Performance and Efficiency of Memetic Pittsburgh Learning Classifier Systems

Jaume Bacardit jqb@cs.nott.ac.uk
ASAP research group, School of Computer Science, Jubilee Campus, Nottingham, NG8 1BB and Multidisciplinary Centre for Integrative Biology, School of Biosciences, Sutton Bonington, LE12 5RD, University of Nottingham, UK

Natalio Krasnogor nxk@cs.nott.ac.uk
ASAP research group, School of Computer Science, University of Nottingham, Jubilee Campus, Nottingham, NG8 1BB, UK

Abstract

In this paper we empirically evaluate several local search (LS) mechanisms that heuristically edit classification rules and rule sets to improve their performance. Two kinds of operators are studied, (1) *rule-wise* operators, that edit individual rules, and (2) a *rule set-wise* operator, which takes the rules from N parents ($N \geq 2$) to generate a new offspring, selecting the minimum subset of candidate rules that obtains maximum training accuracy. Moreover, various ways of integrating these operators within the evolutionary cycle of Learning Classifier Systems are studied. The combinations of LS operators and policies are integrated in a Pittsburgh approach framework that we call MPLCS for Memetic Pittsburgh Learning Classifier System. MPLCS is systematically evaluated using various metrics. Several datasets were employed with the objective of identifying which combination of operators and policies scale well, are robust to noise, generate compact solutions and use the least amount of computational resources to solve the problems.

Keywords

Learning Classifier Systems, Pittsburgh Approach, Memetic Algorithms.

1 Introduction

Recent advances in Evolutionary Computation have been achieved through a variety of innovations such as *Estimation of Distribution Algorithms (EDA)* [Larranaga and Lozano, 2002] or *Memetic Algorithms (MA)* [Krasnogor and Smith, 2005], among others. Some of these techniques estimate a model of the structure of the problem and then generate offspring according to this model. Other techniques combine local search (LS) and global search. This line of research has recently gained interest in the Learning Classifier Systems (LCS) community [Butz, 2004, Llorà et al., 2006], where both Michigan and Pittsburgh methods have been extended with exploration mechanisms based on EDAs.

In a recent paper [Bacardit and Krasnogor, 2006] we proposed a local search based smart crossover operator for GAssist [Bacardit, 2004], a Pittsburgh LCS. This operator takes the rules of N parents ($N \geq 2$) and heuristically selects the minimum subset of rules that obtains maximum accuracy over the training set, also deciding the order in which the rules are to be evaluated. This operator exploits the fact that Pittsburgh LCSs apply a supervised learning paradigm and therefore a large volume of performance

information about the rules is available, which can be used by the local search mechanism to improve the learning performance. That is, given that it is possible to determine the fitness contribution of a given rule to the global solution, an intelligent recombination of the rules available in the population at a given time can be enacted.

We have shown [Bacardit and Krasnogor, 2006] that using this kind of intelligent recombination GAssist can produce rule sets with better accuracy, on 25 real world problems from the UCI repository [Blake et al., 1998]. Also, when using the smart crossover operator it converges in less learning steps than when using the naive crossover to equivalent quality solutions, in synthetic datasets such as the 11 bit multiplexer.

There were two important lines of research that were not addressed in our previous work: (1) The previous operator could not edit rules, only recombine them. Thus a further step would be the development of *rule-wise* local search techniques, and their integration with the *rule set-wise* operator proposed in our previous work and (2) systematically evaluate the local search methods to improve their convergence and to identify and remove unnecessary computational effort.

In this paper we address, systematically, both challenges. First, a set of rule-wise representation-dependant local search operators are analyzed in isolation and also in combination with the, rule representation neutral, smart crossover operator. These operators exploit performance information obtained over the training data to improve their capacity at finding good rules. We employ the *GABIL* [DeJong and Spears, 1991] knowledge representation which uses semantically rich predicates, thus providing us with a large amount of performance information. Secondly, several strategies for integrating the LS operators within the evolutionary cycle are studied.

The combinations of these operators and policies are integrated into a new Pittsburgh approach framework that we call Memetic Pittsburgh Learning Classifier System (MPLCS). This framework hybridizes GAssist with LS operators. Although these operators are integrated and evaluated within the context of a Pittsburgh LCS, their design is bound only to the representation, not to the employed learning paradigm, hence they are of general applicability to any machine learning method that relies on rules with a representation similar to GABIL, e.g., [Llora et al., 2005, Casillas et al., 2007].

Several variants of MPLCS are proposed and are empirically analyzed in a large-scale evaluation process. The objective of this evaluation process is to determine the performance of the different MPLCS variants in order to identify which of them can learn better and more efficiently, according to a variety of performance metrics including standard LCS metrics such as number of learning steps and checking how similar is the generated solution (rule set) to the optimal solution for the domain.

A diverse set of problems are used in this evaluation process to analyze various kind of challenges such as scalability, robustness to noise or unbalanced problem structures. The performance of the best MPLCS settings is also compared against state-of-the-art LCS methods, and the overall results of this study are analyzed from several points of view, identifying which operators act synergistically with the standard evolutionary computation exploration mechanisms. Also, We briefly mention the potential applicability of these operators to other LCSs.

The rest of the paper is structured as follows: section 2 overviews related work. Section 3 summarizes the main characteristics of GAssist, the Pittsburgh LCS used as base for the work of this paper. Section 4 defines the rule-wise and rule set-wise local search operators and the hybridization policies used in MPLCS. Section 5 describes the experimental design used in this paper while section 6 shows the results of the empirical evaluation of our approach. Section 7 contains a global discussion about the obtained

results. Finally, section 8 presents the conclusions and future work.

2 Related work

Intelligent recombination, local search and statistical learning techniques are some of the strategies used to produce better and faster EC algorithms. Different families of techniques using these strategies exist in the literature.

The work presented in this paper, hybridizing global and local search, is part of one of such families, Memetic Algorithms [Krasnogor and Smith, 2005]. These methods are inspired by models of natural systems that combine the evolutionary adaptation of a population with individual learning within the lifetimes of its members.

In this scope, there is a quite early LCS work, the SAMUEL system [Grefenstette, 1991], that has an operator which is similar to one of the LS operators presented here. That system was applied to multi-step domains, and their operator generated an offspring containing high-payoff rules that fired in sequence. Unlike our approach, this operator only used rules from two parents. Our previous work showed that using rules from multiple parents contributed to the generation of better offspring. Also, it was applied to unordered rules, while our approach is specifically designed for ordered rules. This is an important distinction as our rule-set wise LS operator is designed for ordered rule sets. More recently, Wyatt and Bull [Wyatt and Bull, 2004] proposed a Memetic Learning Classifier System, within the Michigan paradigm of LCS, for representation with real-valued attributes. Also, Butz et al. proposed a gradient descent method to improve XCS's [Wilson, 1995] performance on multi-step domains.

Another of such families of techniques are the *Estimation of distribution algorithms*. Usually these paradigms involve applying machine learning or statistics techniques to learn the structure of the problem being solved and allow the system to explore the search space better by creating informed exploration operators.

For example, in [Butz, 2004], the author extended XCS with a crossover operator based on two types of EDAs: the Extended Compact Genetic Algorithm (ECGA) [Harik, 1999] and the Bayesian Optimization Algorithm [Pelikan et al., 1999]. These two methods derive global structural information from the best rules in the population, which later is used to inform the crossover operator when generating new offspring.

The Compact Classifier System (CCS) [Llora et al., 2005], is a recent integration of EDAs within the framework of a Pittsburgh LCS, using the Compact Genetic Algorithm (CGA) [Harik et al., 1999]. CGA is run iteratively to generate different rules. Different perturbations of the initial solution of CGA are needed to generate different rules, and the individuals in CCS store a set of such perturbations. The objective of CCS is to determine the minimum set of rules that creates a maximally general solution. Other early attempts at using EDAs in a Pittsburgh LCS are also discussed in [Llora et al., 2005].

Recently, Llorà et al. proposed $\chi eCCS$ [Llorà et al., 2006], an extension of CCS. $\chi eCCS$ evolves a population of rules using the model building and recombination mechanisms of ECGA. A niching method using Restricted Tournament Selection [Harik, 1995] was employed to guarantee that the population learns all the rules needed to solve the domain. Experiments showed that this method scales quadratically in relation to the problem size for the Multiplexer family of problems.

Finally, the editing procedure of the rule-wise local search operators studied in this paper are related to those originally proposed in [Janikow, 1991]. Janikow's GIL system introduced three operators (RuleSplit, ReferenceExtension, ReferenceRestriction) that edit the rules in the same way that our operators do. However, these operators are applied blindly (although having a probability of application biased by the rule's previous

performance) to all the rules of the population, modifying both good rules and bad rules. Our operators, on the other hand, using the information extracted from their previous performance, are applied only to the rules that actually *need* some editing to improve their performance.

3 The GAssist Learning Classifier System

GAssist [Bacardit, 2004] is a Pittsburgh LCS, originally inspired by GABIL [DeJong and Spears, 1991] and extended through the years with new features [Bacardit and Garrell, 2007, Bacardit et al., 2004, Bacardit, 2005]. Currently it also has similarities to the GIL [Janikow, 1991] system. GAssist applies a near-standard generational GA that evolves individuals representing a complete solution to the classification problem at hand. An individual consists of an ordered, variable-length rule set. A fitness function based on the Minimum Description Length (MDL) principle [Rissanen, 1978] is used. The MDL principle is a metric that can be applied to a theory (being a rule set here) which balances the complexity and accuracy of the rule set. As we deal with variable-length individuals, our system is sensitive to the bloat effect [Langdon, 1997], that is, the growth without control of the size of the individuals. Hence, the need for a fitness function that balances accuracy and complexity of the rule sets. The details and rationale of this fitness formula are explained in [Bacardit, 2004].

Moreover, in order to help the control of the bloat effect, we also use a rule deletion operator to remove rules that do not match any training example. The operator is applied after the fitness computation and has two constraints: (a) the process is only activated after a predefined number of iterations (to prevent an irreversible diversity loss) and (b) the operator is not applied if its application would produce an individual having a number of rules smaller that a certain predefined threshold.

The system also uses a windowing scheme called ILAS (incremental learning with alternating strata) [Bacardit et al., 2004] that reduces the run-time of the system and also introduces generalization pressure [Bacardit, 2004]. This mechanism divides the training set into several non-overlapped subsets and chooses a different subset at each GA iteration for the fitness computations of the individuals.

For this paper we have used GABIL's [DeJong and Spears, 1991] rule-based knowledge representation for nominal attributes. As the representation is important for the design of the rule-wise local search operators, a brief description follows: In *GABIL* each rule consists of a condition part and a classification part: $condition \rightarrow classification$. Each condition is a Conjunctive Normal Form (CNF) predicate defined as:

$$(A_1 = (V_1^1 \vee \ldots \vee V_1^m) \bigwedge \ldots \bigwedge A_n = (V_n^1 \vee \ldots \vee V_n^m))$$

Where A_i is the ith attribute of the problem and V_i^j is the jth value of the ith attribute.

This kind of predicate can be encoded into a binary string in the following way: given a problem with two attributes, where each attribute can take three values {1,2,3}, a rule of the form "If the first attribute has value 1 or 2 and the second one has value 3 then we predict class 1" will be represented by the string 110|001||1. There is a bit associated to each value of each attribute. If a value appears in the disjunction associated to its attribute in the predicate, the bit is set to one. Otherwise it is set to zero.

To initialize each rule, the system chooses a training example and creates a rule that classifies correctly this example, using the mechanism proposed in [Bacardit, 2005]. The covering process first activates the literals that match the chosen training example by setting the bits associated to the instance values to one. Next, it randomly initializes the rest of literals in the rule given a certain probability of having value one.

The individual representation is extended with an explicit default rule mechanism.

The evolved rules can predict all but one of the classes in the domain, an static default rule at the end of the rule set will predict the other class. The apparition of a default rule at the end of the rule set is an usual emergent phenomenon when using decision lists as individual representation (as we do). We explicitly exploit this phenomenon with an static default rule as it helps generating more compact and accurate solutions [Bacardit, 2004]. Finally, we use a parallel implementation of GAssist, applying a standard master-slave paradigm [Cantu-Paz, 2000].

4 Rule-wise and rule set-wise local search mechanisms

This section introduces the Memetic Pittsburgh Learning Classifier System (MPLCS) in its two variants, using either rule set-wise (MPLCS-RS) or rule-wise (MPLCS-R) LS mechanisms. For each kind of mechanism we specify algorithmic descriptions of the operators and how they are integrated inside GAssist. We have used two different policies of integration, either applying the operators to the whole population (controlled by a certain probability) or applying them only to the best individual of the population (in an elitist fashion). The former policy will be identified by appending a suffix *(P)* to the name of the method. The latter will use a suffix *(E)*. For instance, the MPLCS that uses rule-wise operators applied to the whole population will be known as *MPLCS-R(P)*. If we use both kinds of operators at the same time, the method will use the name *RS+R* so, as an example, the combination of rule-wise and rule set-wise operators applied to the best individual of the population will be known as *MPLCS-RS+R(E)*.

It is important to remark that the construction of Competent Memetic Algorithms (including MPLCS) depends on addressing important design issues. In this paper we have followed some of the recommendations and hybridization strategies described in [Krasnogor and Smith, 2005] in what pertains to where hybridization should take place and the appropriate balance of global and local search.

4.1 Rule set-wise Local Search mechanisms - MPLCS-RS

This subsection is divided into two parts. The first one briefly describes the rule set-wise LS operator itself while the second one details how it has been integrated into GAssist to produce the new algorithm MPLCS-RS. An extensive description, use examples and behaviour analysis for the operator when applied in the crossover stage of GAssist can be found in [Bacardit and Krasnogor, 2006], where it was called Smart Crossover (SX). In this work we extend the operator beyond the scope of the crossover stage of the GA. Thus, we call the operator *rule set-wise (RSW) local search* in this work.

4.1.1 The RSW operator

RSW has three main stages (Figure 1):

1. Evaluation of the candidate rules
2. Selection of the rules that will form the offspring rule-set
3. Generation of the final individual

In the first stage we evaluate all rules with all the examples of the training set. This process has as a result a map of correct and incorrect classifications of each rule. The next stage will use this map to evaluate how well each candidate rule can contribute to improve the accuracy of the offspring rule set without re-evaluating the rule set.

The second stage works as follows: We start the process with an empty offspring rule set. Next, for each candidate rule we evaluate which is the position inside the rule

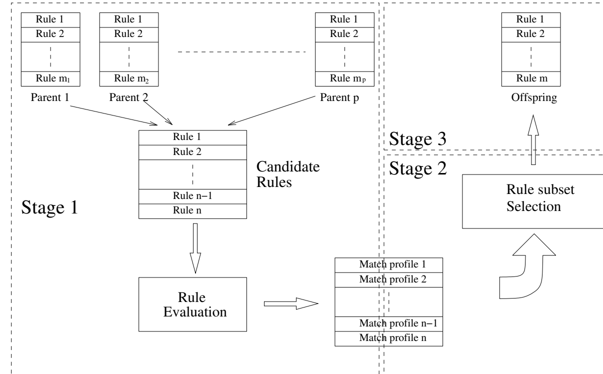

Figure 1: Representation of RSW

set where this rule, in case it was inserted, would contribute more to increase the rule set accuracy. A rule will only be inserted into the rule set if it manages to improve the rule set accuracy, and in the position determined previously to be the best one. When all candidate rules have been evaluated, the rule set is pruned of rules that do not classify correctly any example anymore (because they have been subsumed) and also of rules that cover too few additional positive examples.

As the order in which the rules are inserted into the rule set is important, the process explained above of generating the offspring rule set is repeated several times, reshuffling the candidate rules after each repetition. We pick the rule set that obtains highest accuracy and, in case of a tie in accuracy, smallest number of rules. Finally, an offspring is generated from the selected set of rules.

4.1.2 MPLCS-RS: integration of RSW into the GAssist framework

The operator described in the previous section only recombines *already existing* rules, and it does not generate new ones. We have investigated several ways of integrating RSW with the crossover and the population management strategies. The operator can be used in two different stages of GAssist, the crossover stage and the elitist stage. In this paper we study two different policies[1]:

- Integration of RSW into the crossover stage of GAssist: MPLCS-RS(P)
- Integration of RSW into the elitism stage of GAssist: MPLCS-RS(E)

MPLCS-RS(P) The crossover stage of GAssist will use both RSW and traditional crossover. The combination of both operators occurs through a control variable (P_{RSW}) that randomly selects which one to use. To prevent the operator from selecting the same parent twice we sample without replacement.

The pseudo-code in Figure 2 describes this integration assuming, for simplicity, that all individuals have the same default class, which is not always the case. When the population contains individuals with different default classes, the crossover stage is applied separately for each of the classes, as described in [Bacardit, 2004].

MPLCS-RS(E) In order to preserve the best individual found so far, an EA copies the best individual of the parents population directly into the offspring population, usually replacing the worst offspring. This procedure is known as elitism [Bäck et al., 1997].

[1] In previous work [Bacardit and Krasnogor, 2006] we only tested the first of the enumerated policies

```
Procedure Crossover algorithm with local search
Input : Parents,PopSize
DefaultClass = Get default class from Parents
Offsprings = ∅
TempParent = null
For i = 1 to PopSize
    If random number[0, 1] < P_cross
        If random number[0, 1] < P_MPLCS-RS
            SelectedParents = ∅
            For j = 1 to numParentsMPLCS − RS(P)
                Parent = Sample without replacement from Parents
                Add Parent to SelectedParents
            EndFor
            NewOffSpring = RSW(SelectedParents,DefaultClass)
            Add NewOffSpring to Offsprings
        Else
            If TempParent = null
                TempParent = Sample without replacement from Parents
            Else
                Parent2 = Sample without replacement from Parents
                NewOffSpring = OnePointCrossover(TempParent,Parent2)
                Add NewOffSpring to Offsprings
                TempParent = null
            EndIf
        EndIf
    Else
        Parent = Sample without replacement from Parents
        Clone = Copy of Parent
        Add Clone to Offsprings
    EndIf
EndFor
Output : OffSprings
```

Figure 2: Integration of RSW in the crossover stage of GAssist

The second way of integrating RSW into GAssist is through the elitism phase. The procedure works as follows:

1. Given the offspring population after crossover and mutation
2. Evaluate individuals
3. Select the N best individuals of the population
4. Apply RSW to these N elite individuals
5. Replace the worst individual of the offspring population with the newly created individual

4.2 Rule-wise local search mechanisms - MPLCS-R

This subsection describes the three studied rule-wise local search operators and their integration into GAssist to form MPLCS-R.

4.2.1 Definition of the operators

The studied rule-wise operators are tailored to the GABIL knowledge representation used by GAssist that provides very detailed information about the *behavior* of a rule. Please note that any algorithm using a GABIL-like representation, independently of the problem to which it is to be applied, could benefit from the proposed operators. This information is used to direct the way in which the local search operators function. Three operators are studied:

- Rule cleaning (RC)

- Rule splitting (RS)
- Rule generalizing (RG)

The two first operators edit the rules to make them more specific, while the third operator edits the rules to make them more general. Thus, we seek to provide MPLCS with complementary pressures, namely, specificity and generality.

The Rule Cleaning (RC) operator The rule cleaning operator is applied to a rule based on the set of examples that are matched by this rule, and heuristically disables the literal in the CNF predicate that makes the rule cover the most number of misclassified examples and does not classify correctly any example at all. The motivation and rationale of the operator is best illustrated with an example:

- Given the following rule:

Position	0	1	2	3	0	1	Class
Rule	1	1	1	0	1	1	1

- Given the following set of instances that are matched by this rule
  1. $0, 0||1$
  2. $1, 0||1$
  3. $2, 0||0$
  4. $0, 1||1$
  5. $1, 1||1$
  6. $2, 1||0$
- The rule classifies correctly four examples and incorrectly two of them (3 and 6).
- If we disable the literal associated to the third value (2) of the first attribute, neither of these two negative examples are covered by the rule.

To efficiently identify the literal to be disabled (cleaned), two counters are associated to each literal, one for the positive examples and one for the negative ones. After each example is matched, we increase the corresponding counter for the literal of each attribute that is activated by the example. After all examples are matched, we only need to look for the literal that has its positive examples counter set to 0 and maximum value for its negative examples counter. This literal is then set to 0. Figure 3 contains the code of the operator.

The Rule Splitting (RS) operator The previous operator identifies literals that only cover negative examples. However, this situation may not be very frequent, as usually literals that cover both positive and negative examples are encountered. Thus the rule cleaning operator is expected to be effective only rarely. This is compounded with the fact that GAssist promotes generalized individuals, that is, individuals that have as many enabled literals as possible due to the pressure introduced by the MDL-based fitness. Hence, the new operator shall handle very general rules in a smoother way than the rule cleaning one. The rule splitting operator achieves this aim by splitting the target rule into two, selecting the point that allows the rule cleaning operator to be applied to one of the generated sub-rules. The following example illustrates how the rule splitting local search operates:

- Given the following rule:

Position	0	1	2	3	0	1	0	1	Class
Rule	1	1	1	0	1	1	1	1	1

```
Procedure Rule Cleaning operator
Input : rule,MatchedExamples
PosCount = Initialize Counters for all literals of Rule
NegCount = Initialize Counters for all literals of Rule
Foreach example in MatchedExamples
          If rule.class = example.class
             Foreach att in NumAttributes
                    PosCount[att][example.values[att]] + +
             EndForEach
          Else
             Foreach att in NumAttributes
                    NegCount[att][example.values[att]] + +
             EndForEach
          EndIf
EndForEach

MaxNeg = 0
Foreach att in NumAttributes
          Foreach literal lit of att
                 If PosCount[att][lit] = 0 and NegCount[att][lit] > MaxNeg
                    MaxNeg = NegCount[att][lit]
                    TargetAtt = att
                    TargetLit = lit
                 EndIf
          EndForEach
EndForEach
If MaxNeg > 0
  Disable literal TargetLit of attribute TargetAtt of rule
EndIf
Output : rule
```

Figure 3: Pseudo-code of the RC operator

- Given the following set of instances that are matched by this rule
  1. $0, 0, 0||1$
  2. $1, 0, 1||1$
  3. $2, 0, 1||1$
  4. $2, 1, 1||1$
  5. $0, 1, 0||1$
  6. $1, 1, 1||1$
  7. $2, 0, 0||0$
  8. $2, 1, 0||0$
- The rule classifies correctly six examples and incorrectly two of them (7,8)
- We compute the positive and negative counters of each literal of the rule like in the rule cleaning operator:
  - Positive counters: $2, 2, 2, 0|3, 3|2, 4$
  - Negative counters: $0, 0, 2, 0|1, 1|2, 0$
- The counters show that there is no literal that can be cleaned, as the four of them with negative counts (3rd literal of att. 1, both literals of att. 2 and 1st literal of att. 3) also have positive counts.
- If we split the rule into two assigning to each of them one of the two (active) values of the third attribute, thus creating 1110|11|10||1 and 1110|11|01||1, the matched examples are distributed as following:

- First rule
    1. $0, 0, 0||1$
    2. $0, 1, 0||1$
    3. $2, 0, 0||0$
    4. $2, 1, 0||0$
- Second rule
    1. $1, 0, 1||1$
    2. $2, 0, 1||1$
    3. $2, 1, 1||1$
    4. $1, 1, 1||1$

- Then we recompute the positive and negative counters for both of them:
    - Positive counters of first rule: $2, 0, 0, 0|1, 1|2, 0$
    - Negative counters of first rule: $0, 0, 2, 0|1, 1|2, 0$
    - Positive counters of second rule: $0, 2, 2, 0|2, 2|0, 4$
    - Negative counters of second rule: $0, 0, 0, 0|0, 0|0, 0$
- The third literal of the first attribute for the first splitted rule only covers negative examples and therefore can be disabled
- We disable the identified bad literal and exchange the original rule for the two splitted rules

The split procedure as applied to a rule, selects one attribute of the domain and generates two sub-rules that are identical except for the selected attribute. One sub-rule has only one active literal: one of the active literals of the original rule. The other sub-rule has this literal disabled but all the other active literals of the original rule are enabled. This means that in order to apply the split procedure to a certain attribute of the domain, the original rule must have at least two literals that are enabled and cover both positive and negative examples. The rule splitting operator tries to split all such attributes of the original rule and then tries to apply the rule cleaning operator to the sub-rules. Only one split and clean is actually applied: the one that manages to remove most negative examples. Figure 4 contains the pseudocode of the operator.

The Rule Generalizing (RG) operator The two previous operators share a common characteristic, they try to correct the mistakes done by the rules they manipulate by disabling some literals. The new operator applies the opposite concept: trying to activate literals to the rule in order to cover more positive examples. Our operator identifies the single disabled literal in the rule that, if enabled, would cover as many new positive examples as possible but without covering any new negative example. In order to achieve this objective the operator identifies all non-covered examples that only failed to be matched by a single attribute. From this list of candidate positions, it selects and enables the one that can add more positive examples without adding any new negative example. Figure 5 contains the pseudo-code of the operator.

```
Procedure Rule Splitting operator
Input : rule,MatchedExamples
(PosCount, NegCount) = Compute activation counter for rule on MatchedExamples
MaxNeg = 0
Foreach att in NumAttributes
        Literals = Identify literals lit from att that have
                PosCount[att][lit] > 0 and NegCount[att][lit] > 0
        If |Literals| ≥ 2
          Foreach lit in Literals
                ExamplesLit = Examples from MatchedExamples covered by lit
                Compute positive and negative counters for ExamplesLit
                NumNeg = Check from ExamplesLit if some literal can be cleaned
                If NumNeg > MaxNeg
                  MaxNeg = NumNeg
                  SplitAtt, SplitLit = att, lit
                EndIf

                ExamplesOther = Examples from MatchedExamples covered by the order
                              literals of att
                Compute positive and negative counters for ExamplesOther
                NumNeg = Check from ExamplesOther if some literal can be cleaned
                If NumNeg > MaxNeg
                  MaxNeg = NumNeg
                  SplitAtt, SplitLit = att, lit
                EndIf
          EndForEach
        EndIf
EndForEach

If MaxNeg > 0
  rule1, rule2 = Split rule by attribute SplitAtt and literal SplitLit
  Disable the appropriate literal of either rule1 or rule2
Output : rule1, rule2
Else
Output : rule
EndIf
```

Figure 4: Pseudo-code of the RS operator

4.2.2 MPLCS-R: integration of the rule-wise local search operators into the framework of GAssist

We evaluate three different ways in which we can use these three rule-wise local search operators inside GAssist. The three policies are:

- Applying the operators to the whole population. A new stage is added to the GA cycle after mutation, where the rule-wise LS operators are applied to each individual of the population with certain probability, as represented in the pseudo-code in Figure 6. This option is named MPLCS-R(P).
- Applying the LS operators only to the best individual of the population at the end of each iteration. This option is named MPLCS-R(E).
- Applying the LS operators inside RSW. The operators are applied after trying to insert all candidate rules, and before pruning the selected rule subset. This option is named MPLCS-RS+R (with its two (P) and (E) variants).

The selected rule-wise operators in each of these policies are specified by appending a ':OP' suffix to the policy name. For instance, the MPLCS variant using Elitist Rule Generalizing is defined as MPLCS-R(E):RG, while the probabilistic policy using both Rule Cleaning and Rule Splitting is defined as MPLCS-R(P):RC+RS.

When an individual is selected for the application of the LS operators (in any of the above policies), the LS operators are applied to all of its rules, starting with the

```
Procedure Rule Generalizing operator
Input : rule,NotMatchedExamples
(PosCount, NegCount) = Initialize counters to 0 for all literals of rule
Foreach instance in NotMatchedExamples
        numAttMissed = 0
        Foreach att in NumAttributes
                If literal rule[att][instance.values[att]] is disabled
                  numAttMissed ++
                EndIf
        EndForEach

        If numAttMissed = 1
          lit = Literal missed from rule for example
          If example.class = rule.class
            PosCount[lit] ++
          Else
            NegCount[lit] ++
          EndIf
        EndIf
EndForEach

MaxPos = 0
Foreach Litera lit in rule
        If PosCount[lit] > MaxPos and NegCount[lit] = 0
          MaxPos = PosCount[lit]
          TargetLit = lit
        EndIf
EndForEach

If MaxPos > 0
  Enable literal TargetLit of rule
EndIf

Output : rule
```

Figure 5: Pseudo-code of the RG operator

```
Procedure GA Cycle
Population = Initialize population
Evaluate(Population)
For it = 1 to NumIterations
Selection(Population)
Offspring=CrossOver(Population)
Mutation(Offspring)
LocalSearch(Offspring)
Population=Replacement(Population,Offspring)
Evaluate(Population)
EndFor
Output : Best individual from Population


Procedure LocalSearch
Input : Population
ForEach individual in Population
          If rand(0,1) < probLocalSearch
            Apply Rule-wise Local search operators to individual
          EndIf
EndForEach

Output : Population
```

Figure 6: Integration of MPLCS-R(P) in the GA cycle

first rule and continuing to the last. As the rule sets are ordered decision lists, the order in which the operators are applied is crucial, as the modification of a certain rule will probably affect the rules placed after it in the list. All the selected LS operators are applied always to each rule before continuing to the next one.

The order of application of the operators to a rule is also important. First of all, the rule splitting operator is designed to be applied to the rules that the rule cleaning operator is unable to fix. Therefore it is applied after the cleaning operator. Moreover, the rule generalizing operator will always be applied after the other two operators. The rationale in this case is to compensate the specificity pressure introduced by the first two operators with the generality pressure introduced by the later. [Krasnogor and Smith, 2005] suggested a variety of ways in which one can "schedule" the application of LS within an EA framework; we leave the exploration of other alternatives for future work.

Finally, in some of our initial tests (not reported) we have observed an explosion in the number of rules of the individuals, due to a 'recursive' application of the rule splitting operator. This issue is specially problematic in early iterations of the learning process, when the operators deal with close-to-random rules. In order to avoid this effect, the rule split operator is not applied to any newly created rule that is the result of the previous split. Figure 7 contains the code of the application of the LS operators to an individual, containing all the above considerations.

```
Procedure ApplyRuleWiseLocalSearch
Inputs : individual
lastSplitted = 0
index = 1
Instances = TrainingSet
While index < individual.numRules
        MatchedExamples = Examples from Instances matched by individual.rules[index]
        If RuleCleaning(individual.rules[index],MatchedExamples) disables any literal
            MatchedExamples = Examples from Instances matched by individual.rules[index]
        EndIf

        If lastSplitted = 1
            lastSplitted = 0
        Else
            rule1, rule2=RuleSplitting(individual.rules[index],MatchedExamples)
            If RuleSplitting generated rule1, rule2
                Remove individual.rules[index]
                Insert rule1, rule2 in individual.rules at position index
                MatchedExamples = Examples from Instances matched by individual.rules[index]
                lastSplitted = 1
            EndIf
        EndIf

        Remove MatchedExamples from Instances
        NewMatched = RuleGeneralizing(individual.rules[index],Instances)
        If |NewMatched| > 0
            Remove NewMatched from Instances
        EndIf
        index ++
EndWhile
Output : individual
```

Figure 7: Application of the rule-wise local search operators to an individual

5 Experimental design

This section contains the description of the experimental protocol that has been followed to evaluate the MPLCS methods presented in the paper. Two stages of experiments have been carried out. The first of them contains a large scale evaluation of 75 configurations of MPLCS based on different combinations of LS operators and parameter values, using a single dataset (the 20 bit multiplexer). In this stage our objective is to evaluate when the LS operators/policies are able to contribute towards a better learning process or they just add fruitless computational effort.

In the second stage, the most promising MPLCS combinations arising from the

previous stage are tested on several other datasets to evaluate whether the performance and behavior observations extracted in the first stage still hold. This second stage tests the best MPLCS combinations in terms or robustness to noise, scalability, and capacity of adaptation to other kind of datasets. The rest of the section explains these two stages of experiments and the performance measures used to evaluate them.

5.1 First stage of experiments

5.1.1 MPLCS experimental suits

In the previous section we have defined various kinds of operators and policies of application. For simplicity we will only evaluate a subset of them, organized in six experimental suits. In total 75 MPLCS algorithms have been tested. Due to space limitations, we report here the most salient observations of these experiments. For full details and to aid independent reproduction of our results, please refer to the supplementary material at `http://www.infobiotic.net/papers/MPLCS-first-stage.pdf`.

In all suits where we use the three rule-wise local search operators, we have tested seven combinations of them. First, each LS operator is tested separately, then combinations of two operators are tested together and them finally the three operators in unison. What we expect to see in the results is that the best results are obtained when all the operators are used together, as it introduces a balance between two kinds of pressures, specificity pressure (from the rule cleaning and rule splitting) and generality pressure (from the rule generalizing). This balance of pressures has been studied in depth for Michigan LCSs [Butz et al., 2004].

In our previous work [Bacardit and Krasnogor, 2006] we conducted an extensive sensitivity analysis of the RSW operator when applied probabilistically (affecting MPLCS-RS(P) and MPLCS-RS+R(P)). In this paper we are going to use only the best set of parameters identified in those experiments: Probability of RSW: 0.1; number of parents: 10 ; repetitions of rule subset selection: 5. The last parameter is also used for MPLCS-RS(E) and MPLCS-RS+R(E).

The six experimental suits are:

1. **Basic** suite. As a baseline for comparison, GAssist will be run without any of the local search operators

2. **MPLCS-R(P)** suite. We tested the rule-wise local search operators applied with probabilities ranging from 5% to 25% with increments of 5%. Total: 35 experiments

3. **MPLCS-R(E)** suite. We tested the rule-wise local search operators applied to the best individual of the population. Total: 7 experiments

4. **MPLCS-RS** suite. We tested the rule set-wise operator within the crossover stage - MPLCS-RS(P) - and in elistist way - MPLCS-RS(E) - with 5, 10 or 15 parents. Total: 4 experiments

5. **MPLCS-RS+R(E)** suite. We tested the integration of the three rule-wise local search operators with RSW used in the elitist stage, again using 5, 10 or 15 parents. Total: 21 experiments

6. **MPLCS-RS+R(P)** suite. We tested the integration of the three rule-wise local search operators with RSW used in the crossover stage. Total: 7 experiments

5.1.2 Dataset

For this first stage of experiments we used a single dataset, namely the 20 bit multiplexer problem as it provides us with enough data to illustrate and evaluate the strong and weak points of each algorithm. We selected the 20 bit multiplexer instead of the 11-bit one, which was used in previous experiments with RSW [Bacardit and Krasnogor, 2006], because our aim is also to evaluate the scalability of these methods and this dataset, with over a million examples, is already a fairly large dataset. Also, until recently [Llorà et al., 2006], there were no results reported in the literature about any Pittsburgh style LCS solving this dataset.

5.1.3 GAssist configuration

The configuration used for GAssist/MPLCS is detailed in table 1. For the explanation of all the parameters and operators, please see [Bacardit, 2004]. This is the default configuration of the system, with only one exception: the number of strata of the ILAS windowing scheme. ILAS was designed to alleviate the computational time of GAssist for large datasets by using only a subset of training examples for its fitness computations. Therefore, it would be appropriate to use a large number of strata for all the datasets used in this paper. However, the local search operators studied in this paper explicitly tailor the rules/rule sets to improve their performance for the current training examples. If these examples change after each iteration, the changes introduced by the LS operators for a given iteration may be inadequate for the instances used in the next one. Therefore, in this stage we prefer to avoid external interactions that could affect the LS operators. Hence we use only two strata, thus expecting small impact from ILAS on the LS while the run-time is halved. We note that most of the local search operators studied in this paper have a computational cost that is related to the number of examples used for their operation. However, although the above ILAS settings contribute to a slowing down of MPLCS, we think that it is worth to evaluate MPLCS in isolation, without (most of) the external interaction of ILAS, and this stage it is a good opportunity of doing so because the dataset we use is still relatively small. Also, as the obtained results will show, the relative performance differences between GAssist and the evaluated MPLCS variants remain similar with/without ILAS. Thus, the findings from the first stage can be extrapolated to the experiments in the second stage. In the next stage, where we evaluate datasets of larger size and difficulty, we will evaluate MPLCS exclusively in combination with ILAS as this is the only tractable choice.

5.2 Second stage of experiments

In the second stage of experiments we evaluate the most promising MPLCS configurations identified in the previous experiments and we test them in new conditions. First of all, these configurations are evaluated in combination with a high number of strata of the ILAS windowing scheme to determine whether the combination of windowing and local search is beneficial for the system or not. Two hundred strata for the ILAS windowing stage are used on these new experiments. All other parameters of GAssist remain unaltered using the values in table 1. Moreover, we test MPLCS on other datasets to check if the observations from the previous experiments still hold. We have selected the following datasets:

Noisy 20 bit multiplexer The aim of this dataset is to evaluate how robust is MPLCS to noise. Thus we use tunable noisy version of the dataset used previously. The noise is introduced in the class label, flipping it with a certain probability. Five different levels of noise have been evaluated: 5%, 10%, 15%, 20% and 25%. For

Table 1: General parameters of GAssist/MPLCS

Parameter	Value
General parameters	
Crossover prob.	0.6
Selection algorithm	tournament selection
Tournament size	3
Population size	300
Individual-wise mutation prob.	0.6
Initial number of rules per individual	20
Default class policy	major
Iterations	until convergence
Number of slaves for master-slave parallel model	10
Rule deletion operator	
Iteration of activation	5
Minimum number of rules	6
MDL-based fitness function	
Iteration of activation	25
Initial theory length ratio	0.075
Weight relax factor	0.90
MDL-based fitness function	

each level of noise, ten versions of the noisy dataset have been generated, and the results of learning them are averaged. Again, 200 strata are used for this dataset.

k-DNF datasets The next dataset we use for these experiments is a dataset extensively used in the field of Computational Learning Theory [Kearns and Vazirani, 1994], specially for PAC-learning [Valiant, 1984]. It consists in learning a boolean function defined as a disjunctive normal form predicate with n disjunctive terms (that is, the rules we have to learn), where each term has at most k expressed attributes. Specifically, we have used the same version of the kDNF dataset used by Butz et al. [Butz and Pelikan, 2006] to evaluate XCSBOA, where $k = 5$, $n = 22$ and the total number of attributes in the dataset is 20. Therefore, the dataset has the same number of instances, and slightly higher number of rules (23 vs. 17) as the 20 bit multiplexer (we will use the same number of strata for the ILAS windowing scheme). The reason for using this dataset is to show that the contribution of MPLCS is general, and it performs well in many domains, not only in the multiplexer datasets.

37 and 70 bit multiplexer Finally, the last two datasets used for these experiments are the next two versions of the multiplexer problem of larger size after the 20 bit one. Learning these two datasets means some changes in GAssist: Until now, each dataset was fully loaded from hard disk and stored in RAM memory. However, it is impossible to do the same for the full set of instances of these datasets. We will generate the dataset on-the-fly in a similar way as it was done for XCS [Butz, 2003], where at each learning step a new instance is randomly created with uniform probability for each of its input bits, and then its class label is computed by evaluating the boolean function that defines the multiplexer problem. Our aim in using these two datasets is to evaluate the scalability of MPLCS beyond domains of a million examples, like the ones used before.

In the case of GAssist, before each iteration $N/numStrata$ instances are sampled (with replacement) using a similar approach. These instances are used for all the fitness computations and local search procedures performed during the iteration,

and discarded at its end. This means that the windowing scheme used will not be ILAS (where the whole training set is stratified in non-overlapped strata before the learning process) anymore, but just a random subset sampling of the training set. However, the *numStrata* parameter will maintain the same meaning, determining the number of instances that will be used at each GA iteration. We have tuned this parameter for both datasets to obtain the lowest possible number of examples per iteration that still lets GAssist learn the problem. This means 1373 instances per iteration in the 37 bits multiplexer and 1574 instances per iteration in the 70 bit multiplexer.

5.3 Performance measure and evaluation process

When Michigan LCS are evaluated in the kind of datasets that we employ in this paper their performance is assessed mainly by counting the number of learning steps that are required to achieve the optimal accuracy for the used dataset (e.g. [Wilson, 1995, Butz et al., 2006]). Other performance metrics are the number of classifiers in the population and examining the population to check if the optimal rules for the problem at hand were generated and identified as high payoff classifiers. That is, Michigan LCS are evaluated checking how many "atomic steps" of its learning cycle are necessary to reach the optimal solution. For a Pittsburgh LCS, the atomic step is a GA iteration. If no windowing scheme is used, each iteration uses as many learning steps as the number of training instances. Otherwise, each iteration uses as many learning steps as the size of the window. Thus this seems an objective way of comparing the performance of these systems.

However, is this metric truly fair to compare between the LS operators studied in this paper? The answer is no, as the amount of search effort in an iteration varies among operators. Counting the number of match operations that each configuration performs is more fair but not entirely satisfactory either, as some of these operators perform only partial matching. Our selected performance measure to compare between the MPLCS variants is, simply, the average run-time of GAssist until perfect accuracy is achieved. Although this measure has some drawbacks, especially for replicability purposes across different computing platforms, it is the most objective measure to asses the performance of the LS operators and policies of application, as these have all run in a uniform hardware.

As all the different MPLCS share the same code base, the run-time differences between them can only be a consequence of a more efficient exploration of the search space and not due to implementation/compilation issues. We would like to remark that with these experiments our aim is not to optimize the run-time of the various MPLCS variants, but to check how quick do the different Memetic LCSs converge towards an optimal solution. Also, to allow an easier replicability of the results reported in the paper, we have decided to place online a copy of GAssist and MPLCS source code at `http://www.infobiotic.net/software/GAssist+MPLCS.tar.gz`.

Moreover, the results of the experiments are analyzed using one-way ANOVA tests combined with the post-hoc Tukey HSD test for multiple comparisons using a 99% confidence level[2], as suggested by [Lanzi et al., 2006]. The aim of the tests is to identify, among the compared algorithms, the groups of methods that obtain statistically similar performance and a hierarchy of performance groups, which groups are significantly better than other groups, etc. For example, when we write (G1, G3) $\rightarrow$ G3 $\rightarrow$ G4, it means that the group made up of G1 and G3 methods has better performance than G3, which

[2]Using the implementation of both tests from the R project `http://www.r-project.org`.

in turn has better performance than G4.

Although our evaluation is based on run-time, we also report other measures such as GA iterations until convergence and size of the obtained rule sets, as they help to illustrate the behavior of these methods. The rule-set sizes we report are "worst case" scenarios, as the reported measure is the rule-set size of the first rule set that managed to achieve 100% accuracy. If MPLCS was left running for more iterations, in most if not all of the configurations it would manage to obtain the optimal rule set, which for the 20 bit multiplexer consists of 17 rules (16 rules covering one class plus the default rule for the other class). For the 37 bit multiplexer the optimal rule set consists of 32+1 rules, and 64+1 for the 70 bit multiplexer. The optimal kDNF rule set consists of at most $22 + 1$ rules. In some cases these rules overlap and we can obtain rule sets of less than 23 rules. However, reporting this measure as it is will help us illustrate the behavior of the studied operators as we will show which methods build a well generalized solution while they are learning, or if they only concentrate on achieving perfect accuracy and leave the search for a well generalized solution for later.

6 Results

We report next the results obtained in the two stages of experiments and compare MPLCS against state-of-the-art LCS.

6.1 First stage of experiments

In this subsection we show a summary of the results of testing 75 combinations of MPLCS operators and parameters. The full detail of these results is reported in the supplementary material. Each result is the average of 10 runs using different random seeds. All experiments were performed on the Jupiter supercomputer of the University of Nottingham, using Opteron 248 processors running at 2.2GHz and the Linux operating system. We used a master-slave parallel version of GAssist with 10 slaves.

6.1.1 Category MPLCS-R(P)

Table 2 contains the results of this category of experiments, where we test the three rule-wise local search operators (alone and combined) applied to the whole population given a probability. It is important to remark that GAssist without any of the evaluated operators is already able to solve the 20 bit multiplexer problem. To the best of our knowledge, this is the first time that this problem was solved with a standard Pittsburgh LCS, without any Local Search/Structural Learning add-on. Moreover, these results illustrate the need to use the run-time as evaluation criterion for the experiments: table 2 shows how configurations using the Rule Cleaning and Rule Splitting operators take less iterations to converge compared to the standard GAssist but can be more than two times slower. We can see a general trend about the probability of application of the operator. Higher probability does not help improving the performance of the system, it just slows it down. With the exception of the RC+RS configuration, for the other 6 combinations of rule-wise operators the best configuration was the one with smallest probability, 0.05. For RC+RS the best configuration used a probability of 0.10.

For simplicity, we only run statistical tests comparing the configurations with best probability across the 7 combinations of rule-wise LS operators and the basic GAssist. The statistical tests identified the following performance groups: MPLCS-R(P):RC+RS+RG $\rightarrow$ (MPLCS-R(P):RC+RS,MPLCS-R(P):RS+RG) $\rightarrow$ (MPLCS-R(P):RC+RG,MPLCS-R(P):RC) $\rightarrow$ Basic $\rightarrow$ (MPLCS-R(P):RS,MPLCS-R(P):RG). The best configuration (RC+RS+RG), however, does not manage to generate com-

Table 2: Results of the MPLCS-R(P) category of the experiments. Top configuration is marked in bold

Conf.	LS Prob.	Iter.	#Rules	Run-time(s)
Basic		1073.70±99.11	23.1±2.9	7950.3±792.1
MPLCS-R(P):RC	0.05	609.20±70.38	21.4±2.1	5912.9±752.9
	0.10	553.30±67.89	21.4±2.1	6278.0±842.7
	0.15	473.10±72.45	20.0±1.2	5971.7±1099.3
	0.20	467.90±67.30	19.5±1.3	6729.7±1000.1
	0.25	475.60±61.27	21.4±1.7	7921.6±1328.9
MPLCS-R(P):RS	0.05	742.90±83.52	20.8±1.5	12064.7±1427.6
	0.10	566.80±77.61	21.0±1.5	12399.1±2229.6
	0.15	484.50±72.64	19.4±1.5	15203.8±2910.5
	0.20	434.90±106.38	21.2±1.8	16864.9±4511.4
	0.25	429.10±66.75	20.4±1.5	19949.2±3361.2
MPLCS-R(P):RG	0.05	837.50±137.21	19.2±1.6	10545.7±1932.6
	0.10	718.50±72.69	18.1±0.9	11352.1±1414.2
	0.15	680.60±81.77	17.9±1.0	13313.0±1909.9
	0.20	724.70±85.22	17.8±1.2	16622.9±2209.3
	0.25	690.40±60.45	17.9±0.9	18177.3±1607.3
MPLCS-R(P):RC+RS	0.05	211.90±70.81	18.6±0.8	3298.4±1073.2
	0.10	106.40±45.43	21.7±2.7	2597.1±828.6
	0.15	116.80±41.33	21.3±2.7	3554.9±794.8
	0.20	106.60±44.66	22.5±1.9	4447.3±1283.7
	0.25	88.50±38.08	22.6±3.7	4773.3±1537.4
MPLCS-R(P):RC+RG	0.05	375.90±45.88	**17.5±0.7**	4362.4±594.1
	0.10	309.10±35.30	**17.5±0.7**	4559.3±642.5
	0.15	279.00±79.07	17.3±0.6	4789.0±1448.6
	0.20	264.30±49.00	17.4±0.5	5645.3±1129.5
	0.25	232.50±52.88	**17.1±0.3**	5672.8±1471.8
MPLCS-R(P):RS+RG	0.05	131.90±46.06	17.8±1.0	2861.5±759.6
	0.10	113.80±24.56	18.4±1.0	4405.5±1012.9
	0.15	93.60±8.30	19.1±1.4	5398.2±662.3
	0.20	88.56±11.88	18.6±1.8	6374.2±1039.6
	0.25	96.10±17.38	19.2±1.5	9048.3±1683.8
MPLCS-R(P):RC+RS+RG	0.05	53.20±11.14	19.3±1.4	**1243.0±204.5**
	0.10	**32.40±6.89**	21.7±2.7	1255.7±239.0
	0.15	30.30±7.36	24.7±7.3	1581.2±355.8
	0.20	26.20±9.10	23.8±3.6	1704.1±491.7
	0.25	**21.80±6.57**	23.1±4.8	1658.7±375.8

pact rule sets at the same time it learns. The configuration RC+RG in general managed to obtain rule sets with a number of rules very close to the optimal 17 rules. Moreover, figure 8 plots training accuracy of the best individual of the population against run-time, showing the different learning speeds across methods, as well as visualizing clearly some of the groups of configurations identified by the statistical tests.

As a general observation of this category of experiments we can say that none of the individual local search operators on its own is able to achieve much better performance than the Basic GAssist. Only the RC operator is slightly better. A proper combination of operators, balancing specificity and generality pressure, manage to achieve the best performance. An extreme case of the lack of balance between these two pressures is the RS configuration. The reason it is much slower than the Basic configuration is illustrated in figure 9 where the evolution of the average rule-set size for the first 50 GA iterations is shown. The operator injects many new rules into the population, making the learning process much slower until the MDL fitness function is activated at iteration 25. RS needs

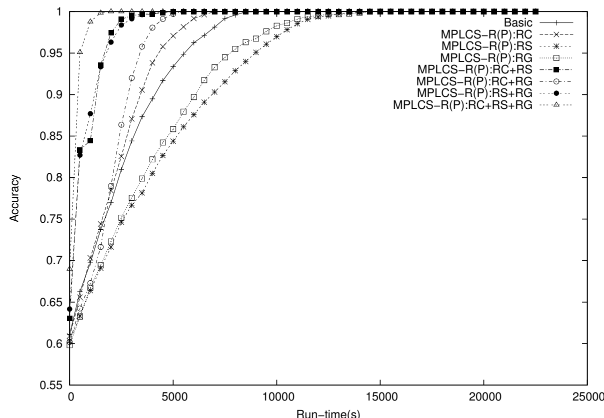

Figure 8: Time vs Accuracy for MPLCS-R(P) experiments

to be combined with some of the other operators to perform well. On the other hand, the three best configurations included Rule Splitting, thus, showing that the operator is necessary too.

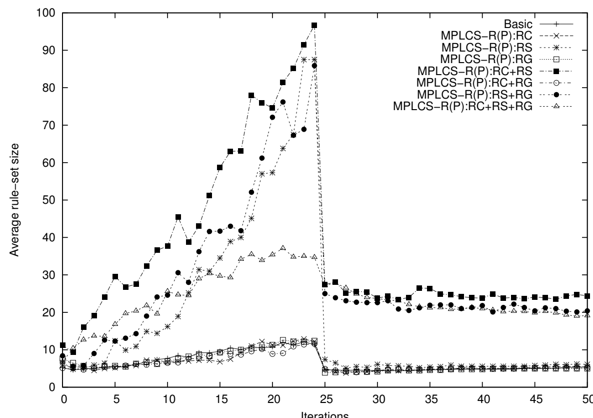

Figure 9: Iterations vs Average rule-set size for MPLCS-R(P) experiments

6.1.2 Category MPLCS-RS

Table 3 contains the results of this category of experiments. In this category we compare MPLCS-RS(P) as was used in our previous work [Bacardit and Krasnogor, 2006] against using MPLCS-RS(E) applied to the 5, 10 or 15 best individuals of the population. These results show very poor performance of all the variations of the RSW operator on its own. The statistical tests indicate the following performance hierarchy: Basic → MPLCS-RS(E)_5p → MPLCS-RS(E)_10p → (MPLCS-RS(E)_15p,MPLCS-RS(P)).

Figure 10 plots training accuracy of the best individual of the population against run-time. Some interesting features are discernible: in the early stages of the learning process, MPLCS-RS(P) is superior to all the other configurations (although only marginally better than Basic). However, after 250 seconds it does not contribute to a better learning and only slows down the system. It may be interesting to evaluate some

policy of application for the operator that decreases its use through time, which is left for further work. In a similar manner, adding more parents to MPLCS-RS(E) does not contribute to improve the system performance, and only increases its cost.

Table 3: Results of the MPLCS-RS category of the experiments. Best configuration is marked in bold

Conf.	Iter.	#Rules	Run-time(s)
Basic	1073.70±99.11	23.1±2.9	**7950.3±792.1**
MPLCS-RS(P)	**695.80±92.17**	**17.0±0.0**	44758.7±6775.6
MPLCS-RS(E)_5p	1032.40±188.87	21.8±7.4	19127.4±4842.1
MPLCS-RS(E)_10p	1035.30±176.22	19.2±4.4	31459.5±6912.6
MPLCS-RS(E)_15p	1132.40±121.38	17.0±0.0	49356.1±7659.9

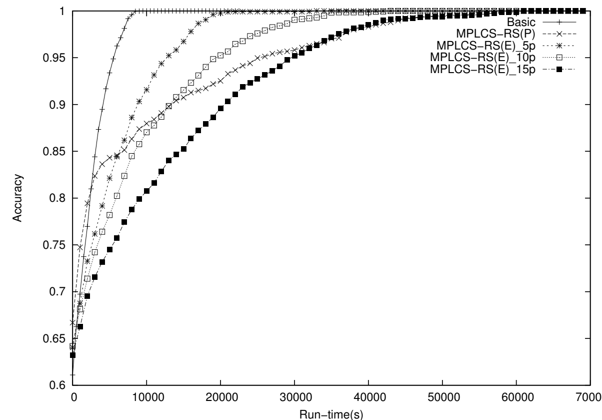

Figure 10: Time vs Accuracy for MPLCS-RS experiments

6.1.3 Category MPLCS-RS+R(P)

Table 4 contains the results of this category of experiments. In this category we are combining RSW, as was used in our previous work, with the rule-wise local search operators. The results of this category of experiments concentrate the best and the worst results of all the experiments observed so far. It contains the configuration with worst performance overall (MPLCS-RS+R(P):RS) and the configuration with best performance overall (MPLCS-RS+R(P):RC+RS+RG). Again we observe similar trends as in previous experiments. All combinations of rule-wise operators perform better than the separate operators, and its combination with RSW helps obtaining optimal rule sets. The statistical tests indicate the following performance groups: (MPLCS-RS+R(P):RS+RG, MPLCS-RS+R(P):RC+RS+RG) → MPLCS-RS+R(P):RC+RS → MPLCS-RS+R(P):RC+RG → Basic → MPLCS-RS+R(P):RC → MPLCS-RS+R(P):RG → MPLCS-RS+R(P):RS. We removed RC, RG and RS from the data fed to the statistical tests because they were distorting the results, as they perform up to an order of magnitude worse than the other configurations.

6.1.4 Comparison across experiment categories

The best configuration for each of the five categories of MPLCS methods in which we splitted the experiments of this stage are now compared. We have not reported the

Table 4: Results of the MPLCS-RS+R(P) category of the experiments. Best configuration is marked in bold

Conf.	Iter.	#Rules	Run-time(s)
Basic	1073.70±99.11	23.1±2.9	7950.3±792.1
MPLCS-RS+R(P):RC	323.30±81.83	**17.0±0.0**	20791.4±5080.3
MPLCS-RS+R(P):RS	648.40±413.90	23.3±12.4	65040.0±41849.0
MPLCS-RS+R(P):RG	385.70±70.00	**17.0±0.0**	32441.2±6324.4
MPLCS-RS+R(P):RC+RS	13.00±3.38	**17.0±0.0**	1861.2±295.2
MPLCS-RS+R(P):RC+RG	34.30±13.35	**17.0±0.0**	2873.1±1092.7
MPLCS-RS+R(P):RS+RG	6.70±1.10	**17.0±0.0**	1399.4±128.2
MPLCS-RS+R(P):RC+RS+RG	**4.40±0.49**	**17.0±0.0**	**1028.5±94.0**

results for two of these categories. For the MPLCS-R(E) stage, the best configuration was MPLCS-R(E):RC, while for the MPLCS-RS+R(E), the best configuration was MPLCS-RS+R(P):RC+RS+RG using 15 parents.

Table 5 shows the results of this comparison. The statistical tests indicate the following performance groups: (MPLCS-RS+R(P),MPLCS-R(P)) $\rightarrow$ (Basic,MPLCS-RS+R(E),MPLCS-R(E)) $\rightarrow$ MPLCS-RS. Except for MPLCS-RS(P), all other categories manage to obtain better performance than the Basic configuration. Also, only the algorithms of MPLCS-RS+R class managed to obtain the optimal rule-set for the 20 bit multiplexer. As we expected, the best algorithm is the one that has the right balance between specificity and generality (related to the rule-wise operators) and the proper balance between rule-wise and rule-set wise operators. Elitist algorithms in general performed worse than the probabilistic ones. Figure 11 plots training accuracy of the best individual of the population against run-time. Two configurations manage to obtain similar performance, MPLCS-RS+R(P) and MPLCS-R(P). However, only MPLCS-RS+R(P) managed to generate the optimal rule set, thus showing superior generalization capacity that may be necessary when applying these operators to real-world datasets. The next stage of experiments will focus on these two configurations to see whether the observations identified in this stage still hold or not.

Table 5: Results of comparing the best configurations of each category of experiments. Best configuration is marked in bold

Category	Iter.	#Rules	Run-time(s)
Basic	1073.70±99.11	23.1±2.9	7950.3±792.1
MPLCS-R(P)	53.20±11.14	19.3±1.4	1243.0±204.5
MPLCS-R(E)	862.50±102.45	23.5±3.9	7264.5±909.8
MPLCS-RS	1032.40±188.87	21.8±7.4	19127.4±4842.1
MPLCS-RS+R(E)	93.70±34.35	**17.0±0.0**	5201.5±1900.1
MPLCS-RS+R(P)	**4.40±0.49**	**17.0±0.0**	**1028.5±94.0**

6.2 Second stage of experiments

6.2.1 Combining the LS operators with the ILAS windowing scheme

We reran the two best MPLCS configurations identified (MPLCS-R(P) and MPLCS-RS+R(P)) plus the original GAssist using 200 strata for the windowing code instead of the two strata used before. Table 6 contains the results of these experiments. All configurations (even Basic) benefit greatly from the use of ILAS. The basic configuration converges 17.6 times faster compared to the previous settings of ILAS. Moreover,

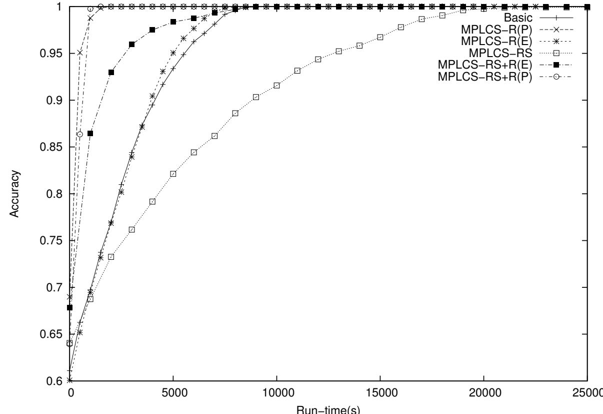

Figure 11: Time vs Accuracy for the global comparison of experimental categories

MPLCS-RS+R(P) is still the best method and the one that converges to the optimal solution, but now it is 25.8 times faster than the Basic configuration. Thus, interactions between the local search mechanisms and ILAS seems to be beneficial, as illustrated by these experiments. The statistical tests performed over the results from table 6 indicate the following performance groups: (MPLCS-RS+R(P),MPLCS-R(P) $\rightarrow$ Basic.

Table 6: Results of comparing the best configurations of each category of experiments using 200 strata for the ILAS windowing scheme. Best configuration is marked in bold

Category	Iter.	#Rules	Run-time(s)
Basic	1198.70±150.75	30.0±2.9	452.3±67.5
MPLCS-R(P)	23.70±4.65	24.8±5.2	30.4±4.4
MPLCS-RS+R(P)	**4.80±0.75**	**17.0±0.0**	**17.5±1.2**

6.2.2 kDNF dataset

Table 7 contains the results of the experiments on this dataset. We can see how both MPLCS configurations manage to obtain better performance than GAssist and they manage to generate very compact rule sets too, exploiting the fact that some of the terms in the DNF may be overlapped and the kind of rules that GABIL evolves have enough expressive power to adapt to this situation. Moreover, we can observe that the deviation of the run-time of MPLCS-RS+R(P) is actually higher than the average. This configuration obtained very poor results on some of the runs, because it was able to learn very quickly most of the rules in the optimal solution but struggled to learn the final ones. This dataset provides very few fitness guidance if a new rule has to be learned from scratch. Moreover, the heuristic behind the RSW operator always tries to generate the most compact rule set with the available rules. Thus, it cannot cope very well in this situation because it generates excessive exploitation power. Figure 12 illustrates this situation plotting the training accuracy of the best individual of the population against run-time. The statistical tests indicate that both MPLCS configurations have similar performance and outperform GAssist.

Table 7: Results for the k-DNF dataset. Best configuration is marked in bold

Category	Iter.	#Rules	Run-time(s)
Basic	2908.4±796.8	31.1±6.3	1164.2±330.5
MPLCS-R(P)	492.9±92.6	**19.6±1.9**	**147.1±23.3**
MPLCS-RS+R(P)	**229.2±292.8**	20.1±2.8	207.1±281.3

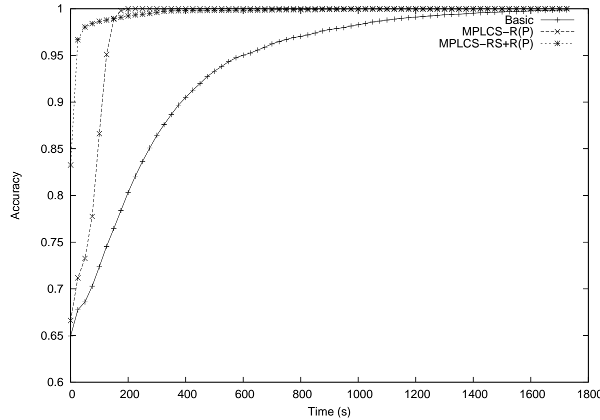

Figure 12: Time vs Accuracy for the kDNF dataset

6.2.3 Noisy 20-bit multiplexer

Table 8 contains the results of testing GAssist and MPLCS on the noisy version of the 20 bit multiplexer. The objective of these tests is to determine what degree of noise can MPLCS cope with. All of the tested configurations managed to learn this dataset and converge to the maximum possible accuracy for these datasets (95% acc. for 5% noise, 90% acc. for 10% noise, etc.). First of all, it is interesting to remark that GAssist is not affected by this kind of noise almost at all. It uses similar amount of iterations and run-time for all tested levels of noise. MPLCS is affected by noise, specially the MPLCS-RS+R(P) configuration: while it is the best configuration for 5% and 10% noise, it becomes the worst configuration for higher levels of noise. MPLCS-R(P) reacts much better to noise. Even if it has slightly slower learning rate with increasing level of noise, it always performs better than GAssist, and better than MPLCS-RS+R(P) when the noise is higher than 10%. Thus, this setting is showing to be relatively robust to noise. In the five tested levels of noise, all performance differences were significant, according to the statistical tests. Figure 13 plots training accuracy of the best individual of the population against run-time.

6.2.4 37 and 70 bit multiplexer

Table 9 contains the results of testing GAssist and MPLCS on these two larger versions of the multiplexer datasets. MPLCS-RS+R(P) was unable to learn the 37 bit multiplexer in a reasonable number of iterations, and neither MPLCS-RS+R(P) nor GAssist could for the 70 bit one. Figure 14 plots accuracy versus run-time of the best individual of the population for both datasets. MPLCS-RS+R(P) may have converged to the perfect solution with more iterations for the 37 bits multiplexer, but definitely not for the 70 bits one. In this dataset we observe the same problem that this MPLCS configuration had for the kDNF dataset, but much more amplified. If the rule set-wise operator does

Table 8: Results of the experiments on the noisy 20 bit multiplexer. Best configuration is marked in bold

Noise degree	Category	Iter.	#Rules	Run-time(s)
5%	Basic	1115.2±176.1	27.4±5.0	419.7±72.5
	MPLCS-R(P)	349.2±120.7	19.6±1.6	120.5±33.1
	MPLCS-RS+R(P)	**21.8±8.3**	**17.0±0.0**	**27.2±7.1**
10%	Basic	1104.7±140.2	27.0±4.6	416.5±59.9
	MPLCS-R(P)	630.9±73.0	21.9±2.5	198.3±23.2
	MPLCS-RS+R(P)	**82.8±33.0**	**17.0±0.2**	**75.9±26.3**
15%	Basic	1170.0±175.9	28.0±5.6	437.4±71.9
	MPLCS-R(P)	764.3±80.3	23.2±3.2	**241.2±27.1**
	MPLCS-RS+R(P)	**724.6±321.4**	**22.9±6.2**	665.8±327.0
20%	Basic	1191.4±183.2	28.3±6.3	444.1±78.2
	MPLCS-R(P)	**856.2±79.8**	**24.6±3.6**	**269.4±32.4**
	MPLCS-RS+R(P)	1208.3±134.8	29.5±5.8	1198.1±176.8
25%	Basic	1178.0±150.9	28.1±5.2	442.1±62.8
	MPLCS-R(P)	**922.0±93.9**	**24.8±3.7**	**295.9±38.9**
	MPLCS-RS+R(P)	1440.3±187.0	30.6±5.1	1442.6±242.2

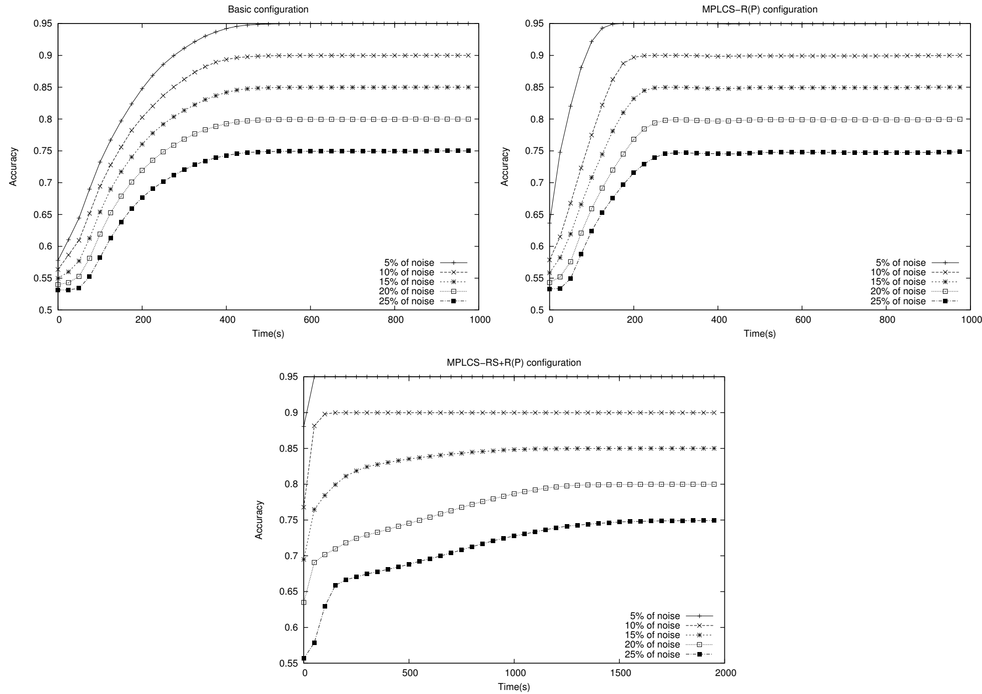

Figure 13: Time vs Accuracy for the noisy 20 bit multiplexer dataset

not have a good supply of rules, its exploitation power is more damaging than beneficial, and this situation becomes critical in datasets with very large search space, such as these two versions of the multiplexer. On the other hand, MPLCS-R(P) can properly explore the larger search space of these datasets, and it was able to solve these problems in a few iterations and short time, converging to solutions almost identical to the optimal solution (33 rules for the 37 bit multiplexer, 65 for the 70 bit one). Actually, if we let

MPLCS train for longer (after it has reached perfect accuracy) it always converged to the optimal rule on the 37 bit multiplexer and in six out of ten runs for the 70 bit one.

Table 9: Results of the experiments on the 37 and 70 bits multiplexer datasets. Best configuration is marked in bold

Dataset	Category	Iter.	#Rules	Run-time(s)
37 bits	Basic	6049.8±1499.3	41.9±5.4	176.3±124.1
	MPLCS-R(P)	**185.0±44.4**	**34.4±1.1**	**5.9±0.9**
	MPLCS-RS+R(P)	—	—	—
70 bits	Basic	—	—	—
	MPLCS-R(P)	**741.9±128.1**	**71.6±2.6**	**41.7±10.9**
	MPLCS-RS+R(P)	—	—	—

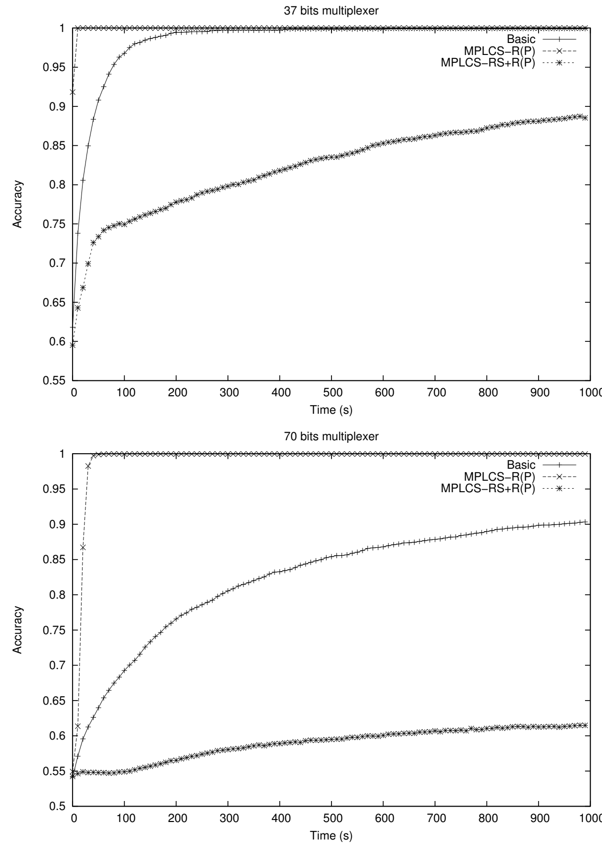

Figure 14: Time vs Accuracy for the 30 and 70 bits multiplexer datasets

6.3 Comparison of MPLCS against other LCS systems

Michigan LCSs employ the number of learning steps as performance metric. When using the ILAS windowing scheme, each GA iteration of MPLCS evaluates $|T|/s$ examples, being $|T|$ the training set size and s the number of strata. Thus, one MPLCS iteration is roughly equivalent to $|T|/s$ learning steps. Table 10 contains the conversion of MPLCS-R(P)'s performance to learning steps, allowing us to compare MPLCS to Michigan LCS

systems. This comparison is specially interesting for the 37 and 70 bit multiplexer datasets, where we have optimized the number of strata to be as high as possible, thus reducing to the minimum the number of learning steps per GA iteration. This makes the learning process of MPLCS more incremental, so closer to how a Michigan LCS learns.

Table 10: Performance of MPLCS-R(P) converted to learning steps

Dataset	#instances per iteration	#learning steps
20 bit multiplexer	5243	124259
37 bit multiplexer	1373	254190
70 bit multiplexer	1574	1167751
noisy 20 bits multiplexer - 5% noise	5243	1830856
noisy 20 bits multiplexer - 10% noise	5243	3307809
noisy 20 bits multiplexer - 15% noise	5243	4007225
noisy 20 bits multiplexer - 20% noise	5243	4489057
noisy 20 bits multiplexer - 25% noise	5243	4834046
kDNF	5243	2584275

We compared against XCSBOA [Butz, 2004], an XCS extended with structural learning based on the Bayesian Optimization Algorithm. As no exact numbers are reported in the literature (just the plots of the online performance are displayed), our comparison is only approximate. For the 20 bit multiplexer, XCSBOA performs similarly to the standard XCS [Butz, 2007], and XCS+TS can learn this dataset in 35000 learning steps using its default parameters. For the 37 and 70 bits multiplexer, results for XCSBOA are reported in [Butz and Pelikan, 2006]. XCSBOA can learn the 37 bits multiplexer in approximately 200000 learning steps, and the 70 bits multiplexer in around 900000 learning steps. Results for the noisy 20 bits multiplexer are reported in [Butz et al., 2005] for XCS+TS for levels of noise of 5%, 10% and 15%. This system could converge to an optimal solution in 50000 learning steps for 5% of noise, 100000 learning steps for 10% of noise. For 15% of noise, XCS was not able to converge to an optimal solution after 300000 learning steps. No further results are reported for higher levels of noise. Finally, results for the k-DNF dataset for XCSBOA are reported in [Butz and Pelikan, 2006]. This system can learn up to 98% accuracy in 120000 learning steps, which roughly means learning all but one of the 22 terms (i.e. rules) that define the k-DNF formula. No performance levels were reported on how many learning steps did it take to reach 100% accuracy.

XCSBOA usually obtains better results than MPLCS according to this metric. However, the two datasets where we have tuned the number of strata of the ILAS windowing system to reduce as much as possible the number of instances per iteration (the 37 and 70 bits multiplexer) are precisely the two datasets with smallest performance difference. In these two datasets the performance of both systems (200000 learning steps vs 254190 in the 37 bits multiplexer and 900000 learning steps vs 1167751 in the 70 bits multiplexer) is quite comparable.

Nevertheless, using learning steps as a performance metric is not entirely fair for MPLCS. As we have shown in our experiments, MPLCS-RS and MPLCS-RS+R use a much lower number of iterations (and therefore also of learning steps) than MPLCS-R, even it they are much slower in terms of run time. To further elaborate the comparison with XCS in this direction we performed the following run-time comparison experiment:

1. We downloaded Martin Butz's XCS code [Butz, 2003] from `http://www.illigal.uiuc.edu/pub/src/XCS/XCS1.2.tar.Z` and run it in the same machine we have

employed to perform our experiments.

2. We run XCS on the 20 bits multiplexer using XCS default parameters for 35000 learning steps (the number of learning steps it needs to converge to an optimal solution). The experiment was repeated 10 times with different initial random seeds. In average it takes XCS 19.7 seconds to complete a run.

3. To have a fair comparison we took the serial implementation (not the master-slave parallel one) of MPLCS and also used the MPLCS code where the instances of the multiplexer function are generated on the fly (not loaded from disk as we did for the mx20 experiments reported in the paper) for the same reason.

4. We took MPLCS-R(P) and run it on the 20 bits multiplexer using 2500 strata of the ILAS windowing scheme. It converges in approximately 100 iterations. Each iteration uses $2^{20}/2500 = 419.4$ examples. Thus, 100 iterations are equivalent to approximately 42000 learning steps. Again, we repeated this experiment ten times. In average it takes MPLCS 3 seconds to complete a run.

Table 11 summarizes this small experiment. MPLCS-R(P) uses relatively similar number of learning steps to converge to the optimal solutions, while requiring much less computational effort. Overall, the comparison of MPLCS with XCS(BOA) is quite inconclusive, as it gives opposite results depending on the metric, which raises a question: what is the most fair performance metric that we can employ (if any)?

Table 11: Run-time comparison between XCS and MPLCS on the 20 bits multiplexer

System	Learning steps	Run-time per run(s)
XCS	35000	19.7
MPLCS-R(P)	42000	3

Finally, we can also compare MPLCS against $\chi eCCS$ [Llorà et al., 2006], a Pittsburgh LCS that uses ECGA to perform structural learning. It takes $\chi eCCS$ approximately 11 iterations to learn the 20 bits multiplexer and 24 iterations to learn the 37 bits multiplexer. Considering that this method does not uses any windowing scheme, each of its iterations using the whole training set, its performance level in learning steps is much worse than MPLCS or XCS. Nevertheless, we do not think that this comparison (using the learning steps as a performance metric) is fair without a properly tuned windowing scheme, because we have showed in this paper and elsewhere [Bacardit et al., 2004] that Pittsburgh LCS can benefit from such methods.

7 Discussion

In this section we take a step back and discuss the general implications of the results presented in the previous section. Our analysis covers the impact that the studied LS operators have when integrated with the underlying GA, detailing potentially positive and negative aspects of this impact. We also discuss the applicability of our methods to other LCS paradigms.

7.1 Integration of the LS operators within the evolutionary cycle

Learning Classifier Systems are sophisticated methods composed of multiple components. Their success depends on a proper integration of these subsystems. Therefore, it is important to analyze and understand the behavior of each component as to identify

how can it contribute to the overall success of the system. This kind of analysis has been extensively performed in the past for Michigan LCS [Butz et al., 2004], where the different components of the system were analyzed separately to model the amount of pressure that they introduce into the LCS. This pressure was expressed into a one dimensional specificity-generality scale. Some components introduce generality pressure, other components introduce specificity pressure.

Such kind of analysis would also bring great insight into MPLCS and the potential integration of these LS mechanisms into other kinds of LCS. Moreover, the evolutionary pressure analysis of MPLCS has to be more complex than in Michigan LCS, as there are pressures that influence at the rule level, while other influence at the rule-set level. This analysis deserves a paper of its own, and in this subsection we provide only a preliminary discussion derived from the results reported in this paper.

The rule-wise operators can introduce pressure at the rule level, changing the individual generality or specificity of the rules, but they can also introduce substantial effects at the rule set level. Figure 9 showed we have a clear example of this for various settings of MPLCS-R(P), that plotted the evolution of the average rule set size of the individuals through the GA iterations. All configurations including the Rule Splitting (RS) operator have much higher rule set size than the other configurations. Thus, it is clear that this operator introduces specificity pressure at the rule set level that complements the existing specificity pressure introduced by the bloat effect [Langdon, 1997], that exists in most variable-length representations in evolutionary computation. This pressure, at the LS level, is balanced by the application of Rule Cleaning (RC) and, specially, the Rule Generalizing (RG) operator. At the evolutionary level it is balanced by the rule deletion operator and, specially, the MDL fitness function. The rule set-wise operator introduces generality pressure. This is shown in many places in the results. For instance, the MPLCS variants where the RSW operator was present were the ones that most often generated the optimal rule sets for the tested datasets. This pressure is, in many cases, excessive and generating over-general rule sets that slow down or even stall the learning process. Examples of this are the kDNF dataset, where the RSW operator could learn all but one of the optimal rules but then needed much longer time to learn the last one, or its inability to learn the MP37 or MP70 datasets. This extreme generality pressure shows that the operator is not able to produce the proper balance between exploration and exploitation.

In relation to pressure at the rule level we note that by design the RC and RS operator introduce specificity pressure while the RG operator introduces generality pressure. The best results in relation to the combination of the operators were produced consistently when the three operators were used simultaneously. Their contribution, however is not equal. The RS operator appears in the top three settings of MPLCS-R(P) (table 2) and MPLCS-RS+R(P) (table 4), but also in the worst configuration for both experiments. Thus our results suggest that while being a crucial operator, it must be compensated to avoid over-specific (bloated) individuals. Both RC and RG are able to compensate it, but using alternative ways. RG compensates it by explicitly applying generalization pressure. RC follows an indirect way: We think that it, sometimes, disables all the literals associated to an attribute, thus creating rules that cannot match any example. Afterwards, these rules are removed by the rule deletion operator, thus introducing the generality pressure due to RC's action. The MDL-based fitness function also introduces generality pressure at a rule level.

The combination of the standard recombination operators with the RW operators, when applied as a group, is beneficial in most of the experiments reported in the paper.

On the other hand, the RSW operator seems, in most cases, to only introduce unnecessary computational effort and it struggles to learn in the most challenging datasets such as the MP37, MP70 or kDNF datasets. Thus we can say that the integration of the RSW operator with the rest of the system, as studied here, is not successful, as neither the standard genetic operators nor the RW operators could synergistically compensate the generality pressure it introduces. This is a clear difference with RS. Both of them perform bad on their own, but the action of RS, when properly compensated, appears in all the best performing settings.

7.2 Applicability of the LS operators across LCS paradigms

Our work should be of interest to, not only Pittsburgh LCS researchers, but also other flavours of LCS and Genetics-Based Machine Learning (GBML). The RSW operator could be easily applied to other kinds of Pittsburgh systems, such as Fuzzy LCS systems, as it is rule representation neutral, and could be relatively easily adapted to deal with unordered rules. It the rules are not overlapped, the same operator would work straight away. In case of overlap, a more complex conflict resolution criterion should be enacted, but the rest of the heuristic could be maintained. Also, most of the results presented in this paper show that the operator performs well if it is supplied with good rules. Thus, there is great potential for using the operator as post-processing stage for any kind of LCS to compact rule sets. For instance, both CCS and $\chi eCCS$ have shown to evolve optimal rules for the multiplexer problems, but do not have any clear mechanism to aggregate the best rules into a final solution. This operator could achieve this aim. The three rule-wise operators could perfectly be used in any Pittsburgh or Michigan LCS and also, for instance, in methods using the Iterative Rule Learning [Venturini, 1993] approach of GBML.

8 Conclusions and further work

In this paper we have extended our previous work [Bacardit and Krasnogor, 2006] on the design of heuristic local search mechanism applied to the GAssist Pittsburgh LCS. In this work we exploit information extracted from the supervised learning process thus editing the rule sets to improve training accuracy. We call this extended framework Memetic Pittsburgh Learning Classifier System (MPLCS). We studied two kinds of operators. One, MPLCS-RS, was proposed previously but has been extended in this work by evaluating other possible places of its integration in the GA cycle. This operator recombines rules from many different parents to generate a single rule set with maximum possible accuracy and compactness. The new rule-wise operators (collectively named MPLCS-R) have been proposed in this paper. Two of these operators (rule cleaning and splitting) edit the rules to eliminate some of their misclassifications, therefore, applying specificity pressure. The third operator (rule generalizing) adds literals to the rules in order to classify correctly more examples, therefore applying generalization pressure. These operators have been tailored to the *GABIL* [DeJong and Spears, 1991] knowledge representation that GAssist uses. Both classes of operators can be used together, and we have also tested different policies for the different stages within the GA cycle where all these mechanisms can be applied.

We performed a large scale evaluation process testing many different algorithms arising from the combinations of operators and policies. Our aim was to identify the MPLCS variants that generate optimal solutions and converge to them after seeing the least amount of training examples, scale well, are able to cope with noise and use the least amount of computational effort. Our experiments allow us to conclude the

following: (1) the best results are always obtained by the proper combination of rule-wise operators applying generality and specificity pressure (RC+RS+RG). Thus, the proper equilibrium of specificity and generality pressures is the key for a successful learning process. (2) The best MPLCS variant overall was MPLCS-R(P), where the three studied rule-wise LS operators are used simultaneously and are applied probabilistically across the whole population. Another MPLCS algorithm, MPLCS-RS+R(P), which extends the previously mentioned configuration with the RSW operator, also generated good results in our first stage of experiments (using the 20 bit multiplexer). However, when tested later in other datasets (with higher difficulty, larger size or noise) this configuration showed inferior performance, due to a lack of balance between exploration and exploitation of the rule set-wise operator. (3) We also tested the combination of these operators with the ILAS windowing scheme, designed to alleviate the run-time of GBML methods in large datasets, and the experiments show that our proposed LS methods and ILAS can be combined successfully, i.e., they act synergistically. When combined with ILAS using proper settings, MPLCS is able to obtain a performance comparable to state-of-the-art LCS systems such as XCS(BOA) and $\chi eCCS$.

We also analyzed how well do the different LS mechanisms integrate with the standard evolutionary exploration mechanisms as well as between them, and analyzed their applicability to other kinds of LCS methods, which leads us to affirm that there is general value to the design and study of LS operators that we have performed, as they can be applied to any other LCS systems using the GABIL representation, and easily adapted to other representations too. Finally, our studies also show that the design issues associated to traditional MAs [Krasnogor and Smith, 2005], such as the effective combination of different LS operators with the appropriate regime of application of each of them, elitist or population-wise application of the operators, etc., also apply to Memetic LCS.

Future work includes: (1) Testing MPLCS on other datasets, eg. hierarchical decomposable problems such as the Parity-Multiplexer domain [Butz, 2004] to determine what are the limits of the rule-wise search operators. (2) The rule set-wise operator has shown to have some limitations on very large datasets. However, in the datasets where this operator worked properly, it always managed to obtain the most compact and well generalized solutions. Therefore, it would be interesting to study other policies of application of this operator, for instance, as a refining stage after finding good candidate rules. This kind of refining stage, if designed properly, could potentially be applied to rules evolved with any kind of LCS. (3) In a more general sense, we would like to rigorously model what conditions are required for the proper integration of the LS operators with the evolutionary exploration mechanisms of LCS systems, and how to adjust the LS operators to achieve these conditions. (4) Integrating the rule-wise local search mechanisms studied in this paper into other kinds of LCS methods. (5) The studied operators could be extended in two directions: (a) to adapt the rule-wise operators to deal with real-world datasets with noise and inconsistency and (b) to adapt them to deal with real-valued attributes. (6) From a MA point of view, in this paper we have only applied very simple strategies for integrating LS into GAssist. In future work we will investigate other important design issues of MAs as suggested in [Krasnogor and Smith, 2005].

9 Acknowledgements

We acknowledge the support of the UK Engineering and Physical Sciences Research Council (EPSRC) under grants GR/T07534/01 and EP/E017215/1. We are grateful for the use of the University of Nottingham's High Performance Computer. Finally, we would like to thank the useful suggestions of the anonymous reviewers.

References

[Bacardit, 2004] Bacardit, J. (2004). *Pittsburgh Genetics-Based Machine Learning in the Data Mining era: Representations, generalization, and run-time*. PhD thesis, Ramon Llull University, Barcelona, Catalonia, Spain.

[Bacardit, 2005] Bacardit, J. (2005). Analysis of the initialization stage of a pittsburgh approach learning classifier system. In *GECCO 2005: Proceedings of the Genetic and Evolutionary Computation Conference*, volume 2, pages 1843–1850. ACM Press.

[Bacardit and Garrell, 2007] Bacardit, J. and Garrell, J. M. (2007). Bloat control and generalization pressure using the minimum description length principle for a pittsburgh approach learning classifier system. In *Learning Classifier Systems, Revised Selected Papers of the International Workshop on Learning Classifier Systems 2003-2005*, pages 59–79. Springer-Verlag, LNCS 4399.

[Bacardit et al., 2004] Bacardit, J., Goldberg, D., Butz, M., Llorà, X., and Garrell, J. M. (2004). Speeding-up pittsburgh learning classifier systems: Modeling time and accuracy. In *Parallel Problem Solving from Nature - PPSN 2004*, pages 1021–1031. Springer-Verlag, LNCS 3242.

[Bacardit and Krasnogor, 2006] Bacardit, J. and Krasnogor, N. (2006). Smart crossover operator with multiple parents for a pittsburgh learning classifier system. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1441–1448, New York, NY, USA. ACM Press.

[Bäck et al., 1997] Bäck, T., Fogel, D. B., and Michalewicz, Z., editors (1997). *Handbook of Evolutionary Computation*. IOP Publishing Ltd., Bristol, UK, UK.

[Blake et al., 1998] Blake, C., Keogh, E., and Merz, C. (1998). UCI repository of machine learning databases. (www.ics.uci.edu/mlearn/MLRepository.html).

[Butz et al., 2004] Butz, M., Kovacs, T., Lanzi, P. L., and Wilson, S. W. (2004). Toward a theory of generalization and learning in xcs. *IEEE Transactions on Evolutionary Computation*, 8(1):28–46.

[Butz, 2003] Butz, M. V. (2003). Documentation of xcs+ts c-code 1.2. Technical Report 2003023, Illinois Genetic Algorithms Lab, University of Illinois at Urbana-Champaign.

[Butz, 2004] Butz, M. V. (2004). *Rule-based Evolutionary Online Learning Systems: Learning Bounds, Classification and Prediction*. PhD thesis, University of Illinois at Urbana-Champaign.

[Butz, 2007] Butz, M. V. (2007). Personal communication.

[Butz and Pelikan, 2006] Butz, M. V. and Pelikan, M. (2006). Studying xcs/boa learning in boolean functions: structure encoding and random boolean functions. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1449–1456, New York, NY, USA. ACM.

[Butz et al., 2006] Butz, M. V., Pelikan, M., Llorà, X., and Goldberg, D. E. (2006). Automated global structure extraction for effective local building block processing in xcs. *Evol. Comput.*, 14(3):345–380.

[Butz et al., 2005] Butz, M. V., Sastry, K., and Goldberg, D. E. (2005). Strong, stable, and reliable fitness pressure in xcs due to tournament selection. *Genetic Programming and Evolvable Machines*, 6(1):53–77.

[Cantu-Paz, 2000] Cantu-Paz, E. (2000). *Efficient and Accurate Parallel Genetic Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA.

[Casillas et al., 2007] Casillas, J., Carse, B., and Bull, L. (2007). Fuzzy-xcs: a michigan genetic fuzzy system. *IEEE Transactions on Fuzzy Systems*, 15(4):536–550.

[DeJong and Spears, 1991] DeJong, K. A. and Spears, W. M. (1991). Learning concept classification rules using genetic algorithms. In *Proceedings of the International Joint Conference on Artificial Intelligence*, pages 651–656. Morgan Kaufmann.

[Grefenstette, 1991] Grefenstette, J. J. (1991). Lamarckian learning in multi-agent environments. In Belew, R. and Booker, L., editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 303–310, San Mateo, CA. Morgan Kaufman.

[Harik, 1999] Harik, G. (1999). Linkage learning via probabilistic modeling in the ecga. Technical Report 99010, Illinois Genetic Algorithms Lab, University of Illinois at Urbana-Champaign.

[Harik, 1995] Harik, G. R. (1995). Finding multimodal solutions using restricted tournament selection. In Eshelman, L., editor, *Proceedings of the Sixth International Conference on Genetic Algorithms*, pages 24–31, San Francisco, CA. Morgan Kaufmann.

[Harik et al., 1999] Harik, G. R., Lobo, F. G., and Goldberg, D. E. (1999). The compact genetic algorithm. *IEEE-EC*, 3(4):287.

[Janikow, 1991] Janikow, C. (1991). *Indictive Learning of Decision Rules in Attribute-Based Examples: a Knowledge-Intensive Genetic Algorithm Approach*. PhD thesis, University of North Carolina.

[Kearns and Vazirani, 1994] Kearns, M. J. and Vazirani, U. V. (1994). *An introduction to computational learning theory*. MIT Press, Cambridge, MA, USA.

[Krasnogor and Smith, 2005] Krasnogor, N. and Smith, J. (2005). A tutorial for competent memetic algorithms: model, taxonomy and design issues. *IEEE Transactions on Evolutionary Computation*, 9(5):474– 488.

[Langdon, 1997] Langdon, W. B. (1997). Fitness causes bloat in variable size representations. Technical Report CSRP-97-14, University of Birmingham, School of Computer Science. Position paper at the Workshop on Evolutionary Computation with Variable Size Representation at ICGA-97.

[Lanzi et al., 2006] Lanzi, P. L., Loiacono, D., Wilson, S. W., and Goldberg, D. E. (2006). Prediction update algorithms for XCSF: RLS, kalman filter, and gain adaptation. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1505–1512, New York, NY, USA. ACM Press.

[Larranaga and Lozano, 2002] Larranaga, P. and Lozano, J., editors (2002). *Estimation of Distribution Algorithms, A New Tool for Evolutionnary Computation*. Genetic Algorithms and Evolutionnary Computation. Kluwer Academic Publishers.

[Llora et al., 2005] Llora, X., Sastry, K., and Goldberg, D. (2005). The compact classifier system: Scalability analysis and first results. In *Proceedings of the Congress on Evolutionary Computation 2005*, volume 1, pages 596–603. IEEE Press.

[Llorà et al., 2006] Llorà, X., Sastry, K., Goldberg, D. E., and delaOssa, L. (2006). The x-ary extended compact classifier system: Linkage learning in pittsburgh lcs. In *Proceedings of the 9th International Workshop on Learning Classifier Systems - IWLCS2006*. (in press), LNAI, Springer-Verlag.

[Pelikan et al., 1999] Pelikan, M., Goldberg, D. E., and Cantú-Paz, E. (1999). BOA: The Bayesian optimization algorithm. In *Proceedings of the Genetic and Evolutionary Computation Conference GECCO-99*, volume I, pages 525–532. Morgan Kaufmann.

[Rissanen, 1978] Rissanen, J. (1978). Modeling by shortest data description. *Automatica*, vol. 14:465–471.

[Valiant, 1984] Valiant, L. G. (1984). A theory of the learnable. *Commun. ACM*, 27(11):1134–1142.

[Venturini, 1993] Venturini, G. (1993). Sia: A supervised inductive algorithm with genetic search for learning attributes based concepts. In Brazdil, P. B., editor, *Machine Learning: ECML-93 - Proc. of the European Conference on Machine Learning*, pages 280–296. Springer-Verlag, Berlin, Heidelberg.

[Wilson, 1995] Wilson, S. W. (1995). Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175.

[Wyatt and Bull, 2004] Wyatt, D. and Bull, L. (2004). A memetic learning classifier system for describing continuous-valued problem spaces. In Hart, W., Krasnogor, N., and Smith, J., editors, *Recent Advances in Memetic Algorithms*, pages 355–396. Springer.