

**Fast Fitness Implementation
of Multiplexer Problems
for Pittsburgh LCS**

**Xavier Llorà
IlliGAL Report No. 2006017
April, 2006**

Illinois Genetic Algorithms Laboratory
University of Illinois at Urbana-Champaign
117 Transportation Building
104 S. Mathews Avenue Urbana, IL 61801
Office: (217) 333-2346
Fax: (217) 244-5705

Fast Fitness Implementation of Multiplexer Problems for Pittsburgh LCS

Xavier Llorà

National Center for Supercomputing Applications,
University of Illinois at Urbana-Champaign,
1205 W. Clark Street, Urbana, IL 61801, USA
xllora@illigal.ge.uiuc.edu

Abstract

This technical report describes how to compute the fitness of a rule for an arbitrary size multiplexer without doing any instance matching. Pittsburgh-style learning classifier systems require the *accuracy* and the *error* of a rule to compute a fitness that promotes maximally accurate and maximally general rules. The *accuracy* (α) may be computed as the proportion of overall examples correctly classified, and the *error* (ε) is the proportion of incorrect classifications issued. Once the *accuracy* and *error* of a rule are known, the fitness can be computed as $f(r) = \alpha(r) \cdot \varepsilon(r)$. This technical note shows how to compute the fitness only by inspecting the rule, requiring a time proportional to number of possible address values $\mathcal{O}(2^{|a|})$ instead of the $\mathcal{O}(2^{|a|+2^{|a|}})$ that requires a traditional rule matching strategy against all the possible instances. The proposed method makes tractable for Pittsburgh learning classifier systems multiplexer problems larger than the 11-input one.

1 Motivation

In order to promote maximally general and maximally accurate rules a la XCS (Wilson, 1995), CCS and later χ eCCS compute the *accuracy* (α) and the *error* (ε) of an individual (Llorà, Sastry, & Goldberg, 2005; Llorà, Sastry, & Goldberg, 2006). In a Pittsburgh-style classifier, the accuracy may be computed as the proportion of overall examples correctly classified, and the error is the proportion of incorrect classifications issued. Let n_{t+} be the number of positive examples correctly classified, n_{t-} the number of negative examples correctly classified, n_m the number of times that a rule has been matched, and n_t the number of examples available. Using these values, the *accuracy* and *error* of a rule r can be computed as:

$$\alpha(r) = \frac{n_{t+}(r) + n_{t-}(r)}{n_t} \quad (1)$$

$$\varepsilon(r) = \frac{n_{t+}}{n_m} \quad (2)$$

The error (equation 2) only takes into account the number of correct positive examples classified. This is due to the close-world assumption of the knowledge representation which follows from using

Table 1: Number of total instances to match for a given multiplexer size.

Address bits ($ a $)	Problem size ($\ell = a + 2^{ a }$)	Number of instances ($\Sigma = 2^\ell$)
1	3	8
2	6	64
3	11	2048
4	20	1.05e+06
5	37	1.37e+11
6	70	1.18e+21
7	135	4.36e+40

a default rule. Once the *accuracy* and *error* of a rule are known, the fitness can be computed as follows.

$$f(r) = \alpha(r) \cdot \varepsilon(r) \quad (3)$$

The above fitness measure favors rules with a good classification accuracy and a low error, or maximally general and maximally accurate rules.

However, computing the fitness measure presented in equation 3 requires to match all the possible multiplexer instances. Given a multiplexer with ℓ conditions ($\ell = |a| + 2^{|a|}$ where $|a|$ is the number of address bits), matching 2^ℓ instances to compute the fitness rapidly becomes intractable due its non-polynomic nature. Table 1 shows, given a multiplexer size, the number of instances to match to compute the exact fitness of a rule after equation 3. The complexity of this process is bounded by $\mathcal{O}(2^\ell)$.

The rest of this short note presents how the same exact fitness can be computed without any instance matching. I introduce in section 2 how the fitness can be computed by only decoding the possible addresses encoded by the rule and inspecting the value in the decoded address. The method, as later explained, assumes a close world with only positive and negative examples. The complexity of the proposed technique is reduced to $\mathcal{O}(2^{|a|})$, making tractable large multiplexers previously out of reach for Pittsburgh LCS (Llorà, Sastry, & Goldberg, 2006).

2 Fast Fitness Implementation for Multiplexer Problems

The method proposed below does not require any instance matching; it only requires inspecting the rule. The method assumes that if a rule is matched, then the instance matched by the rule is labeled as positive. If the instance is not matched, then it is labeled as negative instance—default rule. The idea behind the fast fitness implementation given a rule r for multiplexer problems summarizes as follows:

1. Compute the specificity s_i of the input bits of the rule r .
2. Compute the number of possible input states n_{is} described by the rule r as $n_{is} = 2^{2^{|a|} - s_i}$.
3. Create a set \mathcal{A} containing all the possible addresses encoded in the rule r .
4. For each $a \in \mathcal{A}$
 - (a) If $r[a]=1$ then all the instances described by this address (n_{is}) are correctly classified.
 - (b) If $r[a]=0$ then all the instances described by this address (n_{is}) are incorrectly classified.

- (c) If $r[a]=\#$ then the instances described by this address (n_{is}) are split in two equal size sets of correctly ($n_{is}/2$) and incorrectly ($n_{is}/2$) classified.

For instance, let's assume that $r=0\#01\#\#$. The first step is to compute the specificity of the input conditions of $r-s_i = 2$ —and the number of possible input states described by $r-n_{is} = 2^2 = 4$. The set \mathcal{A} contains two possible addresses $\{00, 01\}$. Given the address $a=00$ the 0^{th} input bit is wrongly set (0). Thus, n_{is} will be incorrectly classified. For the next address in \mathcal{A} , $a=01$, the $= 1^{st}$ input bit is correctly set, hence, n_{is} will be correctly classified¹. It is also true that the total number of states matched by the rule is $2^{\ell-s}$, where s is the specificity of the rule. Given $n_{t+} = 2^2$, $n_{t-} = 2^5 - 2^2$, $n_m = 2^3$, and $n_t = 2^6$, the fitness of r is computed using equation 3 as:

$$f(r) = \alpha(r) \cdot \varepsilon(r) = \frac{2^2 + 2^5 - 2^2}{2^6} \cdot \frac{2^2}{2^3} = \frac{4 + 32 - 4}{64} \cdot \frac{4}{8} = 0.5 \cdot 0.5 = 0.25 \quad (4)$$

The complexity of this approach is $\mathcal{O}(2^{|a|})$, and do not require any instance matching matching. Further details about the implementation of this fitness function for arbitrary length multiplexer problems is provided in appendix A. This appendix contains a Java implementation of the method described above.

3 Disclaimer

The code described in this technical report is distributed for academic purposes only under GPL license. It has no warranty implied or given, and the author assume no liability for damage resulting from its use or misuse. If you have any comments or find any bugs, please send an email to `xllora@illigal.ge.uiuc.edu`.

4 Acknowledgments

This work was also sponsored by the Air Force Office of Scientific Research, Air Force Materiel Command, USAF, under grant FA9550-06-1-0096, and the National Science Foundation under grant IIS-02-09199. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the funding agencies.

References

- Llorà, X., Sastry, K., & Goldberg, D. E. (2005). The compact classifier system: motivation, analysis, and first results. *Proceedings of the Congress on Evolutionary Computation*, 1, 596–603. (Also IlliGAL Report No. 2005019).
- Llorà, X., Sastry, K., & Goldberg, D. E. (2006). The χ -ary extended compact classifier system: Linkage learning in pittsburgh lcs. In *IWLCS2006: Proceedings of the 2006 Workshop of the Conference on Genetic and Evolutionary Computation* pp. in press. Seattle WA, USA: ACM Press. (Also IlliGAL Report No. 2006015).
- Wilson, S. (1995). Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2), 149–175.

¹In case of having a # on the i^{th} input position, half of the possible input states may have a 0 and half of them a (1), giving a half correct-half incorrect distribution of the possible input states.

A Java Implementation of the Fast Fitness Computation of Multiplexer Problems

```
1  /*
   * A class that implements a fast fitness calculation for multiplexer
   3  * problems for Pittsburgh LCS.
   * Copyright (C) 2006 Xavier Llorca
   5  *
   * This program is free software; you can redistribute it and/or
   7  * modify it under the terms of the GNU General Public License
   * as published by the Free Software Foundation; either version 2
   9  * of the License, or (at your option) any later version.
   *
  11  * This program is distributed in the hope that it will be useful,
   * but WITHOUT ANY WARRANTY; without even the implied warranty of
  13  * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
   * GNU General Public License for more details.
  15  *
   * You should have received a copy of the GNU General Public License
  17  * along with this program; if not, write to the Free Software
   * Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, MA
  19  * 02110-1301, USA.
   */
  21 package edu.uiuc.illegal.ga.fitness.functions.gbml;
  23
   import java.math.BigInteger;
  25 import java.util.Vector;

  27 import edu.uiuc.illegal.ga.fitness.Fitness;
   import edu.uiuc.illegal.ga.fitness.functions.FitnessFunction;
  29 import edu.uiuc.illegal.ga.individual.XaryIndividual;

  31 /** Computes the fitness of a Xary rule {0,1,#} for a given multiplexer
   * task. The fitness function used is the one described in "The compact
  33 * classifier system: Motivation, analysis and first results" by
   * Llorca, X., Sastry, K., Goldberg, D. E. (2005). Proceedings of
  35 * the Congress on Evolutionary Computation, 1, 596--603.
   *
   37 * This implementation is meant also to work for MUX 70 and larger.
   *
  39 * @author Xavier Llorca
   *
  41 */
   public class LargeMultiplexerOneRule
  43 extends FitnessFunction {

  45     /** A default serial ID */
   private static final long serialVersionUID = 1L;
  47
   /** The address length */
  49     protected int iAddressLength = 0;

  51     /** The rule length */
   protected int iRuleLength = 0;
  53

   /** Creates a fitness function that implement a multiplexer function
  55 * to evaluate a single rule provided by a XaryIndividual.
   *
  57 * @param iAddressLength The length of the address part
   */
  59     public LargeMultiplexerOneRule ( int iAddressLength )
   {
  61         this.iAddressLength = iAddressLength;
         this.iRuleLength = iAddressLength+(int)Math.pow(2,iAddressLength);
  63     }
}
```

```

65  /** Decode the address value.
66  *
67  * @param iaAdd The address value
68  * @return The decode value
69  */
70  protected int decode ( int [] iaAdd ) {
71      int iVal = 0;
72      int iInc = 1;
73
74      for ( int i=iaAdd.length-1 ; i>=0 ; i-- ) {
75          if ( iaAdd[i]==2 )
76              iVal += iInc;
77          iInc *= 2;
78      }
79
80      return iVal;
81  }
82
83  /** Computes the fitness for an individual encoding and Xary {0,1,#} rule
84  * for the multiplexer problem. This implementations is ment to also work
85  * for MUX 70 and larger.
86  *
87  * @param xi The Xary individual
88  * @param f The fitness to update
89  */
90  public void fitnessForXary ( XaryIndividual xi , Fitness f )
91  {
92      // Declaration variables
93      int          iRuleInCondGen = 0;
94      int []      iaRule = xi.getVariables();
95      Vector vecVals2Check = new Vector();
96      int []      iaTmp = null;
97      int []      iaAux = null;
98      boolean     bGen = true;
99      int         iCnt = 0;
100     // Fitness
101     BigInteger biOk = BigInteger.ZERO;
102     BigInteger biWrong = BigInteger.ZERO;
103     BigInteger biMatched = BigInteger.ZERO;
104     BigInteger biNumIns = BigInteger.ZERO;
105     BigInteger biNumInsD2 = BigInteger.ZERO;
106     BigInteger biTmp = BigInteger.ZERO;
107     BigInteger biTwo = new BigInteger("2");
108     int         iRuleAdd = 0;
109
110     // Initialize the address
111     iaTmp = new int[this.iAddressLength];
112     System.arraycopy(iaRule,0,iaTmp,0,this.iAddressLength);
113     vecVals2Check.addElement(iaTmp);
114     while ( bGen ) {
115         iaTmp = (int []) vecVals2Check.elementAt(0);
116         iCnt = 0;
117         for ( int iMax=this.iAddressLength ;
118             iCnt<iMax && iaTmp[iCnt]>0 ;
119             iCnt++ );
120         if ( iCnt<this.iAddressLength ) {
121             vecVals2Check.removeElementAt(0);
122             iaTmp[iCnt] = 1;
123             vecVals2Check.addElement(iaTmp);
124             iaAux = new int[this.iAddressLength];
125             System.arraycopy(iaTmp,0,iaAux,0,this.iAddressLength);
126             iaTmp[iCnt] = 2;
127             vecVals2Check.addElement(iaAux);
128         }
129         else
130             bGen = false;
131     }

```

```

133 // Compute the rule specificity
134 for ( int i=this.iAddressLength, iMax=this.iRuleLength ; i<iMax ; i++ )
135     if ( iaRule[i]==0 )
136         iRuleInCondGen++;
137
138 // Process each of the entries
139 for ( int i=0, iMax=vecVals2Check.size() ; i<iMax ; i++ ) {
140     // Recover the entry
141     int [] iaAdd = (int []) vecVals2Check.elementAt(i);
142     // Updates
143     iRuleAdd = iaAdd.length+decode(iaAdd);
144     biTmp = biTwo.pow(iRuleInCondGen);
145     switch ( iaRule[iRuleAdd] ) {
146     case 2:
147         // A 2 integer encode a '1'
148         // The rule is correct
149         // Assumes match rules are +
150         // (close world assumption for binary classification)
151         biOk = biOk.add(biTmp);
152         biMatched = biMatched.add(biTmp);
153         break;
154     case 1 :
155         // A 1 integer encode a '0'
156         // The rule is wrong
157         // Assumes match rules are +
158         // (close world assumption for binary classification)
159         biWrong = biWrong.add(biTmp);
160         biMatched = biMatched.add(biTmp);
161         break;
162     case 0 :
163         // A 0 integer encode a '#'
164         // The rule is half half since it has a hash symbol
165         // Assumes match rules are +
166         // (close world assumption for binary classification)
167         biOk = biOk.add(biTmp.divide(biTwo));
168         biWrong = biWrong.add(biTmp.divide(biTwo));
169         biMatched = biMatched.add(biTmp);
170         break;
171     }
172 }
173
174 biNumIns = biTwo.pow(iaRule.length);
175 biNumInsD2 = biNumIns.divide(biTwo);
176
177 BigInteger biCorrect = biOk.add(biNumInsD2.subtract(biWrong));
178
179 double dAcc = biCorrect.doubleValue()/biNumIns.doubleValue();
180 double dErr = biOk.doubleValue()/biMatched.doubleValue();
181
182 f.setFitness(0,(float)(dAcc*dErr));
183 }

```