Machine Learning Res.*, 14(1): 3649–3682.

Blockeel, H.; and Raedt, L. D. 1998. Top-Down Induction of First-Order Logical Decision Trees. *Artif. Intell.*

Corapi, D.; Russo, A.; and Lupu, E. 2011. Inductive Logic Programming in Answer Set Programming. In Muggleton, S.; Tamaddoni-Nezhad, A.; and Lisi, F. A., eds., *Inductive Logic Programming - 21st International Conference, ILP 2011, Windsor Great Park, UK, July 31 - August 3, 2011, Revised Selected Papers*, volume 7207 of *Lecture Notes in Computer Science*, 91–97. Springer.

Cropper, A.; Dumancic, S.; Evans, R.; and Muggleton, S. H. 2022. Inductive logic programming at 30. *Mach. Learn.*, 111(1): 147–172.

Cropper, A.; Evans, R.; and Law, M. 2020. Inductive general game playing. *Mach. Learn.*, 109(7): 1393–1434.

Cropper, A.; and Morel, R. 2021a. Learning programs by learning from failures. *Mach. Learn.*, 110(4): 801–856.

Cropper, A.; and Morel, R. 2021b. Predicate Invention by Learning From Failures. *CoRR*.

Cropper, A.; and Muggleton, S. H. 2016. Metagol System. https://github.com/metagol/metagol. Accessed: 2022-03-07.

Cropper, A.; and Tourret, S. 2020. Logical reduction of metarules. *Mach. Learn.*, 109(7): 1323–1369.

Evans, R.; and Grefenstette, E. 2018. Learning Explanatory Rules from Noisy Data. *J. Artif. Intell. Res.*, 61: 1–64.

Evans, R.; Hernández-Orallo, J.; Welbl, J.; Kohli, P.; and Sergot, M. J. 2021. Making sense of sensory input. *Artif. Intell.*, 293: 103438.

Gebser, M.; Kaminski, R.; Kaufmann, B.; and Schaub, T. 2014. Clingo = ASP + Control: Preliminary Report. *CoRR*, abs/1405.3694.

Genesereth, M. R.; and Björnsson, Y. 2013. The International General Game Playing Competition. *AI Magazine*, 34(2): 107–111.

Kaminski, T.; Eiter, T.; and Inoue, K. 2018. Exploiting Answer Set Programming with External Sources for Meta-Interpretive Learning. *Theory Pract. Log. Program.*, 18(3-4): 571–588.

Larson, J.; and Michalski, R. S. 1977. Inductive inference of VL decision rules. *SIGART Newsletter*, 63: 38–44.

Law, M. 2018. *Inductive learning of answer set programs*. Ph.D. thesis, Imperial College London, UK.

Lloyd, J. W. 2012. *Foundations of logic programming*. Springer Science & Business Media.

Muggleton, S. 1991. Inductive Logic Programming. *New Generation Computing*, 8(4): 295–318.

Muggleton, S. 1995. Inverse Entailment and Progol. *New Generation Comput.*, 13(3&4): 245–286.

Muggleton, S.; and Bryant, C. H. 2000. Theory Completion Using Inverse Entailment. In Cussens, J.; and Frisch, A. M., eds., *Inductive Logic Programming, 10th International Conference, ILP 2000, London, UK, July 24-27, 2000, Proceedings*, volume 1866 of *Lecture Notes in Computer Science*, 130–146. Springer.

Muggleton, S.; De Raedt, L.; Poole, D.; Bratko, I.; Flach, P. A.; Inoue, K.; and Srinivasan, A. 2012. ILP turns 20 - Biography and future challenges. *Machine Learning*, 86(1): 3–23.

Muggleton, S. H.; Lin, D.; and Tamaddoni-Nezhad, A. 2015. Meta-interpretive learning of higher-order dyadic Datalog: predicate invention revisited. *Machine Learning*, 100(1): 49–73.

Plotkin, G. 1971. *Automatic Methods of Inductive Inference*. Ph.D. thesis, Edinburgh University.

Quinlan, J. R. 1986. Induction of Decision Trees. *Machine Learning*, 1(1): 81–106.

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning*.

Ray, O. 2009. Nonmonotonic abductive inductive learning. *J. Applied Logic*, 7(3): 329–340.

Rissanen, J. 1978. Modeling by shortest data description. *Autom.*, 14(5): 465–471.

Srinivasan, A. 2001. The ALEPH manual. *Machine Learning at the Computing Laboratory, Oxford University*.

Stahl, I. 1995. The Appropriateness of Predicate Invention as Bias Shift Operation in ILP. *Machine Learning*, 20(1-2): 95–117.