Mach Learn (2008) 70: 121–133
DOI 10.1007/s10994-007-5029-3

QG/GA: a stochastic search for Progol

Stephen Muggleton · Alireza Tamaddoni-Nezhad

Received: 27 September 2007 / Accepted: 9 October 2007 / Published online: 9 November 2007
Springer Science+Business Media, LLC 2007

Abstract Most search techniques within ILP require the evaluation of a large number of inconsistent clauses. However, acceptable clauses typically need to be consistent, and are only found at the "fringe" of the search space. A search approach is presented, based on a novel algorithm called QG (Quick Generalization). QG carries out a random-restart stochastic bottom-up search which efficiently generates a consistent clause on the fringe of the refinement graph search without needing to explore the graph in detail. We use a Genetic Algorithm (GA) to evolve and re-combine clauses generated by QG. In this QG/GA setting, QG is used to seed a population of clauses processed by the GA. Experiments with QG/GA indicate that this approach can be more efficient than standard refinement-graph searches, while generating similar or better solutions.

Keywords Stochastic search · Refinement · Genetic Algorithms

1 Introduction

There is a long-standing and increasing interest in stochastic search methods in Inductive Logic Programming (ILP) (Page and Srinivasan 2003; Srinivasan 2005). Stochastic methods have been explored both for clause evaluation (e.g. Sebag and Rouveirol 2000) and for searching the space of candidate clauses (e.g. Kovacic 1994; Srinivasan 2000; Tamaddoni-Nezhad and Muggleton 2000; Ruckert and Kramer 2003; Zelezny et al. 2004; Paes et al. 2006).

Most search techniques within ILP are based on clause refinement. Such searches are typically time-consuming, requiring the testing of a large number of clauses. Acceptable

Editors: Ramon Otero, Simon Colton.

S. Muggleton (✉) · A. Tamaddoni-Nezhad
Department of Computing, Imperial College London, London, UK
e-mail: shm@doc.ic.ac.uk

A. Tamaddoni-Nezhad
e-mail: atn@doc.ic.ac.uk

Table 1 Number of consistent clauses (Cs) versus total number of clauses (All) considered by Progol4.5 in training over a range of Progol's example data sets

Data Set	Cs	All	Cs (%)
animals	16	44	36.4
append	16	100	16
arch	9	200	4.5
book	1	7	14.3
chess	16	2904	0.6
cyclic	26	200	13
delete	9	66	13.6
drug	101	8808	1.2
eleusis1	8	75	10.7
eleusis2	19	404	4.7
eleusis3	2	134	1.5
even	4	20	20
family	2	15	13.3
grammar	4	14	28.6
last	5	16	31.3
mult	61	636	9.6
nim	17	43	39.5
order0	51	200	25.5
order1	44	200	22
oeder2	25	119	21
order3	11	28	39.3
order4	14	28	50
order6	3	58	5.2
range	36	374	9.6
range1	6	23	26.1
reverse	19	72	26.4
set	6	51	11.8
train	7	2273	0.3
TOTAL	**538**	**17112**	**3.14**

clauses typically need to be consistent, and are only found at the "fringe" of the search (see Sect. 2.1).

Table 1 shows the number of consistent clauses considered out of all the clauses evaluated by Progol4.5 over a range of Progol's example data sets. The low average density of consistent clauses in Table 1 (around 3%) motivates an investigation in this paper into a novel algorithm called QG (Quick Generalization). QG carries out a random-restart stochastic bottom-up search which efficiently generates a consistent clause on the fringe of the refinement graph search without needing to explore the graph in detail.

QG has been implemented in Progol4.6 and has been compared with Progol's standard generalization search on a range of datasets. The results indicate that when the proportion of consistent clauses is small, QG is more efficient than standard refinement-graph searches, while still generating the same (or similar) solutions in most cases. In this paper we also examine a combination of QG with a Genetic Algorithm (GA). In this QG/GA setting, QG is used to seed a population of clauses processed by the GA. Experimental results suggest that a combination of QG and GA can provide higher predictive accuracy than each individual approach.

This paper is arranged as follows. Section 2 introduces the idea behind QG, gives the background and describes the QG algorithm. An analysis of the QG algorithm is also given in Sect. 2. Section 3 describes an implementation of QG and QG/GA in Progol. Section 4 presents the empirical evaluation of QG and QG/GA on a range of datasets and Sect. 5 concludes the paper.

2 Quick generalization (QG)

Mitchell (1997) notes that most Machine Learning algorithms can be viewed as approximations to the following general Bayesian statistical inference algorithms.

MAP—returns the hypothesis having maximum posterior probability.
Gibbs—randomly samples a consistent hypothesis according to the posterior distribution.
Bayes Prediction—classifies unseen instances based on weighted joint prediction of the entire consistent hypothesis space.

All of the above assume a Bayes' prior distribution over the hypothesis space. In the case of Gibbs this is used for sampling. We can easily imagine Gibbs-like approximations to MAP and Bayes Prediction, based on sampling. We might call these Gibbs-MAP and Gibbs-Bayes-Predictor. However, the key to effective implementation of such algorithms is the availability of algorithms for efficiently sampling from the set of consistent hypotheses. As indicated in Sect. 1 refinement-based algorithms, such as Progol, have severe difficulties in this respect since the density of consistent clauses within the hypothesis space is typically low. The QG algorithm described in this section is a Gibbs-MAP algorithm, which constructs maximally general consistent clauses by stochastically pruning Progol bottom clauses. The algorithm can be made arbitrarily efficient by choice of sample size (down to 1). We might also view Genetic Algorithms (GAs), Boltzman machines and even neural net algorithms as Gibbs-MAP algorithms. However, for every such algorithm there is an obvious trade-off between sample size and expected predictive accuracy. A sampling mechanism based on QG as well as the combination of QG with a GA are explored in Sect. 3. In (Haussler et al. 1994) the authors showed that average-case error bounds for Gibbs algorithms are comparable to other Bayesian algorithms. These theoretical results are consistent with the empirical results in this paper.

Before describing the details of the QG algorithm, we provide some mathematical preliminaries.

2.1 Mathematical preliminaries

The Progol algorithm (Muggleton 1995) is based on successive construction of definite clause hypotheses H from a language $\mathcal{L}$. H must explain the examples $\mathcal{E}$ in terms of background knowledge $\mathcal{B}$. Each clause in H is found by choosing an uncovered positive example e and searching through the graph defined by the refinement ordering $\succeq$ bounded below by the bottom clause associated with e. We define this setting more formally as follows.

Definition 1 (Progol refinement setting) Let $\mathcal{S} = \langle \mathcal{B}, \mathcal{E}, \mathcal{L}, \succeq \rangle$ be Progol's ILP setting as defined in (Muggleton 1995). Let $\mathcal{E} = \langle \mathcal{E}^+, \mathcal{E}^- \rangle$ consist of a set of positive and negative examples (ground unit clauses) respectively. The "top" clause, denoted by $\triangle$, is the maximal$_{\succeq,\mathcal{L}}$ element in $\mathcal{L}$. The "bottom" clause, denoted by $\triangledown_{e,\mathcal{S}}$, is the least$_{\succeq,\mathcal{L}}$ element such that $\mathcal{B}, \triangledown_{e,\mathcal{S}} \models e$. Refinement of clause C, denoted by $\rho_{e,\mathcal{S}}(C)$, is the set of maximal$_{\succeq,\mathcal{L}}$ clauses D such that $C \succ D \succeq \triangledown_{e,\mathcal{S}}$.

We now introduce the notion of consistency within the Progol setting $\mathcal{S}$ to be used in the QG algorithm.

Definition 2 ($\mathcal{S}$-consistency) Given $\mathcal{S} = \langle \mathcal{B}, \mathcal{E}, \mathcal{L}, \succeq \rangle$, $e \in \mathcal{E}$ and $\triangle \succeq C \succeq \triangledown_{e,\mathcal{S}}$ we say that C is $\mathcal{S}$-consistent iff $\mathcal{B}, \mathcal{E}, C$ is satisfiable.

It is normal in ILP to restrict attention to clausal hypotheses which are "head-connected" in the following sense.

Definition 3 (Head-connectness) A definite clause $h \leftarrow b_1, \ldots, b_n$ is said to be head-connected iff each body atom b_i contains at least one variable found either in h or in a body atom b_j, where $1 \leq j < i$.

Next we introduce the idea of a "minimal support set", which is a set of body atoms which is an irreducible set of atoms which ensure that a clause is head-connected.

Definition 4 (Minimal support set) Let $h \leftarrow B$ be a definite clause and B be a set of atoms. $S \subseteq B$ is a minimal support set for b from B iff $h \leftarrow S, b$ is head-connected and there does not exist a set $S' \subset S$ for which $h \leftarrow S', b$ is head-connected.

Lastly, we introduce the notion of the "fringe" set of maximally general clauses in the hypothesis space, from which QG samples.

Definition 5 (Fringe) Clause C is in Fringe($e, \mathcal{S}$) iff it is head-connected and for every D it is the case that $C \in \rho_{e,\mathcal{S}}(D)$ implies D is not $\mathcal{S}$-consistent.

2.2 QG algorithm

The QG algorithm (see Fig. 1) works by finding successively smaller consistent subsets of a Progol-style bottom clause. The notion of a cutoff atom is used within the algorithm to define the minimal consistent prefix of a given clause body.

Definition 6 (Profile and cutoff atom) Let $C = h \leftarrow b_1, \ldots, b_n$ be a definite clause and $\mathcal{B}$ be background knowledge. $E_i \subseteq \mathcal{E}^-$ is the ith negative profile of C, where $E_i = \{e : \exists\theta, e = h\theta, \mathcal{B} \models (b_1, \ldots, b_i)\theta\}$. b_i is the cutoff atom iff i is the least value such that $E_i = \emptyset$.

The QG algorithm works by randomly permuting the given clause body and then applying to the result the deterministic "Reduce" algorithm.[1] The result is a randomly constructed "fringe" clause (see Definition 5).

[1]The Reduce algorithm was first introduced in the Golem system (Muggleton and Feng 1990).

```
Quick Generalization (QG) algorithm
    Input: Bottom clause $\bigtriangledown_{e,\mathcal{S}}$ and setting $\mathcal{S}$
      $R$ is a random head-connected permutation of $\bigtriangledown_{e,\mathcal{S}}$
    Output: Reduce $R$ wrt $\mathcal{S}$
Reduce algorithm
    Input: Clause $C = h \leftarrow b_1, \ldots, b_n$ and setting $\mathcal{S}$
      $Res$ is $C$
      While there is an unseen cutoff atom $b_i$ in the body of $Res$
        For $b_i$ find minimal support set $S_i = \{b'_1, \ldots, b'_m\} \subseteq \{b_1, \ldots, b_{i-1}\}$
         such that $h \leftarrow S_i, b_i$ is head-connected
        $Res$ is $h \leftarrow S_i, b_i, T_i$
         where $T_i$ is $b_1, \ldots, b_{i-1}$ with $S_i$ removed
      Repeat
    Output: Reduced clause $Res$
```

Fig. 1 Quick Generalization (QG) algorithm

2.3 QG analysis

Let us now consider the correctness of the QG algorithm.

Theorem 1 (Correctness of QG) *Let $\bigtriangledown_{e,\mathcal{S}} = h \leftarrow b_1, \ldots, b_n$ be a bottom clause of example e with associated setting $\mathcal{S}$. The clause $F = QG(\bigtriangledown_{e,\mathcal{S}}, \mathcal{S})$ is in Fringe$(e, \mathcal{S})$.*

Proof Assume $F \notin \text{Fringe}(e, \mathcal{S})$. In this case, either F is not head-connected or there exists a clause D such that $F \in \rho_{e,\mathcal{S}}(D)$ and D is $\mathcal{S}$-consistent. However, since by construction QG returns a head-connected clause F which is a subset of $\bigtriangledown_{e,\mathcal{S}}$, then assuming completeness of $\rho_{e,\mathcal{S}}$ it follows that D must exist. Thus D must be $\mathcal{S}$-consistent. Since $D \subset F$ it follows that there must be an atom b_j found in the body of F and not in the body of D. However, by construction every body atom b_j in F was either a cutoff atom or in a minimal support set in one of the iterations of the Reduce algorithm, and by definition the removal of a cutoff atom makes the resultant clause not $\mathcal{S}$-consistent. This implies D is not $\mathcal{S}$-consistent, which contradicts the assumption and completes the proof. □

Let us now consider whether every element of the fringe associated with a given example can be generated by QG.

Theorem 2 (Completeness of QG) *Let $\bigtriangledown_{e,\mathcal{S}} = h \leftarrow b_1, \ldots, b_n$ be a bottom clause of example e with associated setting $\mathcal{S}$. For each $F \in$ Fringe$(e, \mathcal{S})$ there exists a stochastic derivation such that $F = QG(\bigtriangledown_{e,\mathcal{S}}, \mathcal{S})$.*

Proof Assume false. Thus there exists $F \in \text{Fringe}(e, \mathcal{S})$ such that there is no stochastic derivation such that $F = QG(\bigtriangledown_{e,\mathcal{S}}, \mathcal{S})$. Assume $F = h \leftarrow b_1, \ldots, b_j$. Now suppose that in QG the random head-connected permutation R of $\bigtriangledown_{e,\mathcal{S}}$ is $R = h \leftarrow b_1, \ldots, b_j, \ldots, b_n$, i.e. the body of F is a prefix of the body of R. However, in this case, since $F \in \text{Fringe}(e, \mathcal{S})$ it follows that F is $\mathcal{S}$-consistent and that b_j is a cutoff atom. In this case the Reduce algorithm will return F after $j + 1$ cycles. This refutes the assumption and completes the proof. □

1. eastbound(A) :- has_car(A,B), infront(A,B), has_car(A,C), has_car(A,D), has_car(A,E), load(D,circle,1), long(B), short(D), infront(C,E), shape(B,rectangle), load(E,hexagon,1), closed(C), wheels(B,2), open(B), short(C), long(E), infront(B,C), wheels(C,2), shape(D,rectangle), open(D), infront(E,D), shape(E,rectangle), wheels(E,3), load(B, rectangle,3), wheels(D,2), open(E), load(C,triangle,1), shape(C,rectangle).
2. eastbound(A) :- has_car(A,B), load(B,hexagon,1), has_car(A,C), infront(A,C), has_car(A,D), has_car(A,E), load(E,circle,1), long(C), short(E), infront(D,B), shape(C,rectangle).
3. eastbound(A) :- has_car(A,B), load(B,hexagon,1).

Fig. 2 An example of the iterative reduction of a clause by the QG algorithm

We lastly consider the complexity of QG in terms of the number of cycles required for completion.

Theorem 3 (Cycle-complexity of QG) *Let $\nabla_{e,\mathcal{S}} = h \leftarrow b_1, \ldots, b_n$ be a bottom clause of example e with associated setting $\mathcal{S}$. The clause $F = h \leftarrow b_1, \ldots, b_m = QG(\nabla_{e,\mathcal{S}}, \mathcal{S})$ is returned after at most $m + 1$ cycles of the Reduce algorithm.*

Proof By construction each atom b_j in the body of $F = h \leftarrow b_1, \ldots, b_m$ was either a cutoff atom or an element of one of the minimal support sets in one of the cycles of the Reduce algorithm. Thus, since the last atom is a cutoff atom twice, the Reduce algorithm returns F in at most $m + 1$ cycles. □

Assuming time-bounded theorem proving of the kind used in Progol, Theorem 3 indicates that the time complexity of QG is linear in the size of the returned clause.

2.4 Example QG executions

Figure 2 gives an illustration of the cycle-complexity result given in Theorem 3. In the first iteration the atom *load(B,hexagon,1)* is identified as the cutoff atom. The atom is placed immediately after the atom *has_car(A,B)* in order to ensure that variable B has been introduced before being used. In the second iteration *load(B,hexagon,1)* is once more identified as the cutoff atom. Since this atom has already been seen, the algorithm terminates and returns the resulting clause in step 3.

3 QG and QG/GA in Progol

The QG algorithm described in the previous sections can be used for efficiently sampling from consistent clauses. Clauses generated by QG are consistent but not necessarily compressive wrt the positive examples and they may have small or even no positive coverage. On the other hand, an ILP system such as Progol generates hypotheses which provide a positive compression over training data (Muggleton 1995). A simple integration of QG in Progol can be realized by replacing Progol's A^* search by a QG sampling mechanism. This involves an algorithm, QG-sample, which returns the clause with highest positive compression from a sample of S calls to QG. In this mechanism the sample size S is set by the user so that at

Table 2 Binary encoding of the occurrences of literals in a clause wrt a bottom clause. (a) a bottom clause (b) binary encoding of clause eastbound(A):- has_car(A,C), closed(C), short(C)

(a)

eastbound(A):-	has_car(A,B)	has_car(A,C)	closed(C)	open(B)	short(C)	...
1	1	1	1	1	1	...

(b)

1	0	1	1	0	1	...
eastbound(A):-		has_car(A,C)	closed(C)		short(C)	...

least one of the clauses has positive compression. In this setting, the algorithm simply returns a consistent clause with the highest positive compression. A good sample size can be estimated based on the percentage of consistent clauses which have positive compression. An example of choosing a sample size for QG is given in Sect. 4.1. Clauses generated by the simple QG setting described above lack diversity and also it is likely that the optimal solutions are not among them. In this paper we also examine a more advanced setting in which a Genetic Algorithm (GA) is used to evolve and re-combine clauses generated by QG. In this setting QG is used to seed a population of clauses processed by the GA. The GA-ILP setting used in the present study is similar to the one described in (Tamaddoni-Nezhad and Muggleton 2002). In this framework, encoding of hypotheses is based on a most specific clause (or bottom clause) which is automatically constructed using an ILP method (e.g. inverse entailment). In the GA implementation used in the present study, the occurrences of literals from the bottom clause are directly encoded as bit strings as described as further work in (Tamaddoni-Nezhad and Muggleton 2002). Table 2 shows how the occurrences of literals in a clause is encoded as a bit-string. In a "GA only" setting, the initial population of the GA consists of randomly generated bit-strings with length L, where L is the number of literals in the bottom clause. In the QG/GA setting, the initial population of the GA consists of clauses generated by the QG algorithm. As the QG algorithm generates clauses from the permutations of the same bottom clause, encoding these clauses into bit-strings is straightforward and follows the same scheme as shown in Table 2. In order to provide a better comparison, the fitness value used in GA and QG/GA is the same as the evaluation function of A^* (i.e. the compression provided by each clause).

4 Empirical evaluation

4.1 Experiment 1

The goal of this experiment is to compare the performance of QG and A^* on the datasets shown in Fig. 1. In particular we examine the following null hypothesis:

Null hypothesis 1: On our benchmark problems QG cannot provide increased efficiency over Progol's standard A^* search without decrease in accuracy.

4.1.1 Material and methods

In this experiment we use a set of learning problems from Progol4.1 example datasets.[2] This includes an early version of the mutagenesis dataset (drug) which in this experiment is replaced with the mutagenesis 42 dataset with atom-bond background knowledge (King et al. 1996). We use a leave-one-out test strategy and compare total time and predictive accuracies for the following algorithms:

A^*—Progol's standard refinement-graph search.
QG—An implementation of the QG-sample algorithm as described in Sect. 3.

In this experiment the sample size for the QG-sample algorithm is estimated based on the percentage of consistent clauses which have positive compression. These have been estimated from the output of single runs of QG (e.g. with sample size 10) on each dataset and then used for the leave-one-out experiments. For example, if 1 out of 5 consistent clauses have positive compression the probability of QG-sample finding a consistent and compressive clause is $1 - (4/5)^S$ where S is the sample size. By choosing $S = 10$ this probability will be around 0.9. In most problems considered in this experiment, clauses generated by QG all have positive compression (i.e. the sample size can be set to one).

4.1.2 Results

Table 3 shows predictive accuracies and total learning and testing times for A^* and QG. This table also shows an estimation for the density of consistent clauses based on the proportion of consistent clauses considered by the A^* search. According to this table, in most cases QG finds a solution with the same or similar accuracies in less time. Null hypothesis 1 is therefore rejected. These results also suggest that the efficiency advantage of QG is more evident when the density of consistent clauses is small. For example, in the train problem, in which only 0.3% of clauses are consistent, QG is far more efficient that A^* (i.e. around 26 times faster) while still generating the same solution. As one might expect from general considerations it appears that the ratio T_{A^*}/T_{QG} increases with decreasing density of consistent clauses.

4.2 Experiment 2

In this experiment we examine a combination of QG and GA as described in Sect. 3.

Null hypothesis 2: On our benchmark problems a combination of QG and GA cannot provide higher predictive accuracy than each individual approach.

4.2.1 Material and methods

In this experiment we use the mut42 dataset from the previous experiment. We use a leave-one-out test strategy and compare average time and predictive accuracies for the following algorithms:

A^*—Progol's standard refinement-graph search.
QG—An implementation of the QG-sample algorithm as described in Sect. 3.

[2] Available from:http://www.doc.ic.ac.uk/~shm/Software/progol4.1/.

Table 3 Predictive accuracies (Acc) and total learning times from leave-one-out experiments for A^* and QG. Density of consistent clauses is taken as being the proportion of consistent clauses in the A^* search ($Cs(\%)$)

Dataset	A^*			QG		
	Cs (%)	$Acc(\%)$	$T_{A^*}(s)$	$Acc(\%)$	$T_{QG}(s)$	T_{A^*}/T_{QG}
animals	36.4	93 ± 3	3.01	95 ± 3	2.68	1.12
append	16	96 ± 4	1.46	100 ± 0	0.56	2.61
arch	4.5	50 ± 18	0.87	50 ± 18	0.20	4.35
book	14.3	100 ± 0	0.03	100 ± 0	0.04	0.75
chess	0.6	23 ± 8	69.32	8 ± 4	4.94	14.03
cyclic	13	40 ± 22	0.12	40 ± 22	0.05	2.00
delete	13.6	50 ± 18	0.40	13 ± 11	0.10	4.00
eleusis1	10.7	88 ± 6	0.52	92 ± 5	0.24	1.53
eleusis2	4.7	65 ± 9	3.05	69 ± 8	0.33	2.28
eleusis3	1.5	87 ± 6	2.14	87 ± 6	0.24	2.40
even	20	87 ± 6	0.59	48 ± 9	0.13	1.16
family	13.3	80 ± 8	0.17	84 ± 7	0.62	0.27
grammar	28.6	100 ± 0	0.11	100 ± 0	0.08	1.38
last	31.3	50 ± 16	0.11	50 ± 16	0.17	0.65
mult	9.6	96 ± 3	21.56	94 ± 3	5.90	3.65
mut42	0.1	86 ± 5	6226.03	86 ± 5	378.34	16.46
nim	39.5	96 ± 4	0.4	30 ± 9	1.07	0.03
order0	25.5	94 ± 5	1.95	100 ± 0	0.49	3.98
order1	22	100 ± 0	2.02	94 ± 5	1.22	1.66
order2	21	100 ± 0	0.6	92 ± 8	0.88	0.68
order3	39.3	100 ± 0	0.19	92 ± 7	0.48	0.40
order4	50	100 ± 0	0.43	94 ± 6	1.84	0.23
order6	5.2	38 ± 17	0.23	50 ± 17	0.21	0.09
range	9.6	90 ± 6	6.07	62 ± 10	0.77	7.8
range1	26.1	70 ± 14	0.07	70 ± 14	0.03	2.33
reverse	26.4	85 ± 8	0.93	65 ± 10	0.82	1.13
set	11.8	54 ± 10	1.51	54 ± 10	1.57	0.96
train	0.3	100 ± 0	6.07	100 ± 0	0.23	26.39

GA—A Genetic Algorithm (GA) as described as "GA only" in Sect. 3.

QG/GA—A GA-based search in which QG has been used for seeding the GA population as described in Sect. 3.

In this experiment N, the maximum number of clauses evaluated for learning a single clause, is varied for each algorithm. For A^*, N corresponds to Progol's parameter *nodes*. In QG, N is the sample size. In GA and QG/GA, $N = popsize \times (maxgen + 1)$. Other GA parameters used in this experiment are as follows: $p_m = 0.01$, $p_c = 0.6$ and $popsize = 30$. For A^*, the maximum number of literals in each clause is set to 4.

Table 4 Predictive accuracies and average learning times from leave-one-out experiments on mut42 dataset. N is the maximum number of clauses evaluated for learning a single clause

N	A^*		QG		GA		QG/GA	
	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$
2	69 ± 7	0.12	76 ± 7	2.36				
5	69 ± 7	0.13	81 ± 6	4.90				
10	69 ± 7	0.13	86 ± 5	9.30				
20	76 ± 7	0.11	86 ± 5	20.79				
30	81 ± 6	0.11	86 ± 5	27.60	83 ± 6	2.82	83 ± 6	27.68
60	81 ± 6	0.18	86 ± 5	54.70	83 ± 6	2.83	83 ± 6	27.68
120	81 ± 6	0.41	83 ± 6	116.51	81 ± 6	5.50	81 ± 6	31.12
240	81 ± 6	1.09	86 ± 5	228.78	81 ± 6	8.94	81 ± 6	35.89
480	81 ± 6	1.84	86 ± 5	469.08	83 ± 6	13.40	85 ± 5	39.61
960	81 ± 6	4.27	83 ± 6	930.82	83 ± 6	25.85	83 ± 6	47.54
1920	81 ± 6	9.87	83 ± 6	1861.35	83 ± 6	43.46	86 ± 5	64.07
3840	81 ± 6	25.10	83 ± 6	3773.29	83 ± 6	58.49	88 ± 5	85.95
7680	83 ± 6	50.76	83 ± 6	7686.96	86 ± 5	99.98	88 ± 5	106.79
15360	86 ± 5	141.06	83 ± 6	13916.10	86 ± 5	226.78	88 ± 5	165.00

4.2.2 Results

Table 4 summarizes the results of the leave-one-out experiments. This table shows predictive accuracies and average learning and testing times for each example. According to the table, QG finds relatively good solutions right at the beginning when the sample size is small. However, the predictive accuracies have not been improved with increased sample size. The figures even suggest that the accuracies have been decreased possibly due to overfitting the training data. Unlike for QG, predictive accuracies for GA and QG/GA increase with N. In general, QG/GA reaches the highest level of accuracy with fewer evaluated clauses. However, as the computational costs for each clause is relatively low in this problem (e.g. clauses considered by A^* are maximum 4 literals long), the efficiency advantage of QG/GA is not significant in this problem. Given the relatively high accuracy errors in the leave-one-out experiments, the accuracy advantage of QG/GA is also not significant. Null hypothesis 2 is therefore not rejected by this experiment. Nevertheless, the lessons learned in this experiment lead to an experimental setting which will be used in the next experiment.

4.3 Experiment 3

The results of the previous experiment show that QG/GA needs to evaluate fewer clauses in order to reach the same level of accuracy. This observation suggest that QG/GA could lead to a significant speed up in cases where the average computational costs for each clause is relatively high or the clauses which need to be considered during the search are relatively long. In this experiment we evaluate QG and QG/GA on a set of problems with different concept sizes. Both Null hypothesis 1 and Null hypothesis 2 are examined in this experiment.

4.3.1 Material and methods

In this experiment we use a set of seven artificially generated learning problems with varying concept sizes from 6 to 16. These problems are selected from the phase transition

Table 5 Predictive accuracies and learning times for different search algorithms on a set of learning problems with varying concept sizes from 6 to 16. Density of consistent clauses is taken as being the proportion of consistent clauses in the A^* search ($Cs(\%)$)

m	A^*			QG		GA		QG/GA	
	$Cs(\%)$	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$	$A(\%)$	$T(s)$
6	31.71	98	3.22	99.5	3.89	99.5	5.83	99.5	10.32
7	3.36	99.5	633.16	99.5	45.11	99.5	12.99	99.5	86.51
8	1.05	100	1416.55	100	175.03	100	13.92	100	169.55
10	0.0015	97.5	25852.80	99	242.22	95.5	74.68	99	1064.22
11	0.36	80	37593.20	91	774.02	99	30.37	99.5	110.15
14	0	50	128314.00	69	4583.25	79.5	529.67	88.5	1184.76
16	4×10^{-6}	59	55687.44	77.5	4793.01	74	297.93	89.5	4945.20

study (Botta et al. 2003) and correspond to problems $m6.l12$ to $m16.l12$.[3] Each problem includes 100 training and 100 test examples. We use a hold-out test strategy and compare the performance of the same algorithms as in the previous experiment (i.e. A^*, QG, GA and QG/GA). The GA parameters used in this experiment are the same as the ones used in the previous experiment except that $maxgen$ is not varied and it is set to 20 for all experiments. The A^* parameter $nodes$ is also fixed and set to 10000.

4.3.2 *Results*

Table 5 shows predictive accuracies and average learning and testing times for different algorithms. According to this table, QG has found a solution with a similar or better accuracy than the A^* search in significantly less time. Null hypothesis 1 is therefore rejected. These results also suggest that we can get a better predictive accuracy by combining QG and GA (e.g. unlike A^* and QG, QG/GA passes the 80% accuracy criteria of (Botta et al. 2003) for $m = 14$ and $m = 16$). This refutes Null hypothesis 2. According to the table, the efficiency and accuracy advantages of QG and QG/GA are more evident when the density of consistent clauses ($Cs\%$) is low. It is expected that A^* will be more efficient than other algorithms for small values of m. However, this cannot be tested on the present dataset as it does not include problems with small values for m.

5 Conclusions

In this paper we presented a search approach based on a novel algorithm called QG (Quick Generalization). QG efficiently samples consistent clauses on the fringe of the refinement graph search without needing to explore the graph in detail. We proved that the QG algorithm is correct (i.e. only fringe clauses are returned by QG), complete (i.e. all fringe clauses can be returned by QG) and efficient (i.e. QG returns a solution in time linear to the size of the solution). A sampling mechanism based on QG has been implemented in Progol4.6 and has been tested on a range of learning problems. The results of comparison between QG and Progol's standard refinement-graph search (A^*) suggest that in

[3]These problems are selected from the first row of the (m, L) plane so that they only approach the phase transition region.

some cases QG returns solutions with similar accuracies to A^* in less time. Time reduction by QG is substantial when the density of consistent clauses is low. In this paper we also examined a combination of QG with a Genetic Algorithm (GA). In this QG/GA setting, QG is used to seed a population of clauses processed by the GA. Experimental results suggest that QG/GA needs to evaluate fewer clauses than A^* and GA in order to reach the same level of accuracy. QG and QG/GA have been also tested on a set of problems from the phase transition dataset (Botta et al. 2003) which includes learning problems with varying concept sizes. In this experiment, QG/GA significantly outperforms other searches. The QG algorithm can be easily adopted for any ILP system which uses a bottom clause or a template for generating the hypotheses. These include ILP systems which use some form of Inverse Entailment (e.g. Ito and Yamamoto 1998; Inoue 2001 and Ray et al. 2003). As future work, we intend to explore methods for dealing with classification noise by relaxing the requirements of consistency in clauses generated by the QG algorithm. This can be realized by allowing a threshold (more than zero) for negative examples that can be covered by each clause.

References

Botta, M., Giordana, A., Saitta, L., & Sebag, M. (2003). Relational learning as search in a critical region. *Journal of Machine Learning Research*, *4*, 431–463.

Haussler, D., Kearns, M., & Shapire, R. (1994). Bounds on the sample complexity of Bayesian learning using information theory and the VC dimension. *Machine Learning*, *14*(1), 83–113.

Inoue, K. (2001). Induction, abduction and consequence-finding. In C. Rouveirol & M. Sebag (Eds.), *Lecture notes in artificial intelligence: Vol. 2157. Proceedings of the eleventh international workshop on inductive logic programming (ILP01)* (pp. 65–79). Berlin: Springer.

Ito, K., & Yamamoto, A. (1998). Finding hypotheses from examples by computing the least generalization of bottom clauses. In S. Arikawa & H. Motoda (Eds.), *Lecture notes in artificial intelligence: Vol. 1532. Proceedings of discovery science '98* (pp. 303–314). Berlin: Springer.

King, R., Muggleton, S., Srinivasan, A., & Sternberg, M. (1996). Structure-activity relationships derived by machine learning: the use of atoms and their bond connectives to predict mutagenicity by inductive logic programming. *Proceedings of the National Academy of Sciences*, *93*, 438–442.

Kovacic, M. (1994). *Stochastic inductive logic programming*. PhD thesis, University of Ljubljana, Ljubljana, Slovenia.

Mitchell, T. (1997). *Machine learning*. New York: McGraw-Hill.

Muggleton, S. (1995). Inverse entailment and Progol. *New Generation Computing*, *13*, 245–286.

Muggleton, S. H., & Feng, C. (1990). Efficient induction of logic programs. In *Proceedings of the first conference on algorithmic learning theory* (pp. 368–381). Tokyo: Ohmsha.

Paes, A., Zelezny, F., Zaverucha, G., Page, D., & Srinivasan, A. (2006). ILP through Propositionalization and Stochastic *k*-term DNF learning. In S. Muggleton, R. Otero, & A. Tamaddoni-Nezhad (Eds.), *Proceedings of the 16th international conference on inductive logic programming* (pp. 379–393). Berlin: Springer.

Page, D., & Srinivasan, A. (2003). ILP: a short look back and a longer look forward. *Journal of Machine Learning Research*, *4*, 415–430.

Ray, O., Broda, K., & Russo, A. (2003). Hybrid abductive inductive learning: a generalization of Progol. In *Lecture notes in artificial intelligence: Vol. 2835. 13th international conference on inductive logic programming* (pp. 311–328). Berlin: Springer.

Ruckert, U., & Kramer, S. (2003). Stochastic local search in *k*-term DNF learning. In *Proceedings of the 20th international conference on machine learning* (pp. 648–655).

Sebag, M., & Rouveirol, C. (2000). Resource-bounded relational reasoning: Induction and deduction through stochastic matching. *Machine Learning*, *38*, 43–65.

Srinivasan, A. (2000). *A study of two probabilistic methods for searching large spaces with ILP* (Technical Report PRG-TR-16-00). Oxford University Computing Laboratory, Oxford.

Srinivasan, A. (2005). Five problems in five areas for five years. In S. Kramer & B. Pfahringer (Eds.), *Lecture notes in artificial intelligence: Vol. 3625. Proceedings of the 15th international conference on inductive logic programming* (p. 424). Berlin: Springer.

Tamaddoni-Nezhad, A., & Muggleton, S. H. (2000). Searching the subsumption lattice by a genetic algorithm. In J. Cussens & A. Frisch (Eds.), *Proceedings of the 10th international conference on inductive logic programming* (pp. 243–252). Berlin: Springer.

Tamaddoni-Nezhad, A., & Muggleton, S. H. (2002). A genetic algorithms approach to ILP. In *Proceedings of the 12th international conference on inductive logic programming* (pp. 285–300). Berlin: Springer.

Zelezny, F., Srinivasan, A., & Page, D. (2004). A Monte Carlo study of randomised restarted search in ILP. In *Lecture notes in artificial intelligence: Vol. 3194. Proceedings of the 14th international conference on inductive logic programming* (pp. 341–358). Berlin: Springer.