Combining Divide-and-Conquer and Separate-and-Conquer for Efficient and Effective Rule Induction

Henrik Boström and Lars Asker

Dept. of Computer and Systems Sciences
Stockholm University and Royal Institute of Technology
Electrum 230, 164 40 Kista, Sweden
{henke,asker}@dsv.su.se

Abstract. Divide-and-Conquer (DAC) and Separate-and-Conquer (SAC) are two strategies for rule induction that have been used extensively. When searching for rules DAC is maximally conservative w.r.t. decisions made during search for previous rules. This results in a very efficient strategy, which however suffers from difficulties in effectively inducing disjunctive concepts due to *the replication problem*. SAC on the other hand is maximally liberal in the same respect. This allows for a larger hypothesis space to be searched, which in many cases avoids the replication problem but at the cost of lower efficiency. We present a hybrid strategy called Reconsider-and-Conquer (RAC), which handles the replication problem more effectively than DAC by reconsidering some of the earlier decisions and allows for more efficient induction than SAC by holding on to some of the decisions. We present experimental results from propositional, numerical and relational domains demonstrating that RAC significantly reduces the replication problem from which DAC suffers and is several times (up to an order of magnitude) faster than SAC.

1 Introduction

The two main strategies for rule induction are Separate-and-Conquer and Divide-and-Conquer. Separate-and-Conquer, often also referred to as Covering, produces a set of rules by repeatedly specialising an overly general rule. At each iteration a specialised rule is selected that covers a subset of the positive examples and excludes the negative examples. This is repeated until all positive examples are covered by the set of rules. The reader is referred to [7] for an excellent overview of Separate-and-Conquer rule learning algorithms. Divide-and-Conquer produces a hypothesis by splitting an overly general rule into a set of specialised rules that cover disjoint subsets of the examples. The rules that cover positive examples only are kept, while the rules that cover both positive and negative examples are handled recursively in the same manner as the first overly general rule.

For Separate-and-Conquer, the computational cost (measured as the number of checks to see whether or not a rule covers an example) grows quadratically in the size of the example set, while it grows linearly using Divide-and-Conquer.[1] This follows from the fact that Separate-and-Conquer searches a larger hypothesis space than Divide-and-Conquer [2]. For any hypothesis in this larger space there is a corresponding hypothesis with identical coverage in the narrower space. Hence none of the strategies is superior to the other in terms of expressiveness. However, for many of the hypotheses within the narrower space there is a hypothesis with identical coverage but fewer rules within the larger space. Since the number of rules in a hypothesis provides a bound on the minimal number of examples needed to find it, this means that Separate-and-Conquer often requires fewer examples than Divide-and-Conquer to find a correct hypothesis. In particular this is true in domains in which the *replication problem* [13] is frequent, i.e. when the most compact definition of the target concept consists of disjuncts whose truth values are (partially or totally) independent, e.g. $p(x_1, x_2, x_3, x_4) \leftarrow (x_1 = 1 \wedge x_2 = 2) \vee (x_3 = 0 \wedge x_4 = 1)$.

The two strategies can be viewed as extremes w.r.t. how conservative they are regarding earlier decisions when searching for additional rules. Divide-and-Conquer can be regarded as maximally conservative and Separate-and-Conquer as maximally liberal. We propose a hybrid strategy, called Reconsider-and-Conquer, which combines the advantages of Divide-and-Conquer and Separate-and-Conquer and reduces their weaknesses. The hybrid strategy allows for more effective handling of the replication problem than Divide-and-Conquer by reconsidering some decisions made in the search for previous rules. At the same time it allows for more efficient induction than Separate-and-Conquer by holding on to some of the earlier decisions.

In the next section we introduce some basic terminology. In section three, we formally describe Reconsider-and-Conquer, and in section four we present experimental results from propositional, numerical and relational domains demonstrating that the hybrid approach may significantly reduce the replication problem from which Divide-and-Conquer suffers while at the same time it is several times faster than Separate-and-Conquer. In section five, we discuss related work and in section six finally, we give some concluding remarks and point out directions for future research.

2 Preliminaries

The reader is assumed to be familiar with the logic programming terminology [11].

A *rule* r is a definite clause $r = h \leftarrow b_1 \wedge \ldots \wedge b_n$, where h is the head, and $b_1 \wedge \ldots \wedge b_n$ is a (possibly empty) body.

A *rule* g is *more general* than a rule s w.r.t. a set of rules B (background predicates), denoted $g \succeq_B s$, iff $M_{\{g\} \cup B} \supseteq M_{\{s\} \cup B}$, where M_P denotes the least Herbrand model of P (the set of all ground facts that follow from P).

The *coverage* of a set of rules H, w.r.t. background predicates B and a set of atoms A, is a set $A_{HB} = \{a : a \in A \cap M_{H \cup B}\}$. We leave out the subscript

[1] assuming that the maximum number of ways to specialise a rule is fixed.

B when it is clear from the context. Furthermore, if H is a singleton $H = \{r\}$, then $A_{\{r\}}$ is abbreviated to A_r.

Given a rule r and background predicates B, a *specialisation operator* σ computes a set of rules, denoted $\sigma_B(r)$, such that for all $r' \in \sigma_B(r)$, $r \succeq_B r'$. Again we leave out the subscript if B is clear from the context.

Given a rule r, background predicates B, and a specialisation operator σ, a *split* of r is a set of rules $s = \{r_1, \ldots, r_n\}$, such that $r_i \in \sigma(r)$, for all $1 \leq i \leq n$, and $M_{\{r_1,\ldots,r_n\} \cup B} = M_{\{r\} \cup B}$. Furthermore, s is said to be a *non-overlapping* split if $M_{\{r_i\} \cup B} \cap M_{\{r_j\} \cup B} = M_B$ for all $i, j = 1, \ldots, n$ such that $i \neq j$.

3 Reconsider-and-Conquer

Reconsider-and-Conquer works like Separate-and-Conquer in that rules are iteratively added to the hypothesis while removing covered examples from the set of positive examples. However, in contrast to Separate-and-Conquer, which adds a single rule on each iteration, Reconsider-and-Conquer adds a set of rules. The first rule that is included in this set is generated in exactly the same way as is done by Separate-and-Conquer, i.e. by following a branch of specialisation steps from an initial rule into a rule that covers positive examples only. However, instead of continuing the search for a subsequent rule from the initial rule, Reconsider-and-Conquer backs up one step to see whether some other specialisation step could be taken in order to cover some of the remaining positive examples (i.e. to complete another branch). This continues until eventually Reconsider-and-Conquer has backed up to the inital rule. The way in which this set of rules that is added to the hypothesis is generated is similar to how Divide-and-Conquer works, but with one important difference: Reconsider-and-Conquer is less restricted than Divide-and-Conquer regarding what possible specialisation steps can be taken when having backed up since the specialisation steps are not chosen independently by Divide-and-Conquer due to that the resulting rules should constitute a (possibly non-overlapping) split.

One condition used by Reconsider-and-Conquer to decide whether or not to continue from some rule on a completed branch is that the fraction of positive examples among the covered examples must never decrease anywhere along the branch (as this would indicate that the branch is focusing on covering negative rather than positive examples). However, in principle both weaker and stronger conditions could be employed. The weakest possible condition would be that each rule along the branch should cover at least one positive example. This would make Reconsider-and-Conquer behave in a way very similar to Divide-and-Conquer. A maximally strong condition would be to always require Reconsider-and-Conquer back up to the initial rule, making the behaviour identical to that of Separate-and-Conquer.

In Figure 1, we give a formal description of the algorithm. The branch of specialisation steps currently explored is represented by a stack of rules together with the positive and negative examples that they cover. Once a branch is completed, i.e a rule that covers only positive examples is added to the hypothesis, the stack is updated by removing all covered examples from the stack. Furthermore, the stack is truncated by keeping only the bottom part where the fraction of positive examples covered by each rule does not decrease compared to those covered by the preceding rule.

---

function Reconsider-and-Conquer(E^+, E^-)
  $H := \emptyset$
  while $E^+ \neq \emptyset$ **do**
    $r :=$ an initial rule such that $E_r^+ \neq \emptyset$
    $H' :=$ Find-Rules($r, [], E_r^+, E_r^-$)
    $E^+ := E^+ \setminus E_{H'}^+$
    $H := H \cup H'$
  return H

function Find-Rules(r, S, E^+, E^-)
  if $E^- \neq \emptyset$ **then**
    $r' :=$ a rule $\in \sigma(r)$
    $H :=$ Find-Rules($r', (r, E^+, E^-) \cdot S, E_{r'}^+, E_{r'}^-$)
  else $H := \{r\}$
  repeat
  Update S w.r.t. H
  if $S \neq []$ **then**
    Pop $(r, E^{\oplus}, E^{\ominus})$ from S
    if there is a rule $r' \in \sigma(r)$ such that $\frac{|E_{r'}^{\oplus}|}{|E_{r'}^{\oplus} \cup E_{r'}^{\ominus}|} \geq \frac{|E^{\oplus}|}{|E^{\oplus} \cup E^{\ominus}|}$ **then**
     $H' :=$ Find-Rules($r', (r, E^{\oplus}, E^{\ominus}) \cdot S, E_{r'}^{\oplus}, E_{r'}^{\ominus}$)
     $H := H \cup H'$
     $S := (r, E^{\oplus}, E^{\ominus}) \cdot S$
  until $S = []$
  return H

Fig. 1. The Reconsider-and-Conquer algorithm.

An Example Assume that the target predicate is $p(x_1, x_2, x_3, x_4) \leftarrow (x_1 = 1 \wedge x_2 = 1) \vee (x_3 = 1 \wedge x_4 = 1) \vee (x_3 = 1 \wedge x_4 = 2)$, and that we are given 100 positive and 100 negative instances of the target predicate, i.e. $|E^+| = 100$ and $|E^-| = 100$. Assume further that our specialisation operator is defined as $\sigma(h \leftarrow b_1 \wedge \ldots \wedge b_n) = \{h \leftarrow b_1 \wedge \ldots \wedge b_n \wedge x = c | x$ is a variable in h and $c \in \{1, \ldots, 4\}\}$. Now assuming that Reconsider-and-Conquer starts with the initial rule $r_0 = p(x_1, x_2, x_3, x_4)$, Find-Rules recursively generates a sequence of more specialised rules, say:

$$r_1 = p(x_1,x_2,x_3,x_4) \leftarrow x_3 = 1 \qquad |E^+_{r_1}| = 50 \ |E^-_{r_1}| = 10$$
$$r_2 = p(x_1,x_2,x_3,x_4) \leftarrow x_3 = 1 \wedge x_4 = 1 \ |E^+_{r_2}| = 25 \ |E^-_{r_2}| = 0$$

where the last rule is included in the hypothesis.

Find-Rules then updates the stack by removing all covered (positive) examples and keeping only the bottom part of the stack that corresponds to a sequence of specialisation steps that fulfills the condition of a non-decreasing fraction of covered positive examples. In this case, the bottom element r_1 is kept as it covers 25 positive and 10 negative examples compared to 75 positive and 100 negative examples that are covered by the initial rule. So in contrast to Separate-and-Conquer, Reconsider-and-Conquer does not restart the search from the initial rule but continues from rule r_1 and finds a specialisation that does not decrease the fraction of positive examples, say:

$$r_3 = p(x_1,x_2,x_3,x_4) \leftarrow x_3 = 1 \wedge x_4 = 2 \ |E^+_{r_3}| = 25 \ |E^-_{r_3}| = 0$$

After the stack is updated, no rules remain and hence Reconsider-and-Conquer restarts the search from an initial rule, and may choose any specialisation without being restricted by the earlier choices. This contrasts to Divide-and-Conquer, which would have had to choose some of the other rules in the (non-overlapping) split from which r_1 was taken (e.g. $\{p(x_1,x_2,x_3,x_4) \leftarrow x_3 = 2, p(x_1,x_2,x_3,x_4) \leftarrow x_3 = 3, p(x_1,x_2,x_3,x_4) \leftarrow x_3 = 4\}$). Assuming the same initial rule (r_0) is chosen again, the sequence of rules produced by Find-Rules may look like:

$$r_4 = p(x_1,x_2,x_3,x_4) \leftarrow x_1 = 1 \qquad |E^+_{r_4}| = 50 \ |E^-_{r_4}| = 20$$
$$r_5 = p(x_1,x_2,x_3,x_4) \leftarrow x_1 = 1 \wedge x_2 = 1 \ |E^+_{r_5}| = 50 \ |E^-_{r_5}| = 0$$

The last rule is included in the hypothesis and now all positive examples are covered so Reconsider-and-Conquer terminates. Hence, the resulting hypothesis is:

$$r_2 = p(x_1,x_2,x_3,x_4) \leftarrow x_3 = 1 \wedge x_4 = 1$$
$$r_3 = p(x_1,x_2,x_3,x_4) \leftarrow x_3 = 1 \wedge x_4 = 2$$
$$r_5 = p(x_1,x_2,x_3,x_4) \leftarrow x_1 = 1 \wedge x_2 = 1$$

It should be noted that Divide-and-Conquer has to induce seven rules to obtain a hypothesis with identical coverage, as the disjunct that corresponds to r_5 above has to be replicated in five rules:

$$p(x_1,x_2,x_3,x_4) \leftarrow x_3 = 1 \wedge x_4 = 1$$
$$p(x_1,x_2,x_3,x_4) \leftarrow x_3 = 1 \wedge x_4 = 2$$
$$p(x_1,x_2,x_3,x_4) \leftarrow x_3 = 1 \wedge x_4 = 3 \wedge x_1 = 1 \wedge x_2 = 1$$
$$p(x_1,x_2,x_3,x_4) \leftarrow x_3 = 1 \wedge x_4 = 4 \wedge x_1 = 1 \wedge x_2 = 1$$
$$p(x_1,x_2,x_3,x_4) \leftarrow x_3 = 2 \wedge x_1 = 1 \wedge x_2 = 1$$
$$p(x_1,x_2,x_3,x_4) \leftarrow x_3 = 3 \wedge x_1 = 1 \wedge x_2 = 1$$
$$p(x_1,x_2,x_3,x_4) \leftarrow x_3 = 4 \wedge x_1 = 1 \wedge x_2 = 1$$

4 Empirical Evaluation

We first describe how the experiments were performed and then present the experimental results.

4.1 Experimental setting

Reconsider-and-Conquer (RAC) was compared to Divide-and-Conquer (DAC) and Separate-and-Conquer (SAC) in several propositional, numerical and relational domains. The domains were obtained from the UCI repository [1], except for two relational domains: one consists of four sets of examples regarding structure-activity comparisons of drugs for the treatment of Alzheimer's desease, and was obtained from Oxford University Computing Laboratory and the other domain is about learning the definite clause grammar (DCG) in [3, p 455].[2]

All three algorithms were tested both with information gain heuristics and probability metrics based on the hypergeometric distribution, which for DAC are those given in [14] and [12] respectively, while the information gain heuristic modified for SAC (and RAC) is taken from [10, p 165], and the modified version of the probability metric in [12] for SAC and RAC is:

$$P(|E^+_{r'}|, |E^-_{r'}|, |E^+_r|, |E^-_r|) = \frac{\binom{|E^+_r|}{|E^+_{r'}|}\binom{|E^-_r|}{|E^-_{r'}|}}{\binom{|E^+_r \cup E^-_r|}{|E^+_{r'} \cup E^-_{r'}|}}$$

where $r' \in \sigma(r)$. The specialisation r' of r with lowest probability is chosen from $\sigma(r)$ given that

$$\frac{|E^+_{r'}|}{|E^+_{r'} \cup E^-_{r'}|} \geq \frac{|E^+_r|}{|E^+_r \cup E^-_r|}$$

Following [5], cut-points for continuous-valued attributes were chosen dynamically from the boundary points between the positive and negative examples in the training sets for the numerical and relational domains.

An experiment was performed in each domain, in which the entire example set was randomly split into two partitions corresponding to 90% and 10% of the examples respectively. The larger set was used for training and the smaller for testing. The same training and test sets were used for all algorithms. Each experiment was iterated 30 times and the mean accuracy on the test examples as

[2] The set of positive examples consists of all sentences of up to 10 words that can be generated by the grammar (2869 sentences) and the equal sized set of negative examples was generated by applying the following procedure to each positive example: i) replace a randomly selected word in the sentence with a randomly selected word from the corpus, ii) go to step i with a probability of 0.5 and iii) restart the procedure if the resulting sentence is a positive example.

well as the amount of work performed measured as the cpu time[3] are presented below. In addition, we also present the mean number of rules in the produced hypotheses.

4.2 Experimental results

In Table 1, we present the accuracy, cpu time and number of rules in the hypothesis produced by each algorithm using both the probability metrics and the information gain heuristics for all domains. The domains within the first group in the table are propositional, the domains within the second group are numerical, and the domains in the last group are relational. For all accuracies, bold face indicates that there is a statistically significant difference between the method and some less accurate method and no significant difference between the method and some more accurate method (if any). Furthermore, underline indicates that there is a statistically significant difference between the method and some more accurate method and no significant difference between the method and some less accurate method (if any).

The accuracies of the three methods are shown in columns 2-4. To summarise these results, one can see that DAC has a best/worst score (as indicated by bold and underline in the table) of 3/10 (5/8) (the first score is for the probability metrics and the second is for the information gain heuristics). The corresponding score for SAC is 9/3 (8/3) and for RAC 8/3 (8/2). Looking more closely at the domains, one can see that there is a significant difference in accuracy between DAC and SAC in favour of the latter in those (artificial) domains in which the replication problem was expected to occur (Tic-Tac-Toe, KRKI) but also in several of the other (natural) domains (most notably Student loan and Alzh. chol.). One can see that RAC effectively avoids the replication problem in these domains and is almost as accurate as, or even more accurate than, SAC.

DCG is the only relational domain in which DAC is significantly more accurate than SAC and RAC (although the difference is small). In this domain the replication problem is known not to occur since the shortest correct grammar is within the hypothesis space of DAC. The difference in accuracy can here be explained by the different versions of the probability metrics and information gain heuristics that are used for DAC and SAC/RAC. For DAC these reward splits that discriminate positive from negative examples while for SAC/RAC they reward rules with a high coverage of positive examples.

In columns 5-7, the cputime of the three methods are shown. The median for the cpu time ratio SAC/DAC is 5.68 (5.77) and for the cpu time ratio RAC/DAC

[3] The amount of work was also measured by counting the number of times it was checked whether or not a rule covers an example, which has the advantage over the former measure that it is independent of the implementation but the disadvantage that it does not include the (small) overhead of RAC due to handling the stack. However, both measures gave consistent results and we have chosen to present only the former. All algorithms were implemented in SICStus Prolog 3 #6 and were executed on a SUN Ultra 60, except for the numerical domains which were executed on a SUN Ultra I.

Domain	*Accuracy (percent)*			*Time (seconds)*			*No. of rules*		
	DAC	*SAC*	*RAC*	*DAC*	*SAC*	*RAC*	*DAC*	*SAC*	*RAC*
Shuttle	97.98	**99.17**	**99.17**	0.13	0.34	0.17	9.9	5.8	5.8
	98.33	99.17	99.17	0.13	0.33	0.16	8.1	5.8	5.8
Housevotes	93.26	94.11	93.95	1.78	4.98	2.29	17.1	13.8	15.1
	93.64	93.95	93.64	1.73	5.00	2.20	16.7	13.9	15.4
Tic-Tac-Toe	85.59	**99.58**	99.03	2.04	7.10	4.13	108.3	13.0	21.2
	85.63	**99.55**	99.13	1.74	6.74	3.96	107.2	12.4	20.3
KRvsKP	**99.58**	99.20	99.30	96.90	379.77	119.93	18.5	17.4	16.2
	99.59	99.18	99.23	95.77	386.40	145.31	18.2	18.1	20.4
Splice (n)	**92.97**	91.67	92.41	284.76	3508.04	904.60	163.9	82.7	118.8
	92.88	90.07	**91.88**	278.53	3636.01	897.20	162.7	84.9	123.4
Splice (ei)	95.29	**96.32**	**96.23**	208.49	2310.56	318.57	79.6	30.6	42.9
	95.43	**96.04**	**96.09**	204.64	2344.55	311.55	77.4	31.7	42.9
Splice (ie)	93.77	93.29	93.55	243.76	4109.09	375.68	111.8	47.4	67.1
	94.10	93.01	93.51	235.31	3980.19	419.59	106.7	46.9	67.3
Monks-2	66.44	70.28	**73.89**	2.03	31.73	5.31	129.2	107.3	108.7
	72.44	68.78	**73.11**	0.88	31.52	5.00	121.0	107.9	109.1
Monks-3	96.79	96.42	96.42	0.69	3.98	1.15	27.1	23.7	23.7
	97.03	96.30	96.48	0.52	3.87	1.12	26.4	23.6	23.8
Bupa	62.25	65.59	65.00	39.62	192.75	75.88	37.4	25.1	29.4
	63.92	65.69	65.88	43.00	193.31	74.13	36.3	26.9	31.9
Ionosphere	88.10	86.86	88.29	1201.02	1965.23	1273.14	9.1	5.8	7.2
	88.19	86.57	87.05	1371.10	1916.97	1035.10	8.3	7.7	9.6
Pima Indians	72.73	71.26	71.95	323.73	3079.90	587.48	57.5	38.4	45.1
	72.60	71.99	70.39	329.26	2756.38	537.69	56.4	40.2	49.9
Sonar	71.27	71.90	73.49	2049.23	3484.61	2319.63	9.7	6.5	8.5
	72.70	74.13	74.60	2123.46	3247.45	2254.65	9.3	6.8	9.4
WDBC	91.40	**94.09**	91.99	4872.89	6343.40	4885.81	9.1	6.6	8.4
	91.99	**93.63**	92.63	4870.71	6656.17	4758.54	8.9	6.7	10.2
KRKI	98.03	**98.90**	**99.17**	50.37	101.90	55.46	26.8	9.2	13.0
	98.07	**98.90**	**99.13**	50.08	101.75	55.39	26.7	9.4	12.7
DCG	**99.99**	99.93	99.92	43.83	205.76	129.40	28.0	28.8	28.5
	99.99	99.97	99.95	44.17	204.05	130.55	28.0	28.1	28.1
Student loan	90.53	**96.57**	**96.57**	10.72	60.94	21.00	113.4	35.9	44.0
	92.03	**96.83**	**96.40**	10.13	58.50	21.48	96.7	36.7	44.7
Alzh. toxic	77.53	**81.50**	**81.84**	238.27	1581.34	406.28	198.4	83.5	92.7
	77.68	**81.54**	**81.91**	195.89	1593.38	417.64	190.9	84.8	95.4
Alzh. amine	86.14	85.60	85.56	208.13	2146.01	337.96	148.8	81.6	92.0
	85.89	85.85	84.15	162.83	1854.61	330.25	144.8	82.6	94.0
Alzh. scop.	56.15	**59.84**	**60.21**	215.50	2673.01	501.71	263.1	124.5	135.4
	55.47	**60.99**	**60.36**	195.14	2259.70	528.71	261.5	121.4	132.3
Alzh. chol.	64.76	**75.06**	**74.99**	452.24	10148.40	1063.30	450.4	228.2	240.8
	64.86	**75.54**	**75.31**	364.41	9980.11	1103.79	438.8	226.0	244.9

Table 1. Accuracy, cpu time, and number of rules using probability metrics (first line) and information gain (second line).

it is 1.71 (1.78). The domain in which the SAC/RAC ratio is highest is Splice(ie) where the ratio is 10.94 (9.5). Except for Monks-2, the domain for which the SAC/DAC ratio is highest is Alz. chol., where it is 27.31 (22.44). The RAC/DAC ratio in this domain is 3.03 (2.35). In Monks-2, the SAC/DAC ratio is even higher for the probability metrics (35.82) but lower for the information gain heuristics (15.63). The RAC/DAC ratio reaches its highest value in this domain (5.68 for the probability metric).

In columns 8-10, the average number of rules produced are shown. DAC produces more rules than SAC in all domains except one (DCG). The median for the number of rules produced by SAC is 28.8 (28.1), for RAC 29.4 (31.9) and for DAC 57.5 (56.4). These results are consistent with the fact that SAC and RAC search a larger hypothesis space than DAC in which more compact hypotheses may be found.

In summary, both RAC and SAC outperform DAC in most domains tested in the experiments, mainly due to the effective handling of the replication problem. But although RAC is about as accurate as SAC, it is up to an order of magnitude faster.

5 Related Work

Two previous systems, IREP [8] and RIPPER [4], are able to efficiently process large sets of noisy data despite the use of Separate-and-Conquer. The main reason for this efficiency is the use of a technique called *incremental reduced error pruning*, which prunes each rule immediately after it has been induced, rather than after all rules have been generated. This speeds up the induction process as the pruned rules allow larger subsets of the remaining positive examples to be removed at each iteration compared to the non-pruned rules. It should be noted that this technique could also be employed directly in Reconsider-and-Conquer, improving the efficiency (and accuracy) further, especially in noisy domains.

Like Reconsider-and-Conquer, a recently proposed method for rule induction, called PART, also employs a combination of Divide-and-Conquer and Separate-and-Conquer [6]. One major difference between PART and Reconsider-and-Conquer is that the former method uses Divide-and-Conquer to find *one* rule that is added to the resulting hypothesis, while the latter method uses (a generalised version of) Divide-and-Conquer for generating *a set of* rules that is added. The purpose of the former method to use Divide-and-Conquer is not to gain efficiency over Separate-and-Conquer, but to avoid a problem called *hasty generalisation* that may occur when employing incremental reduced error pruning, like IREP and RIPPER do. Again, the former method may in fact be used as a pruning technique in conjunction with Reconsider-and-Conquer rather than Separate-and-Conquer.

In C4.5rules [16], a set of rules is first generated using Divide-and-Conquer, and then simplified by a post-pruning process. However, the cost of this process is cubic in the number of examples [4], which means that it could be even more expensive than using Separate-and-Conquer in the first place to overcome the replication problem. Still, the post-pruning techniques employed by C4.5rules (and other systems e.g. RIPPER) could be useful for both Separate-and-Conquer as well as Reconsider-and-Conquer. The main advantage of using Reconsider-and-Conquer for generating the initial set of rules compared to Divide-and-Conquer as used in C4.5rules is that significantly fewer rules need to be considered by the post-pruning process when having employed the former.

There have been other approaches to the replication problem within the framework of decision tree learning. One approach is to restrict the form of the trees when growing them, which then allows for merging of isomorphic subtrees [9]. It should be noted that these techniques are, in contrast to Reconsider-and-Conquer, yet restricted to propositional and numerical domains.

6 Concluding Remarks

A hybrid strategy of Divide-and-Conquer and Separate-and-Conquer has been presented, called Reconsider-and-Conquer. Experimental results from propositional, numerical and relational domains have been presented demonstrating that Reconsider-and-Conquer significantly reduces the replication problem from which Divide-and-Conquer suffers and that it is several times (up to an order of magnitude) faster than Separate-and-Conquer. In the trade-off between accuracy and amount of cpu time needed, we find Reconsider-and-Conquer in many cases to be a very good alternative to both Divide-and-Conquer and Separate-and-Conquer.

There are a number of directions for future research. One is to explore both pre- and post-pruning techniques in conjunction with Reconsider-and-Conquer. The techniques that have been developed for Separate-and-Conquer can in fact be employed directly as mentioned in the previous section. Another direction is to investigate alternative conditions for the decision made by Reconsider-and-Conquer regarding whether or not the search should continue from some rule on a completed branch. The currently employed condition that the fraction of covered positive examples should never decrease worked surprisingly well, but other conditions, e.g. based on some significance test, may be even more effective.

Acknowledgements This work has been supported by the European Community ESPRIT Long Term Research Project no. 20237 *Inductive Logic Programming II* and the Swedish Research Council for Engineering Sciences (TFR).

References

1. Blake C., Keogh E. and Merz C.J., *UCI Repository of machine learning databases*, Irvine, CA: University of California, Department of Information and Computer Science (1998)
2. Boström H., "Covering vs. Divide-and-Conquer for Top-Down Induction of Logic Programs", *Proc. of the Fourteenth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann (1995) 1194–1200
3. Bratko I., *Prolog Programming for Artificial Intelligence* , (2nd edition), Addison-Wesley (1990)
4. Cohen W. W., "Fast Effective Rule Induction", *Machine Learning: Proc. of the 12th International Conference*, Morgan Kaufmann (1995) 115–123
5. Fayyad U. and Irani K., "On the Handling of Continuos-Valued Attributes in Decision Tree Generation", *Machine Learning* **8** (1992) 87–102
6. Frank E. and Witten I. H., "Generating Accurate Rule Sets Without Global Optimization", *Machine Learning: Proc. of the Fifteenth International Conference*, Morgan Kaufmann (1998) 144–151
7. Fürnkranz J., "Separate-and-Conquer Rule Learning", *Articial Intelligence Review* **13(1)** (1999)
8. Fürnkranz J. and Widmer G., "Incremental Reduced Error Pruning", *Machine Learning: Proc. of the Eleventh International Conference*, Morgan Kaufmann (1994)
9. Kohavi R. and Li C-H., "Oblivious Decision Trees, Graphs and Top-Down Pruning", *Proc. of the Fourteenth International Joint Conference on Artificial Intelligence*, Morgan Kaufmann (1995) 1071–1077
10. Lavrač N. and Džeroski S., *Inductive Logic Programming: Techniques and Applications*, Ellis Horwood (1994)
11. Lloyd J. W., *Foundations of Logic Programming*, (2nd edition), Springer-Verlag (1987)
12. Martin J. K., "An Exact Probability Metric for Decision Tree Splitting and Stopping", *Machine Learning* **28** (1997) 257–291
13. Pagallo G. and Haussler D., "Boolean Feature Discovery in Empirical Learning", *Machine Learning* **5** (1990) 71–99
14. Quinlan J. R., "Induction of Decision Trees", *Machine Learning* **1** (1986) 81–106
15. Quinlan J. R., "Learning Logical Definitions from Relations", *Machine Learning* **5** (1990) 239–266
16. Quinlan J. R., *C4.5: Programs for Machine Learning*, Morgan Kaufmann (1993)