

Abstract

Pattern matching is a very wide category of algorithms. This project is involved with a specific version of matching that uses binary inputs and ternary conditions/rules. (Input example: 10010, Rule example: 10##1). This kind of pattern matching is used in Learning Classifier Systems (LCS) like the sUpervised Classifier System (UCS), which is a classification system that learns classifications rules from a training database. Software like this, figure out how the labels were produced given the set of inputs. This project is only concerned with the matching process not with the learning as most execution time in UCS is spent to determine whether the rules match inputs or not and this is what is tackled in this project.

The normal solution that is used so far in systems that require this kind of categorizations is to go though each character of each input comparing it with each character of the conditions (sometimes referred to as rules). It is reasonable that this approach is very time consuming. This project solves this problem by tackling it from various angles. Either by writing smart algorithms to reduce the number of comparisons taking place, like the “batch matching” algorithm, or by the use of suitable data structures (like trees) to store and retrieve the rules faster.

Experiments are presented with the results (Section 4), comparing the various approaches between themselves and between the approach being used so far in LCS. Furthermore a new Ternary Tree version is being offered that improves accessing times and deletion times of ternary language values.

The main contributions and achievements of this paper are:

- The proposed algorithm by Dr Tim Kovacs called *batch matching* is implemented and generalized to work with any number and size of inputs and optimized for faster execution times. Many variations are offered and examined, see pages 26-33.
- New versions of Ternary Tree are proposed that speeds insertion and deletion of values represented in ternary language, see pages 33-35.
- Integration of the Ternary Tree solution with the UCS that was implemented by Dr Gavin Brown (University of Manchester) and extended by Dr Tim Kovacs (University of Bristol), see pages 35-37.

The improvements will significantly speed up this kind of pattern classification systems.

Table of Contents

1 Introduction	6
2 Background/History	8
2.1 Relevant Terms.....	8
2.1.1 Machine Learning.....	8
2.1.2 Information vs Data Retrieval	8
2.1.3 Pattern Matching / Recognition	8
2.1.4 Learning Classifier System	9
2.2 Batch matching algorithm.....	10
2.2.1 Kovacs's algorithm	10
2.2.2 Pre-computed match sets	11
2.3 Encoding schemes.....	11
2.3.1 Bit Encoding.....	11
2.3.2 Specificity-based Encoding and Matching.....	12
2.3.3 BitSet in Java	14
2.3.4 Enum in Java.....	15
2.4 Knowledge-based Evaluation	16
2.4.1 Knowledge Incorporation in EAs.....	16
2.4.2 Decision Rules.....	16
2.5 Data Structures.....	18
2.5.1 Java Collections	18
2.5.1.1 JCF Interfaces	18
2.5.1.2 Collections Features	19
2.5.2 Problem Specific Data Structures	20
2.5.2.1 Linear Evaluation	20
2.5.2.2 Efficient Evaluation Structure (EES).....	20
2.5.2.3 R-trees	22
3 Project Design & Implementation.....	24
3.1 Problem	24
3.2 Initial Idea.....	24
3.3 Batch Matching.....	26
3.3.1 Batch Matching Size N.....	26
3.3.1.1 Adjacent Batches	26
3.3.1.2 Distant Batches.....	27
3.3.1.3 Variable Batch Size	27
3.3.1.4 Recursive Batch Matching	29
3.3.2 Data Structures	30
3.3.2.1 ArrayLists	30
3.3.2.2 LinkedLists	30
3.3.2.3 Plain Arrays.....	31
3.3.2.4 Data Structures Issues	31
3.3.3 Other Enhancements.....	31
3.3.3.1 BitSet.....	31
3.3.3.2 Enums.....	32
3.3.3.3 PreComputed Match Sets	32
3.4 Ternary Tree	33
3.4.1 Single Depth.....	33
3.4.2 Multi-level.....	34
3.5 UCS Integration.....	35
4 Experiments/Results.....	38
4.1 Introduction.....	38

4.2 Experiment Code.....	38
4.2.1 Experiment.java	38
4.2.2 Averager.java	39
4.3 Batch Matching.....	39
4.3.1 Naïve vs Batch Matching.....	40
4.3.2 Adjacent vs Distant Batches	41
4.3.3 Adjacent vs Recursive Approach	43
4.3.4 Generalities Test for Adjacent.....	44
4.4 Other Tests	45
4.5 Ternary Tree Tests	47
4.5.1 TTree1 vs TTreeN	47
4.5.2 ArrayLists vs LinkedLists vs TTreeN	49
5 Final Thoughts.....	51
5.1 Critical Evaluation	51
5.2 Algorithms Wrap Up.....	52
5.3 Conclusion	52
5.4 Future Development	53
References.....	54
Appendix A: Variable Batch Size Matching	56
Appendix B: Experiment.java.....	60
Appendix C: Averager.java	62
Appendix D: TTree.java of UCS.....	64

1 Introduction

String searching algorithms (or string matching algorithms) are a very important category of string algorithms. They try to find patterns that have desired structures. In practice, how the pattern is encoded can affect the feasible search algorithms. For example if a variable width encoding is used, then pattern matching can become slow (time proportional to the width). A more specific aspect of pattern matching will be addressed in this project. String matching is a general problem; we are interested in a specific version with binary inputs and ternary conditions. This is how a pattern classification system like UCS [1] works.

Assume that the state of the world, or the input to a machine learner, is expressed with a binary string of length 3 (e.g. 000, 001, 010 etc). Now assume that hypotheses are IF-THEN statements, where the condition also has length 3. For example lets take the hypothesis: IF 01# THEN 0. This implies that if the input matches the pattern 01# then action 0 should be taken. The # is a wildcard that matches both 0 and 1. In other words, 010 and 011 both satisfy the previous condition. Lets call this a language ternary language since conditions can only have 3 values; 0, 1 and #. This notation is widely used and has become the standard for Michigan-style learning classifier systems (LCS). Now suppose inputs are 50 bits long and there are hundreds of thousands of hypotheses, you can see why the problem becomes extremely complicated and efficiency on the pattern matching will greatly affect execution times. Using this ternary language it might be easy to implement but it is expensive to compute and not memory efficient [2].

Attempts to implement an algorithm to solve the above problem will be presented. First steps would be simply to make it work; next step will be to improve the algorithm's efficiency in order to use less processing power and less memory if possible. Most execution time in UCS is spent to determine whether the rules match or not therefore some experiments take place in order to see what approach works better.

Section 2 of the report discusses the history behind this project mentioning any related work that has been done previously in the field. Some techniques that have good potential to be used to speed up batch-matching algorithm are also stated. Also some possible tree types to be used are briefly studied.

The next section is about an algorithm proposed by Tim Kovacs. It starts by simply stating what the algorithm does and how it works to find match sets of the two inputs. Then it moves on to extend the algorithm to work in any number of inputs or any other parameters passed to it and finally proceeds to extend and tweak it so that it runs as fast as possible for the given circumstances. Many roots are taken to succeed in improving the algorithm like some low-level programming for speeding up little things. Moreover the proposed versions of data structures (ternary trees) were used, which can cope well with the kind of inputs that Learning Classifier Systems use.

Finally, section 4 gives some experiments and their results so that a more clear view of what was implemented and achieved by this project is visible. Existing Java data structures are used for section 2 whilst new tree structures are proposed in section 4 and used in the testing and experimenting phase in section 5. Comparisons between

how existing systems work and how the new proposed systems work are taking place comparing running times in relation with population time, population time vs hash probabilities and how arrays perform in relation with linked list and the ternary tree. Performance is the only criterion so other considerations such as memory consumption will be ignored if differences are not significant.

To give a more clear view of what will follow, the batch-matching problem is split into two main parts. The first one is how the actual matching is going to take place. There are many ways to do this, either by comparing strings and characters, or by using bit representations or even by creating new Enum types in Java. This can be improved by finding which of those works faster and by performing a number of other tweaks like not evaluating against the # symbol since everything is a match with that. The second part is concerned with data structures. A simple way to search for an input-rule match is to perform a linear search from the arrays were they are stored. But of course this is not efficient enough so better implementations of trees will be used or even created in order to be able to match only the relevant parts of them and not the whole structure.

The final part, which is the conclusion, mentions how successful this project is considered and what could be done differently. Also a brief description of what could follow can be found there.

2 Background/History

2.1 Relevant Terms

2.1.1 Machine Learning

For machines, learning is whatever changes data, structure or program depending on the inputs or any external information in such a way that the future performance improves. An example could be considered an addition of a rule to the database. Even if this example does not clearly justify the “learning”, after some inputs in the database when the machine hears some examples of speech we can see an important improvement in speech-recognition [4]. In that context we can feel that the machine has learned taking some input from the environment it is in.

Usually machine learning refers to the changes in the systems that perform tasks associated with Artificial Intelligence (AI). Examples of those tasks might be diagnosis, recognition, planning etc. Machine Learning will be referred to with the abbreviation ML.

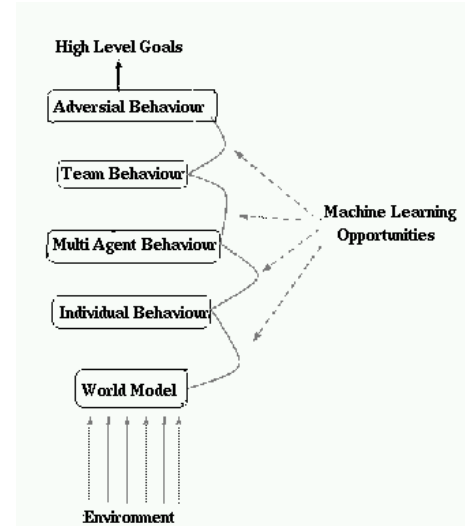


Figure 1 - The principle of layered learning.
Source: Ref. [33]

2.1.2 Information vs Data Retrieval

In the context of information retrieval (IR) system, data retrieval determines which documents out of a collection consist the keywords that the user is asking for (queries). But this is frequently not enough to satisfy the information need since an IR system is worried about retrieving information rather than retrieving data for a given query. A data retrieval language has a purpose to get all objects that clearly satisfy conditions such those in relational algebra or regular expressions. Therefore, in a data retrieval model a single erroneous object amongst many valid is considered to be a total failure but in an IR system the results are allowed to be inaccurate and have small errors because they are likely to remain unnoticed. The reason for this difference is that data is usually clearly define and structured but on the other hand information can be semantically ambiguous [6].

2.1.3 Pattern Matching / Recognition

Pattern recognition is the scientific discipline whose goal is the classification of objects into a number of categories or classes. Some examples can be signal waveforms, images or any other measurements that need to be classified [5]. These objects can also be referred to as patterns. Pattern matching is the act of checking if the elements of a given pattern exist. In contrast to pattern recognition, the pattern is firmly specified. This pattern is about either tree structures or sequences. Pattern matching is used to test if things have a desired structure, to recover the aligning parts, to locate relevant structure and to exchange the matching part with something else.

There are different versions of matching problems appearing in papers. Sometimes we have a static set of rules and we want to match them against inputs; but usually (for example in UCS) the rule population changes over time by adding or deleting to it. Therefore in this project the attempts will be focused on the specific problem described in the introduction since what data structure is better depends on the version of the matching you are trying to solve.

2.1.4 Learning Classifier System

A LCS holds a rule base that is the population of classifiers, which represent the current solution. At every step the LCS gets a problem that requires a decision; what action to do next. Then it matches the incoming instance against the rule based and finds the match set containing the rules that apply for the specific situation. From that match set the action to be executed is decided by the system. By executing the decided action the system receives a numerical reward that is then distributed to the rules responsible for it in order to improve the estimates of the action values. In the meanwhile an algorithm is applied to the rule base so that it finds better rules that will improve the current solution [13].

2.2 Batch matching algorithm

2.2.1 Kovacs's algorithm

The standard matching process is to go through the whole list of conditions (rules) and check each one against the given input to find matches. This is called a linear scan and it performs an exhaustive search, which of course is very inefficient. Therefore the basic question in Kovacs's mind was: "Is there a way to find all conditions [M] that match an input without checking every single condition?"

Let's say that [P] is the set that contains all the rules of a scenario. Now suppose we match 2 inputs at a time (e.g. 2 agents sharing 1 rule-base in a discrete time simulation). By matching input $i1$ it can help match $i2$ if they are similar.

- Intersection Set (IS): set of indexes for which $i1 = i2$
- Disjunction Set (DS): set of indexes for which $i1 \neq i2$

In case that IS is not empty (there are same elements in same positions), then batch matching can speed matching $i2$ because there is no need to match IS twice. Let [Mx] be a set of rules matching input number x and [Mx+] denote a superset of [Mx]. While we are in the process of finding [M1] we are saving rules which match IS in [M2+]. This happens because we know that the elements in the specific index are the same for $i1$ and $i2$ therefore all conditions that will match $i2$ will be in [M2+] by the end of this part. Later on instead of searching inside [P] to find the [M2] we can search in [M2+] which is possibly a match narrower space.

Pseudo code:

Batch matching 2 inputs batchMatch2(i1, i2, r)

```
[M1] = null
[M2] = null
[M2+] = null
computeISDS(i1, i2) -> IS, DS
for each rule(r) in [P]
    findM1AndM2+(i1, r)
for each rule(r) in [M2+]
    findM2(i2, r)
```

charMismatches(inputChar, ruleChar)

```
if inputChar not ruleChar or # then
    return true
return false
```

findM1andM2+(i1, r)

```
for each index i in IS
    if charMismatches(i1i, ri) then
        return
add r to [M2+]
for each index i in DS
    if charMismatches(i1i, ri) then
        return
add r to [M1]
```

findM2(i2, r)

```
for each index i in DS
    if charMismatches(i2i, ri) then
        return
add r to [M2]
```

Figure 2 - The pseudocode was taken from a PDF that Tim Kovacs wrote

2.2.2 Pre-computed match sets

In some cases the inputs (e.g 010, 011 etc) and the rules (e.g. 1##, 0#0 etc) are acknowledged from the beginning and do not change (are fixed). In order to march the inputs against the rules, a set is created let's name in [M]. The algorithm in should add in [M] any rule that matches a specific input. If the input is 010 from the example above, the algorithm will add 0#0 in [M]. Then it will continue to the next input element and forget (override) what has been used for the first (this will happen in every step). This method is called on the fly computation and it is mostly inefficient due to the fact that it needs to be recomputed any matching sets that previously appeared. By using pre-computed match sets the set rules that matches each input is saved in memory (e.g in a map table) and when an identical input reappears, there is no need to re-compute the set since we already have it. This way more memory is consumed in exchange for a speedier algorithm. These maps are called rule maps because they contain all the rules of each input. A good way to implement them in Java is to use a HashMap that stores <String, List>. The string is going to be the input (key) and the List is going to be a list of all rules that match that input (value).

2.3 Encoding schemes

Similar problems have been tackled previously to try and improve running times and memory efficiency of similar algorithms. Since the publication of the work by Wilson in 1995 XCS [7] has been widely adopted by researchers working in Michigan-style learning classifier systems. XCS builds on the work of [8, 9] and Wilson's first classifier system (CS-1) that was later revised and improved by Goldberg [10]. These systems use the ternary language {0, 1, #} or ternary alphabet as they refer to it to describe conditions given a binary input. Most use character-based encoding (represent 0, 1 and # with an 8 bit character in the programming language they use) to describe their conditions because it is easy to implement but proved not to be memory efficient and it is also expensive to compute when the problems size increases. Specifically 75% of the character allocate memory is not used.

2.3.1 Bit Encoding

Llora and Sastry [2] attempted to implement efficient condition encodings and fast rule matching using vector instructions. Vector instructions sets were removed from the CPU market for some time but multimedia and scientific applications drove manufacturers back to including support for them. Implementation using vectors may require GPU usage and many extra hours of implementation. Using this implementations they explored how to reduce the memory footprint of XCS rules. Their findings may prove to work on this paper's problem as well so comparison will take place to determine that and choose whether to proceed with standard character based approach or reduce the rule encoding since characters usually use 8 bits of memory but only 2 bits are required to represent {0, 1, #}. This means that 8-bits encoding wastes 75% of the memory provided thus the memory required to store a populations of rules is

$$\mathcal{M}([P]) = \ell_r \cdot \ell_c \cdot \text{sizeof}(\text{character})$$

Equation 1

[P] is the set of rules
 lr is the number of rules
 lc is the number of conditions

Because the encoding will be the basis for the matching of the conditions and we need that to be as fast as possible and because as mentioned before only 2 bits are required for the encoding of the ternary language, the encoding going to be used is the one proposed in [11] and used by De Jong and Spears [12]. The encoding mapping is as follows:

<i>bit₁</i>	<i>bit₀</i>	<i>ternary value</i>
0	0	#
0	1	0
1	0	1
1	1	#

Table 1. Source: Ref. [2]

Since # exists 2 times in this representation (2 bits give us 4 possible values but we only need 3 for ternary language) the bit combination of 00 was never used for efficiency purposes.

2.3.2 Specificity-based Encoding and Matching

Based on the work of Llorca and Sastry [2], in paper [13] by Butz, Lanzi, Llorca and Laiacomo created their own encoding called specificity, which they claim that it becomes faster and faster as population generality increases (something that does not happen in character-based encoding).

```
// representation of classifier condition
string condition;

// representation of classifier inputs
string inputs;

// matching procedure
int pos = 0;
bool result = true;

while ( (result) && (pos<condition.size()) )
{
    result = ((condition[pos]=='#') ||
              (condition[pos]==inputs[pos]));
    pos++;
}
return result;
```

Figure 3 - Character-based encoding and matching. Source: Ref. [13]

In character-based encoding there is a scan of the whole condition, but checking the wildcard symbol (#) is unnecessary since there is going to be a match anyway as presented in Figure 3. Therefore the matching should be limited to the specific bits and condition that can be represented as a list of position-digit pairs $\langle p_i, d_i \rangle$ that specify which bits should be matched. A pair $\langle p_i, d_i \rangle$ shows that the input position p_i should be equal to d_i . For example the condition $\#1\#0\#$ is represented by the set $\{\langle 2, 1 \rangle, \langle 5, 0 \rangle\}$. In other words, the elements in the positions not mentioned in the

set are going to be # and the rest are going to be whatever the set specifies. By using this notation and therefore limiting the condition representation to the specific bits, less memory is required and the computational cost is lessened. If the average specificity of the rules is σ in the population [14], the cost matching is on average $O(\sigma n)$ for a single rule and $O(\sigma Nn)$ when N population of classifiers is considered.

Let's take s_c as the number of bytes required for storing a character and s_i as the number of bytes required for storing an integer. Then the memory required for this representation is:

$$O(\sigma n N (s_c + s_i))$$

Equation 2

which makes the representation useful when:

$$\sigma \leq \frac{s_c}{s_c + s_i}$$

Equation 3

e.g. in many C compilers $s_c = 1$ and $s_i = 2$ and specificity-based encoding is helpful when the conditions have a specificity lower than 33% (calculated from Equation 3).

The writers continued providing an even simpler representation of that idea. Condition c is now represented as a set (array) of points $\{c_1, c_2, \dots, c_n\}$ that includes both digit and position but they are both encoded. For example $p_i = \frac{c_i}{2}$ and $d_i = c_i \bmod 2$. In an example $\{5, 10\}$ which represents $\{<2, 1>, <5, 0>\}$ of the previous notation we have:

$$\begin{array}{ccc}
 & \{5, 10\} & \\
 \swarrow & & \searrow \\
 \frac{5}{2} = 2 & & \frac{10}{2} = 5 \\
 5 \bmod 2 = 1 & & 10 \bmod 2 = 0
 \end{array}$$

Equation 4

The new implementation is very compact and requires $O(\sigma n N s_i)$ bytes to store the entire population which is less than half (50%) of the character-based representation. Figure 4 shows an implementation of this version.

```

// representation of classifier condition
vector<int> condition_vector;

// representation of classifier inputs
string      inputs;

...

//
// matching procedure
//
int  pos = 0;
bool result = true;

// match ends when a pos did not match or
// all the integers have been considered
while (result && pos<condition_vector.size())
{
    result = inputs[condition_vector[pos]/2] ==
              '0'+condition_vector[pos]%2;
    pos++;
}
return result;

```

Figure 4 – Specificity-based encoding and matching. Source: Ref. [13]

N	gen	char($\mu \pm \sigma$)	spec($\mu \pm \sigma$)	bin ($\mu \pm \sigma$)
1000	0.0	0.11 \pm 0.02	0.12 \pm 0.03	0.08 \pm 0.02
1000	0.25	0.13 \pm 0.03	0.11 \pm 0.03	0.07 \pm 0.02
1000	0.5	0.14 \pm 0.03	0.12 \pm 0.02	0.08 \pm 0.03
1000	0.75	0.15 \pm 0.03	0.12 \pm 0.03	0.08 \pm 0.02
1000	0.99	0.17 \pm 0.03	0.09 \pm 0.02	0.08 \pm 0.02
5000	0.0	0.74 \pm 0.07	1.14 \pm 0.09	0.41 \pm 0.05
5000	0.25	0.72 \pm 0.06	1.13 \pm 0.09	0.44 \pm 0.06
5000	0.5	0.82 \pm 0.07	1.11 \pm 0.08	0.43 \pm 0.08
5000	0.75	0.87 \pm 0.08	0.94 \pm 0.09	0.40 \pm 0.05
5000	0.99	0.90 \pm 0.08	0.55 \pm 0.07	0.44 \pm 0.07
10000	0.0	1.69 \pm 0.12	2.80 \pm 0.11	0.87 \pm 0.09
10000	0.25	1.77 \pm 0.12	2.83 \pm 0.12	0.88 \pm 0.09
10000	0.5	1.89 \pm 0.1	2.73 \pm 0.13	0.89 \pm 0.07
10000	0.75	1.94 \pm 0.09	2.13 \pm 0.14	0.81 \pm 0.08
10000	0.99	1.90 \pm 0.1	1.53 \pm 0.12	0.87 \pm 0.08

Table 2. Source: Ref. [13]

Even though specificity-based encoding requires less memory it still might not be the faster one in respect to execution time. For this purpose in [14] some experiments were performed following the design of [2]. A set N was generated with ternary conditions of length n with different generality and 2000 random inputs. Each random input was matched against the N conditions using character-based, specificity-based and binary-based encodings and the average CPU time required to perform all match operations was recorded. This procedure took place 50 times and the average of those measurements is presented in table 2. Size N was changed from 1000 to 5000 and finally 10000 and generality had the values 0.00, 0.25, 0.50, 0.75 and 0.99. As noted before (and now proved by this table) the specificity-based encoding works faster as generality increases and character-based encoding works slower. But still the binary-encoding approach is faster and not affected by the generality.

2.3.3 BitSet in Java

This class in Java implements a vector of bits that grows as needed. The components of the bit set have a boolean value. This means they can be either true or false. Moreover, the bits of a BitSet are indexed by nonnegative integers and individual indexed bits can be examined, set, or cleared just like any other data structure found in Java. A BitSet's contents may be changed by another BitSet's contents by using

AND, OR and XOR operations. The default value of all bits is initially set to false and the current size of the bit set is the space of bits that it is currently has set. Have in mind that the size of the BitSet is related to the implementation therefore it may change if implementations changes. The length of a bit set relates to logical length of a bit set and is defined independently of implementation [32].

Having this structure in mind, the bit encoding mentioned in 4.1 can be implemented by using BitSet. Although this looks promising and well suited for this purpose I suspect that it will be much slower than simply implementing bitwise operations since only 2 bits are required. Most probably this data structure was created since it can scale up better and would help when the bits are more but tests will take place to confirm or disprove this thought.

2.3.4 Enum in Java

Java Enums are used to create new types. A common example given is in case that a programmer needs to have a type called Month that will contain all the months by their names (e.g. Month.JANUARY). Moving this concept to the scope of this project, what if a new Enum type was create in order to hold the rules and conditions. For example we could have Rules.01# as one of the valid elements of this new Enum. Having in mind that this rules are frequently used since the matching is based upon them, a specialized Set for use with Enum types could be used (EnumSet). All of the elements in an Enum set must come from a single Enum type that is specified, explicitly or implicitly, when the set is created and this would be the rule Enum set. Bit vectors are how Enums are represented internally and is a very efficient and compact representation. The space and time performance of this class should be good enough to allow its use as a high-quality, type-safe alternative to traditional int-based “bit flags”. Operations such as containsAll should run very fast in case the collection specified is also an enum set. Furthermore, iterations though elements of “rules” would be fast since the enhanced for loop of Java is supported:

```
for (Rule r : EnumSet.range(Rule.000, Rule.01#))
```

EnumMap is an implementation of a Map that uses Enum types as keys. All of the keys in an Enum map must also come from a single Enum type as in EnumSets. Arrays are used to represent Enum maps internally. This representation is extremely compact and efficient. Moving this to the batch matching algorithm previously introduced, EnumMap could be used to create maps that their key is the input values. The value then is going to be a list of rules and this will hopefully provide high performance to the algorithm.

2.4 Knowledge-based Evaluation

Nowadays information keeps increasing and the need of finding efficient techniques to improve data mining is rising. Knowledge-based techniques to make information recovery faster are discovered and studied thoroughly so that better solutions will come up and existing solutions will be improved.

Evolutionary computation techniques are used to tackle problems that have large search spaces and exhaustive search will simply take too long to finish. Because of this Genetic Algorithms (GAs) and/or Evolutionary Algorithms (EAs) have been used to solve machine learning and optimization tasks [15]. Attempts have been made to drive individual solutions to better solutions by integrating some domain knowledge in evolutionary algorithms.

2.4.1 Knowledge Incorporation in EAs

Knowledge incorporation in EAs can be categorized into two sections. The first category is the one that uses techniques that extract the domain knowledge before starting to search for solutions (Evolutionary Algorithms with A Priori Knowledge Extraction). The second one is the one that knowledge is extracted throughout the whole evolutionary processes (Evolutionary Algorithms with Dynamic Knowledge Extraction). Furthermore, EAs can also be categorized into other two categories that depend on when the extracted knowledge is used. There is Adapted EAs where extracted knowledge is used before starting the process and Adaptive EAs where extracted knowledge is adapted during the execution.

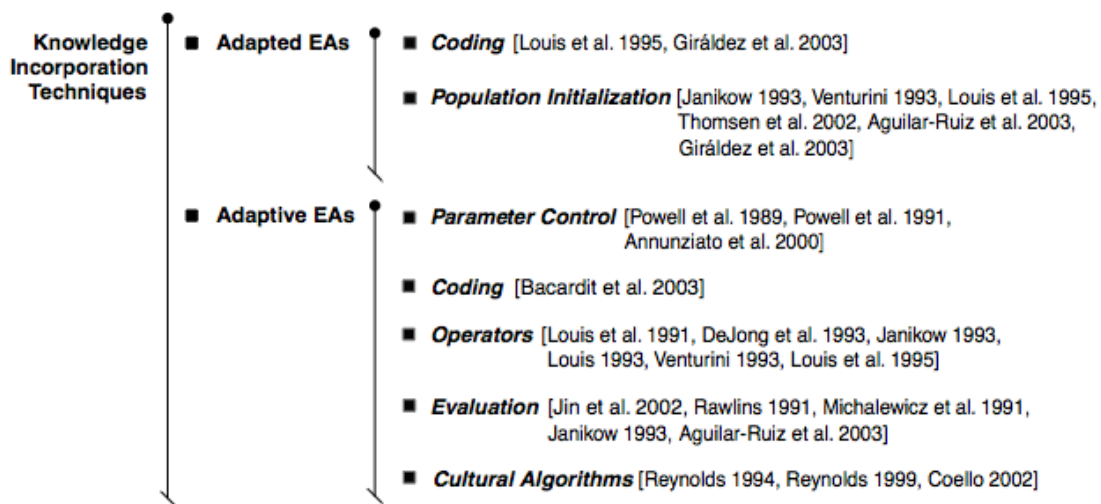


Figure 5 – Categorization of knowledge incorporation into EA techniques. Source: Ref. [16]

2.4.2 Decision Rules

The evaluation of some rules using a specific strategy that adds new knowledge in the evaluation process in Evolutionary Learning is discussed in [16]. A knowledge model can be represented using various structures (decision trees, decision rules etc).

Decision rules are the conditions that the example must fulfill in order to be classified. If attributes of an example match the conditions of the rule, this means that the rule covers that example.

If $Cond_1$ and $Cond_2$ and...and $Cond_M$ then $Class = C$

Figure 6 – A decision rule

In Figure 6 there is an example of a decision rule where $Cond_i$ is the conditions that the i^{th} attribute of an example has to satisfy so it will be classified with class C and M is the total attributes in the dataset.

Usually the EA learning methods evaluate the rules directly from the dataset. The dataset is searched through sequentially taking the examples and testing the rules on them. This means that the learning processes can be very time consuming and not efficient in terms of space and time. Some have tried to improve the learning methods so that computational costs are reduced [17] [18]. Some others tried a different approach using scalability [19]. The rules can be matched against the conditions by using various techniques. The features that influence the performance and efficiency are two; encoding that was discussed previously in this paper and evaluation that is the external function that determines how good some potential solutions are.

Giraldez, Aguilar-Ruiz and Riquelme in [16] attempted to design a new data structure that incorporates knowledge on the example distribution in the attribute space. They called their new data structure Efficient Evaluation Structure (EES) and it organizes information from dataset so that it is not required to check all examples in order to evaluate the decision rules of a supervised learning system. This new data structure will be discussed in the next section.

2.5 Data Structures

2.5.1 Java Collections

Java offers various data structures and many of them have approximately the same functionality. Same functionality does not mean same runtimes though. Runtime efficiency depends a great deal on the kind of data and how that data is going to be used. A Collection in java is a general-purpose object that is used to hold and manipulate other objects (strings). One type of collection is the ArrayList. Usually in Java those collections have two-word names, e.g. Array-List. The second word is used to specify how they are implemented (List, Set etc) and the first name specifies the interface (Array, Linked etc) [20] [21].

Java offers the Java Collections Framework (JCF) is a coupled set of classes and interfaces that implement commonly reusable collection data structures. It was primarily developed and designed by Joshua Bloch. It helps programmers by providing readymade solutions.

2.5.1.1 JCF Interfaces

An interface in Java is an abstract type that is used to specify an interface that the classes should implement. Interface keyword is used to declare interfaces, which can only contain

variable declarations and method signatures. Method definitions cannot be included in an interface. JCF interfaces of modules are a set of public methods and the class must implement all of the methods described in the interface. An advantage that interfaces have is their ability to simulate multiple inheritance.

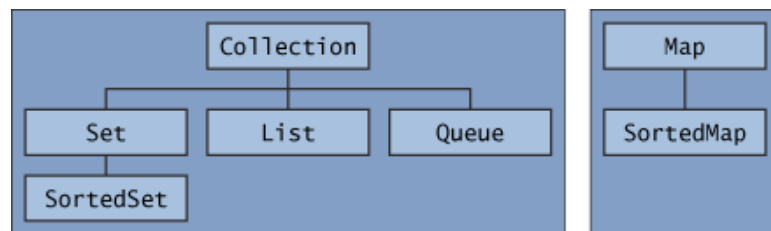


Figure 7 – some JCF Interfaces. Source: Sun Java API website

Collections are used like this:

Collection<String>, Collection<Integer> etc

As shown in Figure 7 collections come in four categories:

- Lists of things ordered, duplicates allowed, with an index.
- Sets of things may or may not be ordered and/or sorted; duplicates not allowed.
- Maps of things with keys may or may not be ordered and/or sorted; duplicate keys are not allowed.
- Queues of things to process ordered by FIFO or by priority.

There are also sub-categories:

- Sorted (Order in the collection is determined according to some rule)
- Unsorted

- Ordered (When a collection is ordered, it means you can iterate through the collection in a specific (not-random) order)
- Unordered

2.5.1.2 Collections Features

Most frequent interfaces	General purpose implementations
Collection	HashMap
Set	HashSet
SortedSet	ArrayList
List	PriorityQueue
Map	Collections
SortedMap	HashTable
Queue	LinkedHashSet
	Vector
	Arrays
	TreeMap
	TreeSet
	LinkedList
	LinkedHashMap

Table 2

Note that the Map-related classes and interfaces do not extend from Collection. Even if thought of as collections, none of the following actually extends Collection:

- SortedMap, Hashtable, HashMap, TreeMap, LinkedHashMap.

One-word descriptions:

- List: Cares about the index. They are ordered (not sorted). Methods related to the index unlike other collection types.
- ArrayList: Fast iteration and fast random access.
- Vector: It's like a slower ArrayList, but it has synchronized methods.
- LinkedList: Elements are doubly-linked to one another. Suitable for adding and removing elements at the end. Good (e.g. queues and stacks).
- Set: A Set cares about uniqueness—it doesn't allow duplicates.
- HashSet: Fast access, assures no duplicates, provides no ordering.
- LinkedHashSet: No duplicates; iterates by insertion order.
- TreeSet: No duplicates; iterates in sorted order. Implements SortedSet.
- Map: Cares about unique identifiers. Has key/value pairs like a phone book directory.
- HashMap: unsorted, unordered Map. Updates fast and does not allow duplicates in key (only one null). Duplicates are allowed for values.
- HashTable: Like a slower HashMap (as with Vector, due to its synchronized methods). Null keys and values are not allowed.
- LinkedHashMap: Faster iterations; iterates by insertion order or last accessed; allows one null key, many null values.
- TreeMap: A sorted map. Implements SortedMap.

- Queue: Although other orders are possible, queues are typically thought of as FIFO (first-in, first-out).
- PriorityQueue: A to-do list ordered by the element's priority.

The reason for this part in the report is to indicate that there is a suitable collection type depending on what data you have and what you are trying to do. For example you must choose an ArrayList over a LinkedList when you need random access since you can go directly to the index position you specify. On the other hand you must choose LinkedList over ArrayList when you are going to execute many insertions and deletions because arrays have to copy whole chunks of memory in order to do that which makes runtimes much slower.

2.5.2 Problem Specific Data Structures

There exist many different kinds of data structures each targeting a specific problem in order to speed up the access of information. Multidimensional access methods (MAM) are used when the information is spread in multidimensional spaces and some examples are: Point to access methods (KDB-tree [22], LSD-tree [23], BV-tree [24] etc) and Spatial access methods (K-D-tree [25], R-tree [26] etc).

All those variations of trees were not really helpful in giving a good solution to the problem in [16] because they index a dataset having as only goal to speed up queries on such data. But in this project speed is the main goal, so some may be considered.

2.5.2.1 Linear Evaluation

EA encode possible decision rules individually in the genetic population. After the first population has been created each entry is evaluated based on its quality applying the crossover mutation and therefore creating the new population. It is clear that the number of evaluations that have to be executed is constant since each individual has to be evaluated for each population. For example, an EA with 300 generations, each having 100 individuals needs 30000 evaluations. Now, in case the dataset has 1000 examples then the evaluations become 30 million! Consequently, the computational cost is $O(NM)$ (N is the population of examples and M is the population of attributes). As previously mentioned, the evaluation may need 65%-85% of all the time required by the calculations which is a huge amount of time. This can be improved either by designing a new data structure that will not need to evaluate every entry of the dataset (this is what EES does and will be discussed in the next section) or by building models to estimate efficiently the individual fitness evaluation.

2.5.2.2 Efficient Evaluation Structure (EES)

EES [16] has as aim to distribute in space all the information in such a way that is spread by attribute rather than by example. If the attributes of the set are not discrete, first they must be transformed to discrete and then stored in this data structure so that they can be handled appropriately.

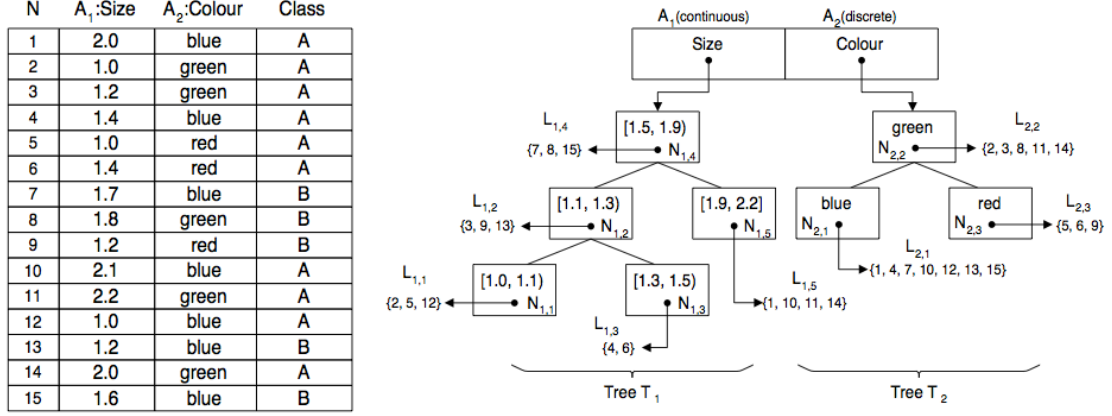


Figure 8 – EES of two attributes (continues and discrete). Source: Ref. [16]

As shown in Figure 8, EES stores the values of each attribute independently. The most interesting values are the ones that define the geometrical boundaries amongst the labels. In case the examples are sorted by SIZE, we can see that only 1.1, 1.3, 1.5 and 1.9 need to be investigated since these are the values that produce change of class.

The EES has a node with interval $[1.0, 1.1)$ for attribute size: 1.1 being the mean value of 1.2 and 1.0 so therefore change of label takes place. This is very helpful in mutation for real valued attributes since any value in the range of 1.0 and 2.2 may be generated. Now the number of possible mutations is limited by the boundary limits.

For every attribute A_i , the finite set of attributes it can take is denoted by Ω_i .

- If A_i is a discrete attribute, Ω_i will contain values that are represented by $V_{ij} (1 \leq j \leq |\Omega_i|)$.
- If A_i deals with continued attributes, Ω_i is represented as $I_{ij} (1 \leq j \leq |\Omega_i|)$ and the lower bound is l_{ij} and the upper bound u_{ij} .

The EES arranges information so that i^{th} element of the vector will have information about the i^{th} attribute (A_i) in the dataset. Different intervals or values that A_i can take are sorted in one tree that is denoted by T_i . Each node also contains a list L_{ij} of numbers that show the positions of examples in the dataset.

- If A_i is discrete, L_{ij} indexes will correspond to the examples whose i^{th} attribute take the j^{th} value (V_{ij}) within Ω_i .
- If A_i is continuous, L_{ij} indexes will correspond to those examples whose values for the i^{th} attribute are included in the j^{th} interval (I_{ij}) within Ω_i .

In the tree example of Figure 8 the tree is sorted by disjoint intervals and in case of discrete values it is sorted alphabetically. This way any search will have logarithmic cost.

EES testing shows about 50% improvement in performance:

BD	A.T. ESS	A.T. Vector	Improv.(%)
breast cancer (Wisc.)	5.572	13.421	58.5
bupa liver disorder	1.464	4.370	66.5
cars	2.173	4.870	55.4
cleveland	3.460	6.042	42.7
glass	1.634	2.884	43.3
hayes-roth	0.720	1.499	52.0
heart disease	3.018	6.228	51.5
iris	0.517	1.536	66.3
led7	20.689	34.905	40.7
letter	222.556	842.784	73.6
pima indian	4.283	11.309	62.1
soybean	1.661	2.808	40.8
tic-tac-toe	7.623	11.031	30.9
vehicle	8.937	18.785	52.4
wine	2.084	4.120	49.4

Table 4 – Comparison of average results of the EES and the liner vector. Source: Ref. [16]

BD	Linear Eval.		EES Eval.	
	ER	NR	ER	NR
breast cancer (Wisc.)	4.3	2.6	4.0	2.0
bupa	35.7	11.3	35.2	3.8
cars	14.3	16.2	11.9	8.1
cleveland	20.5	7.9	24.6	4.9
glass	29.4	19.0	33.8	9.8
hayes-roth	21.3	8.4	20.7	7.1
heart disease	22.3	9.2	21.8	4.3
iris	3.3	4.8	3.3	3.2
led7	27.9	41.9	27.0	42.3
letter	11.7	235.8	10.1	97.2
pima indian	25.9	16.6	25.6	5.1
soybean	11.7	47.2	1.8	4.0
tic-tac-toe	3.9	11.9	5.2	11.8
vehicle	30.6	36.2	34.2	16.9
wine	3.9	3.3	8.8	5.6

Table 5 – Comparison of average results of ErrorRate and NumberOfRules. Source: Ref. [16]

2.5.2.3 R-trees

Another interesting presentation of a data structure that is worth discussing is the R-tree that was first presented in [26]. The aim was to create a tree suitable and efficient for retrieving objects according to their spatial location. Often in spatial data and computer aided design it is required to find all objects that lie within some distance of a given point. Before the R-tree was introduced, there was no efficient way to do this since classical single-dimensional indexing structures are not appropriate for multi-dimensional spatial searching. Additionally collections like hash tables are not suitable either because matching works against one exact value where now there is a need for a range search.

A number of various structures have been proposed over the past (some of them are mentions earlier in this report) in order to handle multi-dimensional point data. A survey of those can be found in [27]. There is a number of reasons that all those proposals are not good enough for multi-dimensional point data some of which require the cell boundaries to be set in advance [28, 29], others do not take paging of secondary memory into account (Quad trees [30]) and so on. Note that these data structures, mostly tree types, have been around for over 2 decades therefore it is common to see them working in some applications where they were considered suitable.

R-tree is a height-balanced tree alike B-tree [2] with index records in its leaf nodes containing pointers to data objects. The nodes correspond to disk pages (if the index is disk-resident) and the tree is designed in such a way that a spatial search only requires visiting a small number of nodes. Moreover, the tree is completely dynamic therefore deletions and insertions can be used in any order between searches and not affect execution times.

The spatial database consists of a collection of tuples that represent the objects. Each of them has a unique ID in order to be able to retrieve the object. Leaf nodes in the tree have the form $(I, \text{tuple-identifier})$ and *tuple-identifier* refers to a tuple in the database and *I* is an n-dimensional rectangle that indicates the boundaries of the

spatial object ($I = (I_0, I_1, \dots, I_{n-1})$). I_i is a closed bounded interval $[a, b]$ and n is the number of dimensions. I_i can have both or just one endpoints equal to infinity showing that the object extends indefinitely. Node with no leafs have the form $(I, \text{child-pointer})$ where *child-pointer* is the address of a lower node in R-tree and I covers all rectangles in the lower's node's entries.

The best way to understand how the R-tree works is by example:

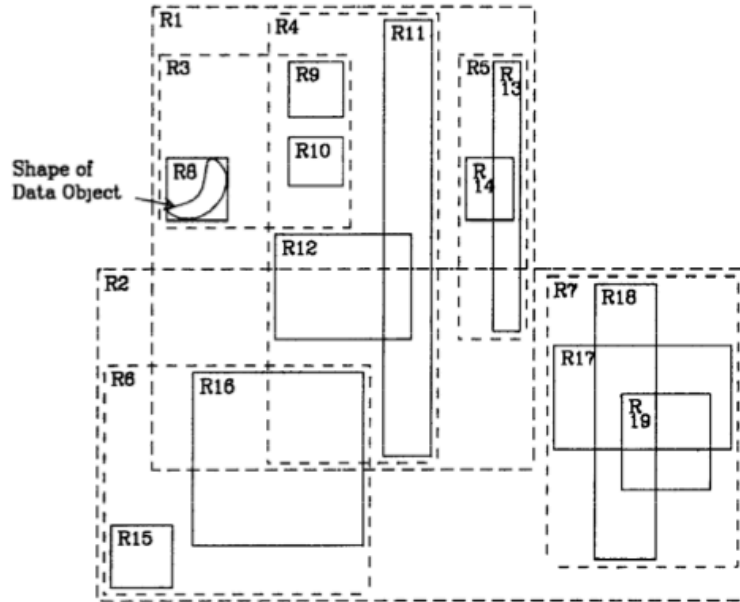
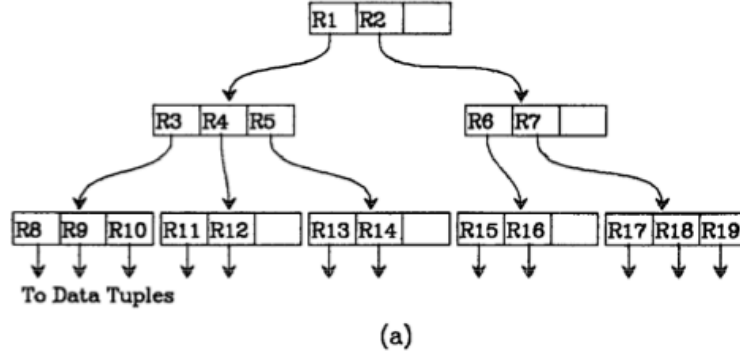


Figure 9 – Containment and overlapping relationships between rectangles. Source: Ref. [26]

If M is the maximum number of entries that fit in a node and $m \leq \frac{M}{2}$ be the minimum number of entries in a node, the height of the tree containing N index records is at most $\lceil \log_m N \rceil - 1$ since the branching factor of each node is at least m .

3 Project Design & Implementation

3.1 Problem

Pattern matching is the most time consuming part of some Learning Classifier Systems. As mentioned before, classifier systems are software that given the inputs and labels (outputs) they can compute how those labels were produced based on those inputs. Taking as an example the sUpervised Classifier System (UCS) introduced by Ester Bernado-Mansilla and Josep M. Garrell-Guiuand in [34] and implemented by Dr Gavin Brown and expanded by Dr Tim Kovacs [35], amongst the many operations that are taking place during its evaluation process, the one that takes the longest time to complete is the matching process between inputs and rules.

Inputs are binary strings, of a specified length (e.g. 10011) and rules or sometimes referred to as conditions are ternary strings. Rules may contain 0, 1 or #. Hashes (#) are the “don’t care” symbol that means that it matches both 0s and 1s. This is also called wildcard in some cases. A rule example that matches the previous input is 10##1.

The normal matching process involves going through each element of the input and checking it against each element of the rule. For this scenario it would require 5 comparisons since the length of the inputs and rules is five. Now image that there are thousands of inputs, thousands of rules and each of them is about a hundred characters long. Since the goal is to find which of the rules match every input (match set of every input) the number of comparisons becomes huge making the algorithm run very slow. The main aim of this project is to solve these delays by providing more efficient algorithms or data structures that will reduce comparisons, and execution times. This is a research project since almost none of the work has been implemented before therefore the results cannot be clear prior to the experiment phase. Experiment results of the most important aspects are located in section 4.

3.2 Initial Idea

The initial idea is mentioned in section 2.2.1. The aim behind it is not to make the same calculations more than once but to make more calculations on a single pass in order to find the match sets of more than one input (or at least find a superset of the match sets).

A simple example of the **Batch Matching**:

For this example we have 2 inputs and 3 rules. The total number of inputs is sometimes referred to as the *match instances* and the total number of rules as the *population size*. Pseudocode for the functions that will follow is mentioned in the example located in section 2.2.1.

Inputs	Rules [P]
00	1#
01	01
	#0

Instead of going through each element/character of the inputs and comparing it with each element of the rules (Naïve matching), why check the first element of the two inputs twice since we know it is the same in both inputs? Therefore before starting the matching process we have to check what positions/indexes are common in the two inputs. This is where `computeISDS()` steps in. This function finds the common elements and stores them in a list that is called *Intersection Set* (IS) and the non-common elements are stored in a different list called *Disjunction Set* (DS). After the execution of this method we have:

$$IS = [0] \text{ and } DS = [1]$$

Meaning that elements in index 0 are common and in index 1 are not.

After finding the IS and DS we move to the next step that is to find the match set of the first input. The method that does that is called `findM1andM2+()`. Instead of going through each index of the input, we start by going through the IS indexes and then the DS indexes (which is essentially the same thing). But on the process of going through the IS and before moving on to check the elements of the DS, a new subset is the populations is created called `[M+]`. In there we add all rules that match the first input in the IS positions; this means that it will also match the second input on the same positions. The result after the execution of this method is that we have the match set of input1 and a superset of the match set of input2 (`[M+]`). This superset is usually a smaller set than the initial population, unless the IS was empty or it happened that no rule in population matched the first input.

$$\begin{aligned} \text{Input1 Match Set (M1} &= [\#0]) \\ \text{Input2 super set (M+} &= [01, \#0]) \end{aligned}$$

The basic tradeoff of batch matching is that there are two extra steps that do not happen in Naïve matching:

- A. Computer the IS and DS
- B. Construct `[M+]` to be used instead of the Population
- C. `[M+]` is smaller than `[P]` (population of rules) so we do not have to check as many rules. Specifically, any rule which mismatches IS won't appear in `[M+]`.
- D. Do not have to check all the indexes in `[M+]` since we have already checked IS. Therefore now only check the DS indexes.

The first 2 points are extra work that naïve matching does not do, but the 2 last points save time compared to naïve matching. So if $A+B > C+D$ then batch matching will be faster.

The method that goes through the DS elements of `[M+]` so that it finds the match set of the second input is `findM2()`. After its execution we get:

$$\text{Input2 Match Set (M2} = [01])$$

The goal was to have the match set of each input by the end of the execution and because we have accomplished that, the approach is considered successful. Saving in time and comparisons here may not seem very tremendous but if there are thousands of inputs and rules and each one of the is about a hundred elements long then the difference between Naïve and Batch matching becomes enormous.

Batches of size two should run up to twice as fast as standard matching. Surely this is the theoretical maximum: you process 2 inputs in the same time it takes to do one. This would happen if the 2 inputs were identical therefore the superset of the match set of the second input ($[M+]$) would contain exactly the same elements as $[M2]$ should. Also the DS will be empty (inputs are the same) and it would not need to make any more checks inside `findM2()`. So a theoretical maximum speedup for batches of size N should be close to N times.

3.3 Batch Matching

3.3.1 Batch Matching Size N

The Batch Matching algorithm is no good in a scenario where the inputs are more than 2. The version of the batch matching I have just given in the previous section is only good for length 2. Therefore the next step was to find a way to convert it to a more general approach so that it can cope with arbitrary inputs.

3.3.1.1 Adjacent Batches

Example of adjacent matching with batch size 2:

```
Input[0] = 00
Input[1] = 01
Population = [00, 01, 0#, 10, 11, 1#, #0, #1, ##]
IS = index 0
DS = index 1 (IS and DS are computed in computeISDS())
M0 = [00, 0#, #0, ##] (set of rules matching input[0])
M1+ = [00, 01, 0#, #0, #1, ##] (rules matching indexes in IS, no
matching of DS indexes yet. i.e. of the form 0*)
([M0] and [M1+] are both computed in the function findM1AndM2Plus())
M1 = [01, 0#, #1, ##] (set of rules matching input[1])
(M1 is computed in findM2())
```

Everything that is included in parenthesis is either comments or guidelines.

How it works:

First step was to repeat the simple batch matching for every two adjacent inputs until it reaches to an end. The way to do this was to first compute the IS and DS for all pairs, then compute the match set $[M1]$ of every first input of a pair along with the superset of each second input $[M2+]$. Last step was to go through each $[M2+]$ and search only the elements in the equivalent DS of every second input. To explain a bit better, $[IS]$, $[DS]$ and $[M2+]$ are all ArrayLists of ArrayLists in order to be able to store everything. The figure below shows the important method calls in the sequence they are called.

```
//computes IS-DS for ALL inputs
computeISDS(input);

//initialize match sets and super sets for each second input
for (int i=0; i<input.size(); i++) {
    //create one Mplus for each second input (pair)
    if (i % 2 == 1)
        Mplus.add(new ArrayList<String>());
    M.add(new ArrayList<String>());
}

for (String r : P)
```



```

        findM1AndM2Plus(input, r); //for ALL inputs

    for (int i=0; i<Mplus.size(); i++) { //go through all Mplus (pairs)
        for (String r : Mplus.get(i))
            findM2(input, i, r);
    }

```

Figure 10 – Taken from *batch-matching/BatchSizeN/Population.java*

Of course some tweaks had to be made to findM2() method in order for it to know which input to compare with which super set and against which DS positions.

3.3.1.2 Distant Batches

A second approach of batch matching for any size of inputs (Size N) was to make something similar to the first approach that was taking pairs but instead of taking pairs take distant batch sets. The most straightforward way to do this is to compare each input against the first one and make similar calculations for finding their match sets. For example, compare input[0] with input[1] and then again input[0] with input[2] and so on.

Example of distant matching with 3 inputs batch size 2:

```

Input[0]
Input[1]
Input[2]
Population = [000, 010, 0#0, 100, 110, 1#0, #00, #10, ##0]
IS for inputs [0] and [1] = index 0, 2
DS for inputs [0] and [1] = index 1
IS for inputs [0] and [2] = index 1, 2
DS for inputs [0] and [2] = index 0
(rules matching IS of inputs [0] and [1]. i.e. rules of the form
0*0, in other words 0, then either 0 or 1, then 0)
M1+ = [000, 010, 0#0, #00, #10, ##0]
(rules matching IS of inputs [0] and [2]. i.e. rules of the form
*00)
M2+ = [000, 0#0, 100, 1#0, #00, ##0]
M1 = [010, 0#0, #10, ##0]
M2 = [100, 1#0, #00, ##0]

```

At this point the changes in code had to be made in three different locations so that the first input always remains the same. Therefore computeISDS(), findM1andM2plus() and findM2() were modified to fit this description. The code that makes this happen is located in *batch-matching/BatchSizeN/Population2.java*.

3.3.1.3 Variable Batch Size

Since the code that was working on any number of inputs, why not take it one step further and instead of always having a batch size 2 you can have a variable batch size. It may happen that in some cases comparing more than 2 inputs at a time produces better results therefore the algorithm was modified so that it can support this.

Example of variable batch size with 3 inputs starting with batch size 3:

This example uses the same inputs and rules as the example above. Now we can compute the joint IS and DS for all 3 inputs instead of separate [M1+] and [M2+] we just have the same [M+] for both.

```

Input[0] = 000
Input[1] = 010

```

```

Input[2] = 100
Population = [000, 010, 0#0, 100, 110, 1#0, #00, #10, ##0]
IS for all inputs = index 2
DS for all inputs = index 0, 1
M0 = [000, 0#0, #00, ##0] (rules matching IS. i.e of the form **0)
(note this is not smaller than the Population but only because all
rules in it happen to end with 0)
M+ = [000, 010, 0#0, 100, 110, 1#0, #00, #10, ##0]
(now we forget about input[0] and do a batch size 2)
IS for input[1] and input[2]: index 2
DS for input[1] and input[2]: indexes 0, 1
M1 = [010, 0#0, #10, ##0]
(rules from [M+] matching indexes in IS, rules of the form **0. This
happens to be all rules in [M+] since they all happen to end with 0)
M2+ = [000, 010, 0#0, 100, 110, 1#0, #00, #10, ##0]
M2 = [100, 1#0, #00, #00, ##0]

```

If there are more than three inputs the process repeats. In other words it puts inputs [0], [1] and [2] in one batch and the inputs [3], [4] and [5] in the next batch and so on.

In order to have a variable batch size a new algorithm was created (*batch-matching/BatchSizeN/Population3.java*). Now the loop inside computeISDS() and findM1andM2plus() was altered from:

```
for (int i=0; i<input.size(); i+=2) {...}
```

Figure 11 – From batch-matching/BatchSizeN/Population.java

to:

```
for (int i=0; i<INP_SIZE; i+=BATCH_SIZE) {...}
```

Figure 12 – From batch-matching/BatchSizeN/Population3.java

so that it can jump from one batch set to the next and perform the same calculations.

At this point a new global variable is introduced called BATCH_SIZE. The only thing that has to change in Population3.java in order for it to run with different batch size is to change the value of that variable and the code takes care of the rest.

There are many chances that the last set of inputs contains less than batch size number of inputs. For this purpose and so that the algorithm does not crash, some new variables and checks were introduced to overcome the problem:

```

int i = pos*BATCH_SIZE;
int tempBATCH_SIZE = BATCH_SIZE;
if (i==LAST_ROUND) {
    if ((LAST_SET_SIZE) == 1) return; //this scenario is dealt with in
    findM1andM1Plus
    tempBATCH_SIZE = LAST_SET_SIZE; //change BATCH SIZE to match the
    number of inputs in the last set
}

```

Figure 13 – findMs() in Population3.java

```

if (i==LAST_ROUND) {
    if ((LAST_SET_SIZE) == 1) { //in case only one input left
        //more code here
    }
}

```

```
//more code here
}
```

Figure 14 - findM1AndM1Plus() in Population3.java

Because these checks have to happen in every iteration (one for every batch size) until all the inputs matching has been completed, it adds overheads and extends the execution time; but it is still faster than the naïve matching!


3.3.1.4 Recursive Batch Matching

A promising idea was to use a “recursive” algorithm starting from a predefined batch size and making it smaller on the go until only one input is left. This would happen for each batch set. For example if the initial batch size is set to 4, then the comparison would take place amongst the first 4 inputs and then computer the match set for the first and superset for the rest three inputs. At this point the batch size would decrement to 3 and instead of using the initial population of all rules it would you the superset that was calculated in the previous step to continue and find match sets for the rest inputs.

Algorithm illustration:

Inputs

001
011
010
000
110
...



Find the IS and DS of the first 4 inputs (IS = [0], DS = [1, 2])
Based on that IS and DS find the set of rules that match the first input and store it in [M1] and the set that match the rest three inputs and store it in [M+]. [M+] basically in this case removes all elements of the population that start with a 1.

Repeat the steps above until the batch size becomes greater than 1 (2).

Figure 15 – explanation of how batch-matching/BatchSizeN/Population5.java works.

To implement the algorithm shown above, some methods had to be modified so that they would accept as an input the size of the batch. Therefore now computeISDS() is defined as computeISDS(int start, int bsize) which tells it at which input to start the batch process and what the batch size is. In the scenario where the batch size equals 4 it gives:

computISDS() call	Starting Input	Batch Size
Call 1	0	4
Call 2	1	3
Call 3	2	2
Call 4	4	4
Call 5	5	3
Call 6	6	2
...

Table 3 – Recursive method calls of computeISDS()

Something to notice here is that the batch size never becomes 1. This is why if an input is single (by itself) there is no “batch” and therefore a different procedure is

followed in order to be matched against the rules. This procedure is the plain Naïve procedure that goes through each element of the input comparing it with each element of the population. But remember that by that stage the population is smaller since it only keeps the subsets of the batch matching process for the previous inputs! In other words, the population size at this point contains only the rules that match in the intersection set (IS) this and the previous inputs of the batch.

3.3.2 Data Structures

After experimenting with the various algorithms of batch matching it was time to check the performance of various built-in data structures of Java. Each one has its own strengths and weaknesses therefore a comparison had to be made. By this stage the new proposed data structure (a version of ternary string) was not implemented yet therefore another comparison is located in section 3.4 that also takes that into consideration.

3.3.2.1 ArrayLists

ArrayLists use arrays for internal storage. This gives us fast random accesses (e.g. get the element #102), since the array index gets you right to the point where the element is located. On the other hand, adding and deleting at the start and middle of the ArrayList are going to be slow, because all the succeeding elements will have to be copied backwards or forwards. (Probably using `System.arraycopy()`).

An ArrayList would also give performance problems at the point where the internal array fills up. Then the ArrayList has to create a new array and move all the elements there. The ArrayList has a growth algorithm of $(n*3)/2+1$, which basically means that when the buffer cannot hold all the data it will create a new one of size $(n*3)/2+1$ where n is the number of elements of the current buffer. Therefore if we can know in advance the size of the array that we are going to use, it makes more sense to create the ArrayList with that capacity in mind during the object creation and using the overloaded constructor so it will not have to grow on the process.

3.3.2.2 LinkedLists

A LinkedList is composed of a sequence of nodes. Each node keeps an element and the pointer to the next node. A singly linked list only has pointers to next and a doubly linked list has a pointer to both the next and the previous element. This makes the process of going backwards through the elements easier.

Linked lists are slow when it comes to random access since getting the element of a predefined position (e.g. element #102) means it has to go through all the elements that are before that in order to get there (or start from the end and move backwards on a doubly linked list). Moving forward or backwards searching for an element most often depends if the position you are looking for is closer to the first element or the last. This is defined by comparing the position with the size of the linked list. Each element of a linked list (especially a doubly linked list) uses a bit more memory than its equivalent in array list, due to the need for next and previous pointers. Furthermore deleting or inserting an element somewhere in between existing elements is relatively cheap since only the pointers of the elements before and after have to be updated.

3.3.2.3 Plain Arrays

Plain arrays are more or less the same with ArrayLists. They are slightly more difficult to work with from a programmers point of view since if they get full they have to manually be moved to a new larger location in memory. Issues like that are taken care of automatically for ArrayLists. Also in plain arrays, the size has to always be predefined therefore they probably use more space than ArrayLists. ArrayLists are also predefined internally and expanded if required.

3.3.2.4 Data Structures Issues

In the beginning I was creating all IS and DS at a go, then all M+s and then Ms. In other words, every batch number of inputs, in order to find their match sets following the batch matching approach, I had to compute the Intersection set, the Disjunction set and then the subset of the population and super set of the second input's M (M+). In order to do this I was using ArrayLists of ArrayLists to store all those values (different IS, DS and M+ for every batch were computed and stored). When I checked and compared speed times between ArrayLists and plain arrays I discovered that arrays are faster so it would be better to use them instead. But since I was calculating all of them and storing them at the beginning before computing Ms, I would have to pre-initialize all the arrays to do that and it would make the code impossible to follow. Instead I used a single IS, DS and M+ (for batch size=2) and I was making the complete calculations for each batch; I was calculating all Ms of all inputs in the batch size set and after completing the work with the first batch I was moving to the next one, therefore I was able to overwrite IS, DS and M+ and reuse them again without having to create an ArrayLists of ArrayLists to store everything as I was doing before.

3.3.3 Other Enhancements

3.3.3.1 BitSet

In section 2.3.1 bit encoding was mentioned. This encoding is used so that the symbols of 0, 1 and # that are used for this problem are represented in a new manner that saves memory in storing them. If input and rules are stored as strings, they occupy their length times 8 bits of memory. Some languages may use a different amount of bits to store characters but 8 is most often the case.

<i>bit₁</i>	<i>bit₀</i>	<i>ternary value</i>
0	0	#
0	1	0
1	0	1
1	1	#

Table 4. Source: Ref. [2]

Java's BitSet was used to represent this encoding:

```
static BitSet zero = new BitSet(2);
static BitSet one = new BitSet(2);
static BitSet wildcard = new BitSet(2);
zero.set(1);
one.set(0);
wildcard.set(0);
wildcard.set(1);
```

Figure 16 - /PatternMatching/src/patternmatching/MatchNBitSet.java

BitSets are sets of bits as the term describes. The bits can be either set (equal to 1) or unset (equal to 0). The code above describes the definition of the elements 0, 1 and # in terms of bit sets. The default value of each bit in the BitSet is unset.

- To represent '0': the first bit has to be zero, therefore nothing is done for that and the second bit has to be 1 so it is set.
- To represent '1': it is exactly the opposite with the '0' representation but setting the first bit instead of the second.
- To represent '#': both bits are set.

3.3.3.2 Enums

Enums are a feature of Java that was introduced in version 5.0. An Enum in Java stands for Enumeration. This component was introduced in order to provide an easy way to represent entities that have fixed properties and belong in a certain group. A Java Enum class is defined whose fields (properties) are constants. For instance, the months in a year may be represented as an Enumeration. Enum fields are represented in uppercase letters because they are constants. They are very powerful and flexible since they allow developers to model objects within them.

The use in this project is to represent the elements of the inputs and rules as Enums. For this purpose a new java file was created called Ternary.java that only defines this new Enum type:

```
public enum Ternary { ZERO, ONE, WILDCARD }
```

Figure 17 - batch-matching/BatchSizeN/Ternary.java

Figure above displays the whole file. It is very short and only defines this new Enum type so that it can be used by the rest of the code to represent inputs and rules as arrays of this new type (Ternary[]). PopEnums.java is an example of this case. The gain of using Enums instead of strings is that we can save more memory representing all the inputs and rules. But this project is not concerned with memory issues; it is only concerned with speed issues. As a result from the experiments that took place, Enums also helped in gaining some speed of about 8%. Here is an example of adding an Enum input (11100) in the matching instances:

```
input.add(new Ternary[] {Ternary.ONE, Ternary.ONE, Ternary.ONE, Ternary.ZERO, Ternary.ZERO});
```

3.3.3.3 PreComputed Match Sets

In the process of finding match sets for inputs, the algorithm accepts an arbitrary amount of inputs and rules and tries to figure out what rules match to each input. As reasonable, some inputs may appear more than once in the database. The idea behind pre-computed match sets is that in case of having the same input more than once, we should avoid re-computing the match set for the second instance since they are going to be the same with the first one and the only thing we are accomplishing is to waste CPU cycles. The algorithm should somehow verify that an input has already been matched and fetch its match set instead of doing the same calculations over and over.

A nice approach for this problem is to store inputs in a HashMap instead of an array or a linked list. HashMaps, and generally maps, have the feature not to add duplicate

values. By doing so we ensure that duplicates do not exist, therefore are not re-calculated. Adding values in HashMaps is slower than adding them in arrays or linked list since some calculations have to take place in order to find the position best suited for the insertion. Therefore it is not clear whether it will be faster or slower than by simply having inputs in an array.

3.4 Ternary Tree

After the completion of the batch matching algorithms and the various tweaks to improve the running times it was time to try a completely different methodology. The idea behind this is to find or create a suitable data structure, tree in this case, to add the rules of the population so that searching through them would be more efficient than simple arrays or linked lists.

In 1963, the Dutch mathematician F.J.M. Barning described an planar, infinite, ternary tree whose nodes are just the set of Primitive Pythagorean triples [36]. Seven years later, A. Hall independently discovered the same tree [37]. Both used the method of uni-modular matrices to transform one triple to another. A number of rediscoveries have occurred more recently. It is essentially a variation of the binary tree but instead of each node having two children it has three. This tree seems to be suitable for inserting ternary language values since they contain three symbols (0, 1 and #) so they will be split accordingly by having each symbol in each child. For the purpose of this project two variations of the ternary tree are proposed which are the two extremes. One just splits the rules only based on their first element and creates three categories (Single Depth) and the other fully splits and organizes the rules depending on all the elements they contain (Multi-Level).

3.4.1 Single Depth

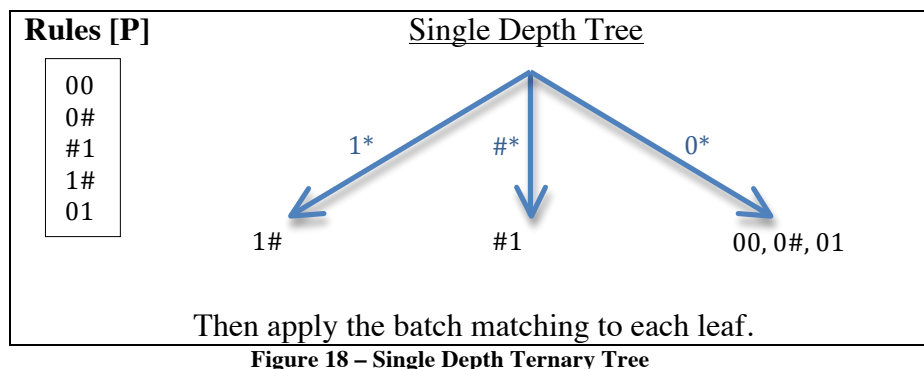


Figure 18 – Single Depth Ternary Tree

The figure above describes the first version of the proposed single depth tree. Its purpose is to split the population of rules into three categories (one that starts with a 0, one that starts with a 1 and another that starts with a #). This categorization of rules will take place during the insertion in the tree. Another simpler way to look at it is that it takes the rules from a bigger array and puts them in three separate arrays that are smaller in order to make searching more efficient. For example, if an input starts with a 0, then it will only search leafs (or sub arrays) that start with a 0 and a # since in the leaf starting with 1 it will certainly not find any matches. In theory this will save about 1/3 of the time on each input trying to find its match set of rules.

Since the tree just splits rules in three categories, it can be used together with batch matching algorithm to speed things up even more. Each leaf is basically the same as the whole population, the only different is that it contains fewer elements and all elements have their first character in common. So why not locate leafs that need to be searched and then apply the batch matching on those!

In Source/trees/TernaryTree/ you can find the implementation of the single depth version as well as the implementation of the multilevel tree. TTree1.java is for the single depth. What it basically does is to create three arrays called Mone, Mzero and Mhash. As their names imply, each stores values that start with that element. E.g. if a rule starts with a 0 then it is stored in Mzero, if it starts with a 1 it is stored in Mone and if it starts with a hash is saved in Mhash. Now as soon as an input comes in to find its match set it only searches:

- If the input starts with 1, search Mone & Mhash.
- If the input starts with 0, search Mzero & Mhash.

3.4.2 Multi-level

This is a more complex version of the ternary tree. It categorizes rules to a full depth. The depth of the tree is equivalent to the length of the rules.

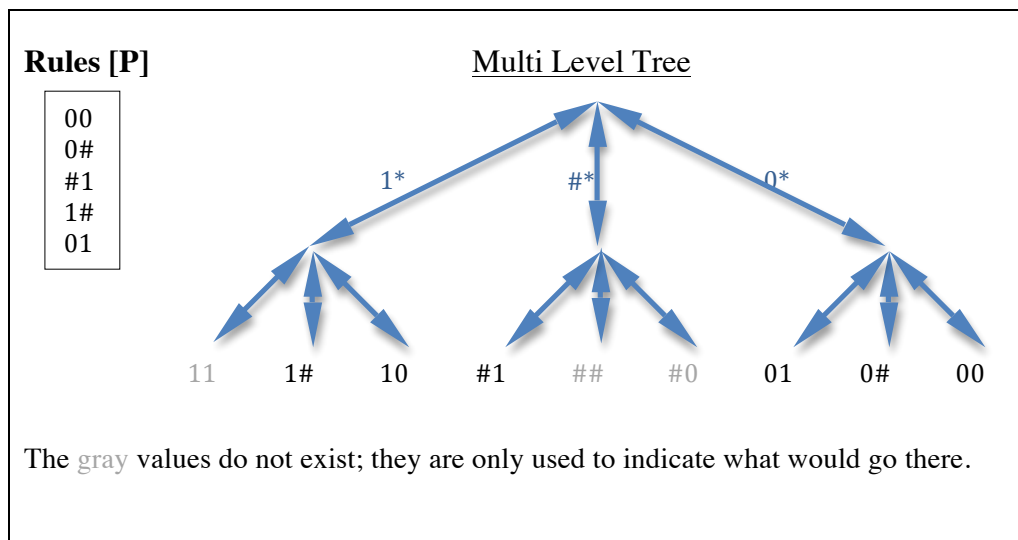


Figure 19 – Multi level Ternary Tree

The main goal of this multi level ternary tree is to fully categorize the rules. All rules are located at the leaf nodes and each leaf can only contain a unique rule. Therefore the path that needs to be followed to find or to insert or to delete a specific rule is one and only one. This is exactly the reason why this tree has good potential for fast execution times on inserting and deleting specific rules.

Another way to look at the tree is that it categorizes (sorts) the rules in an optimal manner for inserting and deleting specific rules. Each node, as shown above, points to both child and parent. The reason for this is that in case that there is a rule deletion from the tree, we need to start moving backwards and also delete its parent if it has no other children (the one deleted just now was its only child). This will keep the tree as compact as possible avoiding extra steps in searching to find existing rules.

To implement this approach I used two different classes. The one is called `Node.java` and stores the nodes of the tree; remember that only leafs can have values stored therefore the value variable remains empty on the non-leaf nodes. At this point the `Node` contains a value, and three pointers to its three children (left, middle and right). This is extended for the use along with the UCS since this implementation has some issues on deleting items. See section 3.5. After that `TTreeN.java` was implemented which basically implements the population of the rules. It contains methods to add, remove and find specific rules and also a method that returns the match set of a given input. The method is called `getMatchSet()`.

3.5 UCS Integration

The scope of this project was to build versions of Batch Matching that would work with any number of inputs and rules (size N). Then make various tweaks and experiments to see which of those is the fastest. Many variables are introduced into this problem and many roots were taken to find what works best. After the completion of the first stage, to implement and evaluate batch matching, the second major milestone was to propose and implement, if time was adequate, a new tree type that would support this type of process. A quick reminder, the process that we are looking for is to find which rules match to which input from a given database of inputs and rules. The second major milestone was also completed successfully proposing two different variations of the Ternary Tree, and one of those does actually seem to work wonders for the purpose that is needed. More details in the Ternary Tree and experiment sections.

Since all the work in the scope was completed successfully and I still had some time available I thought to take this project one step further and attempt to integrate the solution in the UCS and see how they work. My supervisor Tim Kovacs provided me with the UCS implementation of Dr Gavin Brown of University of Manchester which he had expanded. The system contains many classes and it is quite difficult to follow since you do not know where to start. With Kovacs's help and guidelines I opened only about 5 classes of the system and started making changes in order to integrate the Batch Matching system. This task proved to be impossible to do within the time span of the project because the way that UCS works is to take one input at a time and perform on that many calculations and then move to the next input. Batch matching on the other hand works differently; it takes a batch size of inputs together and performs operations on all of them in order to come up with their match sets. For this reason and after a couple of days struggling with the code I could not integrate them properly. Some of this work can be found in `Source/ucs/ucs/`. The classes I worked with mainly are `UCS.java`, `Population.java` and `Indiv.java`. Also in `Source/ucs/demos/` you can find the location of the various test cases that UCS can run properly. Once again, this work does not belong to me, Brown and Kovacs wrote it, I just made some changes.

Some of this work was done in parallel with the implementation of the Ternary Trees proposed. After the integration of Batch Matching into UCS had come to a dead end I finished with the implementation for Ternary Tree and it was time to test how it performs. See section 4.3 for the results. Ternary tree implementation is quite

complex; therefore this is added as an appendix so that the reader can be looking at the source while reading this section. Appendix D helps to grasp what is going on. Also Node.java implementation is important but not located in the appendices. Both these versions are those that were used along with the UCS code.

Since the results were better than I expected, and because the way Ternary Tree works is to get each input in the database and search the whole tree for matches by following certain paths of the tree, it seems to be more reasonable and easy to integrate in the UCS than Batch Matching proved to be. For this purpose I created a new copy of the ucs directory which is located in Source/efficient-matching/ucs/ and had a go at this. In the folder just mentioned, there are some files that I created and some I modified. TTree.java and Node.java implement the multi-level ternary tree proposal. Of course some changes had to be made in order to be properly integrated with the existing system. First of all, the current UCS code uses double arrays to represent inputs and rules (1.0 for 1, 0.0 for 0 and -1.0 for the hashes). This notation had to be used in this version of the tree in order to be properly combined with the rest of the source.

After implementing the first step, I noticed that the current version of UCS used the Indiv.java class to create the rules. This class contains the rule in the form of the double array and some other values. Therefore the new tree was altered once again to accept nodes of Indivs instead of double arrays. To make those changes I had to modify both Source/efficient-matching/ucs/ucs/TTree.java and Node.java to support it.

Next step was to have a look at the existing Population.java class and try to implement all the method it contains but to work with the ternary tree instead of ArrayLists. All were converted and are located in TTree.java but not all of them have been tested. In order to do that I had to introduce some new global variables in the tree class. Finally it was time to modify the UCS.java to make it work with the new data structure.

The methods that I was concerned with are:

- `getMatchAndCorrectSets()`: this method is called once for each input and goes through all the population, essentially through all the elements of the ArrayList and tries to find the rules that match. The old version is commented out and a new one that works with ternary trees was introduced. Now the method is much more compact since as mentioned before, TTree.java contains a method that finds the match set of a given input. Therefore all that had to be done in the newer version of this method is to call `getMatchSet()` passing the input.
- `deleteIfNecessary()`: this method deletes random rules from the ArrayList, and now it should remove random rules from the tree. This is a bit more complex than expected and it is described in the next paragraph.

In order to delete a random rule from the list a random path has to be followed until a leaf is reached. Note that only leaves hold the actual values so this is the only approach. Therefore in order to implement this solution I had to add some extra code in Node.java that return which of the children are not NULL. Then the `removeRandom()` method in the tree class knows which path to follow (a random one as long as there are still children available). Once a child is found, it is then deleted but there is still

work to be done! As the tree needs to remain as compact as possible we have to check if the deleted node was the last child of its parent and if it was delete its parent as well (this is the reason that in Figure 19 the arrows are bidirectional). This process had to repeat itself until a node was reached that also had another child available. To do that, I added a parent node in the Node.java which is basically a pointer to the parent of each node. This way I was able to move backwards and delete nodes. Also a new method called `getChildren()` is introduced which returns the number of not null children a node has. This is used to check if the one just deleted was the last.

A notion that was not known before and had to be faced with during the UCS integration was the numerosity. Numerosity is the number of times that each rule appears. For example if a rule already exists in the Ternary Tree and the scenario requires to add the same rule again, in Indiv class, there exists a variable called numerosity which is incremented instead. Basically this variable contains the number of time that each rule appears. Therefore, if one rule is added once, then the code has to add a new node, if it is added again then it only has to increase the numerosity variable of that node. The same applies for removing rules. If the rules numerosity is larger than one, then you just decrement it by one, else you entirely remove the node and its parent if that was the last children.

Finally I would like to mention that the code was not fully tested using the UCS. Only partial testing to make sure that individual methods were working correctly. But once again, this part is outside the project scope and should be considered as an extra.

4 Experiments/Results

4.1 Introduction

This section describes the experiments that took place and evaluates and justifies the results. Although the reason behind this project is to be integrated some day in Learning Classifier Systems, for this point it is better and more reliable to run tests on random inputs and rules. Running standalone tests can focus on more specific aspects of the new implementations and measure individual sections to find out how they perform.

My supervisor realized that UCS is not such a good test anyway. Although we ultimately want to improve the speed of the systems like UCS, there are a lot of different ways to implement it and this has a big effect on the results. It's even possible that one improvement (like batch matching of trees) is better than another with one version of UCS but worse with another. So, although matching random rules is not the ultimate goal of studying matching, it gives more objective results than using any one implementation of UCS.

4.2 Experiment Code

The testing code for the batch matching part is split into two files.

4.2.1 Experiment.java

Experiment.java is used to set up the experiment. It has a main loop that repeats the test as many times as defined. This is for alleviating any noise from other reasons to produce more clean values as results. It has functions inside to construct inputs or rules of a specified length. The length is defined as the DIMENSION. Moreover, it can accept a percentage of HASH values (don't care symbols, #) called HASH_PROB and make the rules/conditions contain that percentage of hashes.

This method also sets output parameters and creates a specified number of random inputs (NUM_INPUTS) and a specified number of random rules (POP_SIZE). It can also set the batch size (BATCH_SIZE) that it needs the batch matching to have. By varying that number different results can come out, and we can compare how well each batch size performs.

The way the experiment is set up is to call two methods as many times as the NUM_REPS variable defines. Those methods are:

- repeatNonBatch() – finds the match sets of the given inputs following the naïve matching algorithm.
- repeatBatch() – finds the match sets of the given inputs following the batch matching algorithm.

these methods are located in the population classes.

Both these methods are called one after another in order to ensure that both use the same inputs and rule sets so the results are fair to both parties. If that was not the case and they were using different data, one might benefit by the fact that its input/rules sets were easier to work with.

After calling `repeatNonBatch()` and `repeatBatch()` it uses another class called `Averager` in order to be able to compute various averages that are explained in the next section and finally prints the results.

4.2.2 Averager.java

`Averager` is a class used by `Experiment` class in order to store the data of each iteration and find the average of them at the end, then pass it back to `Experiment`, which simply prints it.

The code in `Averager` can collect:

- Number of comparisons
- Times that each iteration requires (in milliseconds)

It stores everything in `ArrayLists` and when asked it can compute and report back to `Experiment` the following:

- Average of comparisons
- Average running time
- Variance
- Standard Deviation

4.3 Batch Matching

There are many different implementations of batch matching in this project, but the question is which one performs better and under what circumstances. This section aims to give an overview of the most important comparisons that were tested during the experiment phase. Note that if you compare the batch matching of one section, with the times of the batch matching of another section the times might not be exactly the same. The parameters that each experiment uses may alter the results. Each section will include information of under what conditions the experiment was carried out.

This is a table that shows some how some properties are called and what is equivalent with what. This project is related with some papers that implemented experiments in the same field but for different approaches and those paper may call things differently than this project.

This project	Equivalent	Butz et al. [13][14]
Population	=	Population (sometimes N)
Hash Probability	=	Generality
Inputs	=	Match instances

Rules	=	Conditions
Dimensions	=	Problem Size

Table 5 – Names for Properties

Inputs are binary strings (e.g. 10010), rules are ternary strings (e.g. 10##1), population is the number of rules, Hash probability is the number of hashes (#) found in the rules divided by the number of positions, inputs is the number of inputs and dimensions is the length of the inputs and rules. In this project there were also some other names introduced such as batch size, which is the number of inputs compared at a pass, Intersection Set (IS) which is the indexes of the common elements of batch inputs and Disjunction Set (DS) which is the indexes of the non-common elements.

4.3.1 Naïve vs Batch Matching

This experiment compares the naïve matching implementation with the plain array batch matching implementation. Up to this point in the implementation there was no comparison between linked lists, only between ArrayLists and plain arrays. Out of the two, plain arrays seem to perform better therefore they are used in this experiment, as they are the best approach so far.

The hash probability used for here is 33%, the number of inputs is 2000, the dimensions of input/rules is 100. Population size and batch size are variables for this test therefore they have different values. And the results are on an average 50 runs to eliminate any noise.

Population	Naïve/Linear	Batch Size 2	Batch Size 3	Batch Size 4	Batch Size 5	Batch Size 6	Batch Size 7	Batch Size 8
1000	85.66	47.26	32.68	28.96	31.96	47.5	64.9	77.02
2500	207.24	113.16	77.52	69.08	80.64	114.98	160.74	189.16
4000	345.06	180.62	123.8	99.84	121.88	184.3	268.68	306.84
5500	486.68	254.46	171.64	138.38	181.64	277	383.42	454.94
7000	650.62	327.54	222.58	172.84	221.68	370.7	513.52	594.02
8500	815.52	383.6	254.92	221.5	306.02	446.6	602.6	729.14
10000	1008.64	449.74	307.58	257.94	332.18	487.44	625.4	739.42

Table 6 – Naïve vs Batch matching results

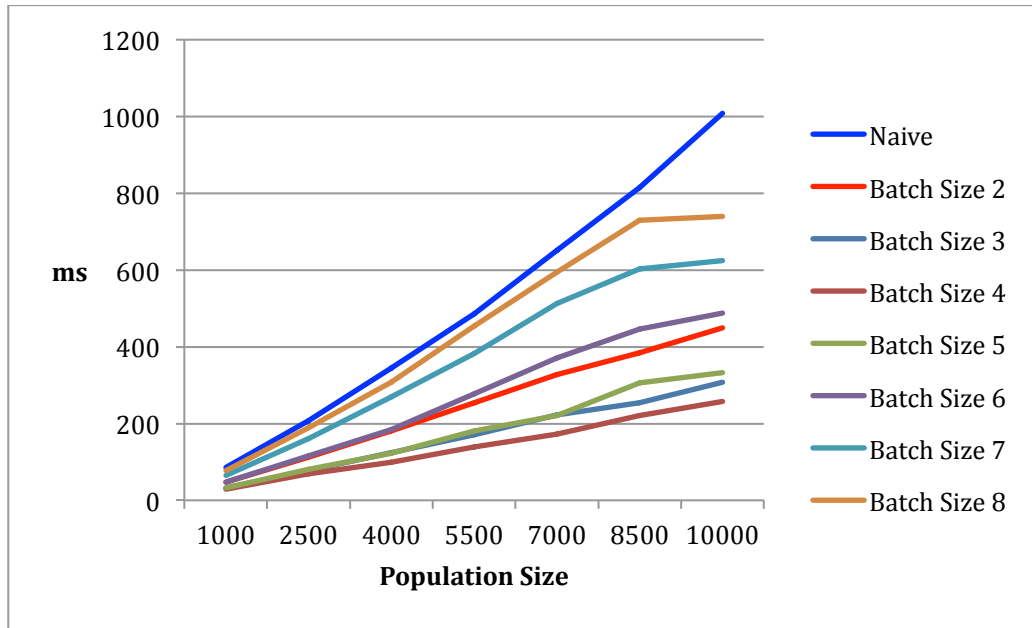


Figure 20 - Graph representation of the table above

As we can see from the table and graph above, batch matching is better than naïve matching in all the cases that we have tested. The best times for these data sets are given by batch size 4. It seems that getting 4 inputs at a time and comparing them finding their intersection and disjunction set and then creating a smaller population size for the remaining 3 inputs of the batch, was faster than any other combination. All matching processes are more or less linear; you probably noticed that the lines are not exactly straight and the reason for this is simple. For every execution the inputs and the rules are randomly calculated in Experiment.java therefore those variations are what we see in the graph.

As the number of batch size increases the proportional time to compute the match sets of the inputs also increases. This is because the more inputs compared together, the smaller the chances that the intersection set is large, since the elements of specific indexes of ALL batch size inputs must be the same. Having a big Intersection Set is essentially what saves times in the long run. If their common elements are a lot, this means that calculation on those element positions will only have to be carried out for the first input of every batch therefore saving time on the next inputs. Only the Disjunction Set indexes will have to be checked, and if the Intersection indexes are a lot, it means that DS is going to be small. Having a large IS, will not remove many elements from the population since many of them will match all elements, but in relation with naïve matching, that has to keep comparing as long as the elements match, it is much faster! Consequently, $A+B > C+D$ as mentioned in section 3.2.

4.3.2 Adjacent vs Distant Batches

The two major approaches for batch matching are compared in this section. Adjacent batches are those who are next to each other and distant batches are batches that all contain the first input of the matching instances. In order to compare them they had to be implemented in java. Inside Source/efficient-matching/batch-matching/BatchSizeN/ you can find many files related to batch matching.

Population.java is the first implementation of the normal batch matching, in this case referred to as adjacent batches. Population2.java contains an implementation of the distant batches approach, which basically always uses the first input and compares it with the rest.

ArrayLists were used to store data and the test parameters for this experiment are:

- Batch size = 2
- Hash probability (generality) = 50%
- Inputs (match instances) = 2000
- Problem size (dimensions) = 100
- Averages on 50 runs
- Population size changes

Population	Naïve	Naïve St. Dev.	Adjacent Batches	Adj. St. Dev.	Distant Batches	Dis. St. Dev.
1000	134.48	0.361	91.18	0.477	138.5	1.418
2500	340.14	0.4	213.68	0.466	338.28	0.634
4000	548.32	3.075	353.34	0.552	536.06	0.676
5500	733.58	0.724	450.96	0.72	696.74	5.328
7000	925.64	0.933	557.98	0.648	841.9	1.1
8500	1128.6	4.894	687.12	0.765	1005.1	1.33
10000	1337.04	4.591	794.88	0.993	1256.76	6.58

Table 7 – Naïve vs Adjacent Batches vs Distant Batches matching

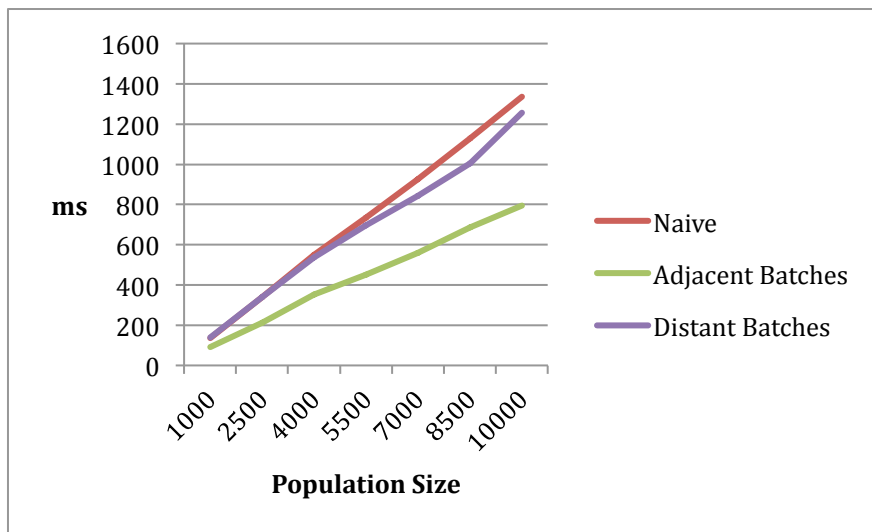


Figure 21 – Graph representation of the table above

The reason that both table and graph are displayed is that the table shows the exact values and also indicated the standard deviations, but the graph helps in a better understanding on what is going on. Obviously here the adjacent matches approach proved to be the fastest of the three. Also as expected, both adjacent and distant matches are faster than naïve matching. The question that rises here is why distant matches are slower than adjacent.

If you think about it, the reason is quite easy to understand. As noted before, batch matching has the ability to find more than one match sets on a single pass, or at least find their supersets (and subsets of the initial population). By keeping always the first input and comparing it with the rest of the inputs, we loose the element of “batching” because always the single pass passes through the 1st element. Therefore, it is not gaining any extra time starting comparing with a new input each time. On the other hand, adjacent matches changes the first input each time. Consequently, on a single pass it finds the match set of a different input rather than finding the match set of the same constant input. For this reason, distant batch method is going to be used from now on, as it is the most efficient.

4.3.3 Adjacent vs Recursive Approach

In order to be able to test the recursive approach, I had to implement code that would allow the user to change the batch size in both adjacent and recursive. Recursive would only make sense if the batch size were at least 3, and then would gradually decrease until it reaches 2. The reason that it never gets down to 1 is explained in section 3.3.1.4 where the recursive algorithm is explained. For this reason, and in order to be fair amongst the two algorithms compared in this section, adjacent approach also had to be extended to support any size of batch size. The code that allows the adjacent approach to accept any size of batch size was first written in order to get results for the first experiment, which compares how each batch size performs under some given parameters.

I will compare now how adjacent and recursive perform. There are many implementations of these two algorithms. For example in folder Source/efficient-matching/batch-matching/BatchSizeN/, Population4.java is for the adjacent approach and Population5.java is for the recursive. These files use ArrayLists. There are also implementations available that use plain arrays (Pop4.java and Pop5.java) which actually proved to be faster. But for these test Population4 and Population5 will be used since their code is cleaner and easier to comprehend. The relation on how they execute does not change if the ArrayList implementation are used. Only that both of them perform faster. From the naïve vs batch matching section, it proved that for the specific parameters given, batch size 4 was the fastest. For this reason, and because the parameters in this test are similar, batch size 4 will also be used so that we know that the adjacent batches algorithm performs at its best.

Parameters for this experiment:

- Batch size = 4
- Hash probability (generality) = 50%
- Inputs (match instances) = 2000
- Problem size (dimensions) = 100
- Averages on 50 runs
- Population size changes

Population	Naïve	Naïve St. Dev.	Adjacent	Adj. St. Dev.	Recursive	Rec. St. Dev.
1000	92.64	0.48	49.04	0.631	38.22	0.414
2500	237.36	1.292	123.56	0.496	90	0.283
4000	374.92	2.067	198	0.4	141.48	0.5
5500	520	1.497	277.06	2.213	196.62	0.914
7000	669.46	2.051	370.36	2.347	268.08	0.744
8500	811.88	1.996	455.46	4.916	322.48	2.766
10000	957.84	4.361	597.28	2.691	414.86	1.876

Table 8 – Adjacent vs Recursive approach

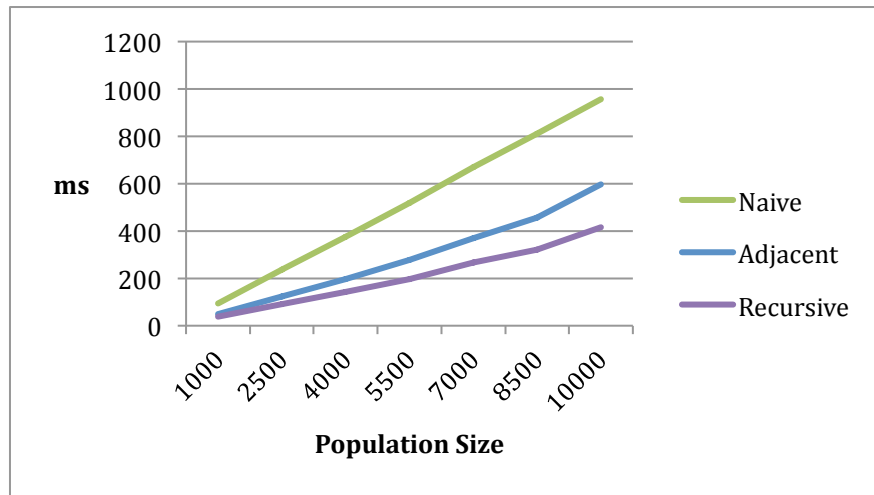


Figure 22 – Representation of the table above in a graph form

The outcome of this test denotes that recursive algorithm is faster than the adjacent. Also the number of comparisons that were required for the process to find all match sets of every input were slightly less for recursive (not shown by the numbers in the test here). A reduced number of comparisons is an indication that less processing power is required.

Speed is what this project is aiming for and it seems like recursive can produce even faster execution times. By the use of the recursive algorithm we gain all the benefits of all the batch sizes. It starts using a big batch size and it gets smaller and smaller up to the point that it becomes 2. This effectively means that we alleviate the problem appeared in the Distant matches approach since the first element of the batch is constantly changing (moving to the next). Furthermore, each stage reduces the $[M+]$, which is the population that needs to be checked for the rest of the inputs of the batch. By the time that the algorithm reaches the final input of the batch the size, the population is small enough to gain important amounts of time since it had to match to all the common elements of all the inputs that have been checked so far. Thus, this algorithm gives slightly better results than adjacent batches methodology.

4.3.4 Generalities Test for Adjacent

Many different types of experiments can take place in this project because the parameters that the project uses are plenty. By changing once of the population, input values, hash probability, batch size or even comparing various approaches to solve

this problem like the alternative methods of batch matching and the use of new data structures like the proposed versions of ternary tree. This test was inspired by the paper by Butz et al. [13] where they performed a similar test for their experiment comparing how the generalities (hash probabilities) behave in relevance to the population size. Also this test was displayed on the poster.

For 2000 inputs, batch size 3, dimensions 100, on average 50 runs and variable population (N) and generalities these were the results:

N	gen	Average Time (ms)	Average Time (ms)	Standard Deviation	Standard Deviation	Comparisons	Comparisons
1000	0%	53.04	21.94	0.196	0.369	535823	1334248
1000	25%	69.38	29.12	1.164	0.325	5358239	1783471
1000	50%	90.4	36.9	0.49	0.361	7752661	2629500
1000	75%	118.94	54.7	1.515	0.608	1.60E+07	5760610
1000	99%	263.62	403.78	0.485	1.082	1.59E+08	1.28E+08
5000	0%	268	108.7	1.342	2.516	2.00E+07	6668477
5000	25%	360.28	141.54	1.04	1.153	2.68E+07	8916142
5000	50%	465.94	178.5	1.19	2.678	4.00E+07	1.34E+07
5000	75%	621.18	267.98	1.956	1.319	7.90E+07	2.88E+07
5000	99%	1331.6	2003.12	18.887	9.013	7.88E+08	6.32E+08
10000	0%	607.38	232.72	7.175	1.059	4.00E+07	1.33E+07
10000	25%	753.9	299.4	4.192	2.771	5.33E+07	1.78E+07
10000	50%	993.16	381.64	11.606	5.214	7.97E+07	2.68E+07
10000	75%	1263.16	562.64	7.514	8.65	1.60E+08	5.76E+07
10000	99%	2754.26	4179.32	5.524	10.578	1.58E+09	1.27E+09

Table 9 – Population size and generality behavior. Blue = naïve, green = batch (adjacent)

From this table everything seems normal. Batch matching seems to always be the best of the two which is what someone would expect. Having a closer look though, shows that in generalities 99% batch matching actually becomes slower than naïve matching something that my supervisor did not anticipate. So this is the first sign of weakness that the new approach has shown. This can be explained by the fact that because there are a lot of hashes in the rules (99% of them are don't care symbols), [M+] does not get significantly smaller than [P] when we have the pass of the first input. Therefore, time is wasted on generating this new [M+] that should be smaller than P in normal circumstances and no time is saved later whilst checking the rest of the batch inputs against this new set since it has about the same size as P.

4.4 Other Tests

The project is composed on many parts. Some of them did not evolve as much as other did. For example there are source files that attempt to use Enum types instead of strings. At the initial stages, in the NetBeans project located in Source/PatternMatching/src/patternmatching the enums seem to perform about 8% faster than the code that uses strings. Here is a table of 10 execution times of the two codes:

	BatchN	BatchNEnum	
	8100000	9399000	
	12365000	8532000	
	8074000	8748000	
	8115000	8203000	
	8149000	9380000	
	10761000	10826000	
	8194000	9145000	
	9404000	9269000	
	16266000	8657000	
	8272000	10212000	
	10406000	8235000	
Average	9827818.182	9146000	nanoseconds

Table 10 – Strings vs Enums comparison

But in later stages when the code became more and more complex, if the strings were changed to Enums, they were actually found to be a bit slower.

In the aforementioned folder there is also an attempt to avoid computing match sets for already computed inputs if they appear more than once. In order to achieve that I tried using HashMaps to store inputs inside. This ensured that if an input appeared more than once, it would be stored only once since Maps do not save any duplicates. Then I was using the values of the HashMap to find their match sets instead of using the inputs ArrayList. After finding all the match sets using batch-matching approach and storing them in the HashMap, then I used the ArrayList to retrieve and print the match sets of all the inputs that appeared in the ArrayList. Therefore the final outcome still appears the same since if an input was found more than once in the ArrayList, it would fetch its match set twice from the HashMap (but only compute it once). This way of implementing the pre-computed match sets was found to be unsuccessful and was running slower than the simple batch matching, which computes match sets again for duplicates. After making some research I discovered that the reason that this has happen is that it actually takes some time to insert and retrieve data from the HashMap, since it has to compute a location and then go and fetch them. Thus, the unceasing need of retrieving and storing data into that HashMap made the code run slower than the original one.

Yet another attempt was to try and convert everything into the specificity encoding described in section 2.3.2. The files can be found in Source/efficient-matching/batch-matching/BatchSizeN/. SpecConverter.java converts strings to and from specificity encoding and Popul4Spec.java uses that class to run the batch-matching algorithm. This effort also was not very helpful since a lot of time was wasted on the conversion to and from specificity encoding therefore was not adding anything as far as efficiency is concerned. This would probably be a good idea if this project was concerned with building a memory efficient application but we are only concerned with speed.

Sorting inputs is an important aspect of this project that was not mentioned much. Sorting inputs helps every version of batch matching in performing better. Inside Experiment.java there is a commented out command that sorts inputs before making any calculations. This works only on the ArrayLists implementations since the limited

time span of the project allowed me to implement only the most important sections. The general consensus is that sorting helps but there are cases, which it does not. I will give an example where that might happen. Lets say that we have two inputs that start with 000 and they are followed by any other combination. Sorting inputs will bring those two inputs next to each other, therefore batch-matching algorithm will perform faster since adjacent nodes will have a bigger Intersection Set. On the other hand if there were 2 inputs in the matching instances, 0000000 and 1100000, those would be placed apart if they were sorted even if they have 5 out of 7 indexes in common. So this would not help batch-matching to perform faster. The best possible way of sorting inputs is based on their common indexes.

4.5 Ternary Tree Tests

4.5.1 TTree1 vs TTreeN

Two versions of the ternary tree have been proposed. The one is the single-depth that has the ability to work with batch matching and the other one is multi-level that works by itself. TTree1 refers to the single depth, and TTreeN to the multi-level.

Now we have to see which of those two extremes of ternary tree performs better. For this purpose a new directory is created (Source/efficient-matching/Trees/TernaryTree) that compares the two. Experiment.java and Averager.java were used to perform these experiments. Having a hash probability of 33%, 2000 inputs each being 100 long and batch size 3, these were the average results on 50 runs.

Population	Array/Naïve	Array/Batch	TTree1/Naïve	TTree1/Batch	TTreeN
1000	75.52	31.12	61.46	31.76	18.16
2500	197.26	74.2	163.28	75.52	35.04
4000	314.12	119.66	247.78	122.98	51.96
5500	431.94	119.66	247.78	122.98	69.8
7000	563	216.76	441.62	214.62	86.02
8500	688.44	264.72	541.86	267.96	102.24
10000	838.46	317.08	649.84	328.4	118.36

Table 11 – TTree1 vs TTreeN

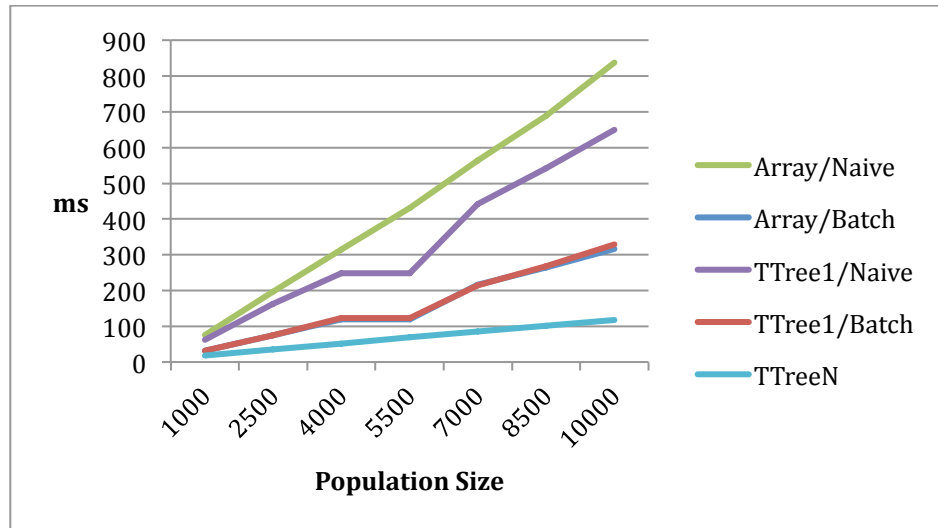


Figure 23 – Graph representation of the table above

Naïve is always present in the graphs to indicate the difference with the existing approaches before this project. This was the approach used in other papers to find match sets of different inputs.

Viewing the graph you can notice that single depth ternary tree (TTree1) slightly improves naïve matching if used by itself. Think of the tree (figure 18). It splits the population in 3 parts. Then depending on what the first element of the input trying to be matched, it only uses 2 out of the 3 parts. Basically it ignores about 1/3 of the population. This is exactly the reason that the TTree1/Naïve line is faster.

Now lets move to the next two, which are Array/Batch and TTree1/Batch. As shown in the graph the results seem to be about the same. The cause of this is the fact that in order for the single depth tree approach to be faster it has to happen that the first elements of the inputs of each batch are the same. E.g. 10010, 10000, 11111 these three inputs have the first element common (1). Now imagine the size of the batch being larger; the chances of the first element being the same in all the inputs gets very small. Since the single depth tree splits rules only based on their first element, this means that in order for TTree1/Batch approach to save extra time it has to happen that all batch inputs have the same first element. Because the chances of that happening are not a lot, the results of the array and single depth ternary tree in relation with batch matching are about the same.

Finally, multi-level ternary tree seems to perform surprisingly well; certainly better than any implementation of batch matching will ever be. It seems that this version of the tree has many benefits for the purpose that we need it. Instead of going through all the elements of the inputs and rules checking them against each other if they match (naïve), or even doing more calculations on a single pass to gain speed by discarding some rules for the next inputs (batch) it has the ability to follow certain paths in the tree and check only those for rules that match. For instance it can take a specific input and only follow the paths that if a rule is found on those paths then it belongs to the inputs match set. It does not bother with the rest of the tree no matter how large it is. In this version, the inputs and the population (rules) are all given prior to starting the actual matching process. But in a real time scenario, like in the UCS implementation, some rules may be removed on the way. For this purpose I had to perform yet another

test to check how well does this version to ternary tree perform on adding and deleting elements.

4.5.2 ArrayLists vs LinkedLists vs TTreeN

In order to test how well does the single depth ternary tree and multi-level ternary trees perform against arrays or ArrayLists, I had to write some extra code to execute the tests for me. The source can be found in Source/efficient-matching/Trees/DataStructures/. For this specific test I got some advice from my supervisor, Tim Kovacs, who guided me in what form the experiment should have. Specifically he told me how UCS behaves and how it works with rules/conditions and I tried to simulate this behavior on a simpler test scenario.

Timing.java in the folder just mentioned, takes an input from the terminal. That input indicates the size of the population. Then the code creates a smaller population of size 400 and then it keeps deleting 2 random elements and adding the next two of the larger population. This repeats until all the elements of the larger have been added to the smaller population. The large population is created so the time it takes to create the random rules that are included in it is not included in the test. The aim is to check how fast does each data structure tested here perform.

Testing parameters:

- Hash probability (generality) = 33%
- Population (rules) = 10000
- Dimensions = 100

Results:

INSERT/DELETE TEST			
ArrayList:	Avg 669 ms	St.Deviation 40.496913462633174 ms	
LinkedList:	Avg 640 ms	St.Deviation 25.748786379167466 ms	
TTree:	Avg 206 ms	St.Deviation 9.486832980505138 ms	
DATA ACCESS TEST			
ArrayList:	Avg 0 ms		
LinkedList:	Avg 0 ms		
TTree:	Avg 29 ms		

Figure 24 – ArrayList vs LinkedList vs Ternary Tree (multi-level)

The results of this figure show that the ternary tree is much faster when it comes to inserting and deleting rules from the population. This is because it knows the position where the element to be deleted is located and only follows one path to find if it exists and also during adding, it never needs to expand as ArrayLists do. Also the ternary tree keeps itself as small as possible because on rule deletion it also deletes the parent of the node if it has no remaining children. Hence, this version seems to make a perfect candidate for the optimal solution of this project. On the other hand, ternary tree is slower in listing all the elements of the population. ArrayLists and LinkedLists work much faster in that since they start from the beginning and display the next as long as there is a next element. The tree has to search the whole tree, meaning that it needs to go all the way to the leafs to discover if a rule is located there. This makes it slower. But in UCS there is never a need to list all the elements so for the reasons explained it is believed that it has good potential to work quite fast if integrated properly with the UCS.

Note that even if linked lists proved here to be slightly faster than ArrayLists they are not suitable because batch-matching process needs to be able to access different batches directly. For example if batch size equals 4, then it needs to access, 0,1,2,3 and 4,5,6,7 and 8,9,10,11 fast. Since linked lists cannot go to a specific index directly and they have to start from the beginning and visit each element until they reach the predefined position, they are not suitable and work very slow.

5 Final Thoughts

5.1 Critical Evaluation

The project involved a lot of coding and spending a lot of time waiting for the experiments to finish. To fully realize the effort that has been put into it you have to take a look at all the source code that is submitted. During the coding phase of the project I phased many time weird results like for example naive being faster than batch matching for some reasons. Most of the times it was something wrong with my code that I had to figure out, but some other times I could not find a reasonable explanation. For example when the population and the matching instances are a relatively small number, then naive appears to be faster. Then as the size increases, batch matching catches up and if the database is very big (that is the case in the normal situations in LCS) it starts really making a big difference. It seems that Java is not the best language for comparing execution times of algorithms, but it is one of the best choices if you have to do with complex data structures as this project did. Having to handle this type of data structures with C for example would be next to impossible. Furthermore, the programmer does not know when Java's Garbage Collector might decide to free up some memory and therefore it could be the reason of the unexpected results since it introduced noise. Of course, you can always use `System.gc()` which basically tells the Java Virtual Machine (JVM) to consider running the garbage collector at that point of the execution. Another thing I noticed about java is that it runs partially in parallel by itself if the code permits it to do so. This was notice during some `exit()` statements I was using for debugging. In some cases if a `print()` statement was followed by an `exit()` statement, the print would not appear. I had to insert a pause code in between to give the chance to print to execute before force quitting the execution. Also the JVM only loads certain libraries when they are first used. This would add a fixed amount of time to the first repetition and none after that. If the amount added is different for two algorithms that would make a difference on their efficiency. Finally, if one algorithm fits in the cache then it will be faster to repeat it than the other algorithm that does not fit. As you can see, the third party parameters are too many but I managed to produce clean outcomes be repeating the tests over and over again to eliminate unwanted behavior.

Number of comparisons was not mentioned too much in the results. The concept I was following for batch matching was to try and figure out ways to decrease the number of comparisons, this would also decrease the running times (increase efficiency). But since timing was the goal, I used the time results instead to display the outcomes. Source files also have the ability to produce the number of comparisons, and `Averager.java` contains methods that calculate the average comparisons of each technique.

5.2 Algorithms Wrap Up

The list that follows indicates the fastest codes, starting from the faster and moving backwards:

1. Multi-level ternary tree
2. Recursive batch matching
3. Adjacent batch matching

All algorithms thought of and implemented in this project are faster than the Naïve matching approach that is used so far in LCS. These three were in my opinion the most important algorithms of the project. Batch matching solutions can be further improved by using Enum types or by sorting the inputs. Multi-level ternary tree is the fastest in inserting and deleting specific elements from the population. Most probably batch matching will never be quite as fast since it is limited by the fact that it uses arrays that are not very efficient. Even my supervisor was impressed by the speed that the multi-level ternary tree provides and it would be very practical technique so we can expect it to be used to speed up systems like UCS in the future.

5.3 Conclusion

Every project is measured based on how successful it is. From results of the interim report and later by the feedback of my supervisor, the project did really good. It moved beyond what was already known and discovered new ways to implement the matching process in classifier systems. The experiment results indicated that the new approaches have good potential to be used in existing learning classifier systems. All approaches proved faster than the Naïve matching that is used so far and each one might be suitable for a different implementation of LCS since there are too many. One implementation of UCS for example may work better with recursive batch matching and other with multi-level ternary tree. Also results of the multi-level ternary tree turned out to give an extreme performance based on what was required for the current implementation of UCS that we had in our hands.

Some of the results that came up while implementing and testing various approaches to speed up the matching process were not expected from the supervisor. These approaches and relevant results are mentioned in the report and explanation follows on the reasons why they occurred. This is another sign in my opinion that designates how the project did and it makes me feel confident that it adds value in its field.

Lastly, if I could make the project again from the beginning I would not have changed much. I am very satisfied with what I did and what the outcome is. What I would have possibly done better is to study more related papers during my interim report. The more you know about what is accomplished in the field so far, the more you can offer since you get more ideas and eventually do not reinvent the wheel.

5.4 Future Development

The first step for the future development of this project is to fully integrate with UCS or any other Learning Classifier System that required this type of computations. After successful integration, some tests have to run in real-time scenarios to indicate the difference between this new approach and newer ones.

This project was composed of two main parts, the batch matching algorithm and the ternary tree implementations. As came out in the end, batch matching proved to be slower than the tree for this kind of usage. But it is still a very important step forward since it uses the same data structures as some classifier systems use but does it in a more clever way. Maybe there is still room for improvement on the idea of batch matching, something I did not think about or never came across.

Multi-dimensional trees are very complex data structures that might be considered in the future. For example R-trees that are mentioned in the History section have the potential to define rules as hyperrectangles in space and inputs as single points. Then if the point is located within a certain hyperrectangle this means that the specific rule matches that input. This would probably require years of study to discover and implement something suitable.

References

- [1] Ester Bernadó-Mansilla and Josep M. Garrell-Guiu, *Accuracy-Based Learning Classifier Systems: Models, Analysis and Applications to Classification Tasks*, 2003.
- [2] X. Llorà and K. Sastry. Fast rule matching for learning classifier systems via vector instructions. In Cattolico [3], pages 1513–1520.
- [3] M. Cattolico, editor. *Genetic and Evolutionary Computation Conference, GECCO 2006, Proceedings, Seattle, Washington, USA, July 8-12, 2006*. ACM, 2006.
- [4] Nils J. Nilsson, *Introduction to Machine Learning*, 1996.
- [5] Sergios Theodorides, Konstantinos Koutroumbas, *Pattern Recognition 3rd edition*, 2006.
- [6] Ricardo Baeza-Yates and Berther Ribeiro-Neto, *Modern Information Retrieval*, 1999.
- [7] S. Wilson. Classifier fitness based on accuracy. *Evolutionary Computation*, 3(2):149–175, 1995.
- [8] J. H. Holland. *Adaptation in Natural and Artificial Systems*. University of Michigan Press, Ann Arbor, MI, 1975.
- [9] J. H. Holland and J. S. Reitman. Cognitive systems based on adaptive algorithms. In *Pattern Directed Inference*.
- [10] D. E. Goldberg. *Genetic algorithms in search, optimization, and machine learning*. Addison-Wesley, Reading, MA, 1989.
- [11] X. Llorà, K. Sastry, and D. Goldberg. The compact classifier system: Motivation, analysis and first results. In *Proceedings of the Congress on Evolutionary Computation*, volume 1, pages 596–603, 2005.
- [12] K. A. De Jong and W. M. Spears. Learning Concept Classification Rules using Genetic Algorithms. In *Proceedings of the Twelfth International Conference on Artificial Intelligence IJCAI-91*, volume 2, pages 651–656. Morgan Kaufmann, 1991.
- [13] Martin V. Butz, Pier Luca Lanzi, Xavier Llorà, Daniele Loiacono, *An Analysis of Matching in Learning Classifier Systems*, 2008.
- [14] M. V. Butz, T. Kovacs, P. L. Lanzi, and S. W. Wilson. Toward a theory of generalization and learning in XCS. *IEEE Transaction on Evolutionary Computation*, 8(1):28–46, Feb. 2004.
- [15] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley, 1989.
- [16] R. Giraldez, J. S. Aguilar-Ruiz, J. Riquelme, *Knowledge-based Fast Evolution for Evolutionary Learning*, 2003.
- [17] R. Quinlan. See5.0 (<http://www.rulequest.com>), 1998-2001.
- [18] S. Ruggieri. Efficient C4.5. *IEEE Transactions on Knowledge and Data Engineering*, 14(2):438–444, April, 2002.
- [19] K. Shim. SIGKDD Explorations. December 2000. Volume 2, Issue 2.
- [20] Sartaj Sahni, *Data structures, algorithms, and applications in Java 2nd edition*, 2005.
- [21] Tim Kovacs, *Object Oriented Programming with Java* (University of Bristol), 2010.
- [22] J. T. Robinson. The K–D–B–Tree: A search structure for large multidimensional dynamic indexes. In Y. E. Lien, editor, *Proceedings of the 1981 ACM SIGMOD International Conference on Management of Data*, Ann Arbor, Michigan, April 29 - May 1, 1981, pages 10–18. ACM Press, 1981.

- [23] A. Henrich, H.-W. Six, and P. Widmayer. The LSD tree: Spatial access to multidimensional point and nonpoint objects. In P. M. G. Apers and G. Wiederhold, editors, *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, August 22-25, 1989, Amsterdam, The Netherlands, pages 45–53. Morgan Kaufmann, 1989.
- [24] M. Freeston. A general solution of the n-dimensional b-tree problem. In M. J. Carey and D. A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data*, San Jose, California, May 22-25, 1995, pages 80–91. ACM Press, 1995.
- [25] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, 1979.
- [26] A. Guttman. R-Trees : A dynamic index structure for spatial searching. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages pp. 47–57, 1984.
- [27] J L Bentley and J H Fildeman, Data Structures for Range Searchmg, *ComputZng Surveys* 11, 4 (December 1979) 397-409.
- [28] J L Bentley, D F. Stanat and E H Wilhams, Jr, The complexity of fixed- radius near neighbor searchmg, *hzf Proc L&t* 6, 6 (December 1977) 209 212.
- [29] A Guttman and M Stonebraker, Usmg a Relational Database Management System for Computer &ded Design Data, *IEEE LMubase IZkgineerkng* 5, 2 (June 1982).
- [30] R A Fmkel and J L Bentley, Quad Trees - A Data Structurefor Retneval on Co oslte Keys, *Acta Informutica* 4, (1974) 1-9.
- [31] R Bayer and E McCreight, Orgamzation and Maintenance of Large Ordered Indices, *Proc 1970 ACM-SIGFTDET Workshop on Data lkscrzphon and Access*, Houston, Texas, Nov. 1970, 107-141.
- [32] Java v6 API, BitSet class.
- [33] Akhil Gupta, Rajat Raina, An Intelligent RoboSoccer Client, <http://www.clsp.jhu.edu/people/rahulr/FAM/RAJAT/robocup/proposal.html>, 2000.
- [34] Ester Bernado-Mansilla, Josep M. Garrell-Guiuand, *Accuracy-Based Learning Classifier Systems: Models, Analysis and Applications to Classification Tasks*, 2003.
- [35] Gavin Brown, Tim Kovacs, James Marshall, UCSpv: Principled Voting in UCS Rule Populations, July 2007.
- [36] F. J. M. Barning, On Pythagorean and quasi-Pythagorean triangles and a generation process with the help of unimodular matrices. (Dutch) *Math. Centrum Amsterdam Afd. Zuivere Wisk. ZW-001* (1963).
- [37] A. Hall, Geneology of Pythagorean Triads, *Math. Gazette* 54, No. 390 (1970), 377-379

Appendix A: Variable Batch Size Matching

Population4.java

```

import java.util.ArrayList;
import java.util.Collections;

/**
 * Same as Population3.java but not use only one M+, IS and DS.
 * The way to do it is to calculate them on every iteration for the
 * inputs of the iteration.
 *
 * @author Yiannis Tsentas
 * @version August 2010
 */
public class Population4 {
    private ArrayList<String> P = new ArrayList<String>();
    private ArrayList<String> INPUT = new ArrayList<String>();
    private ArrayList<ArrayList<String>> M = new ArrayList<ArrayList<String>>();
    private ArrayList<String> Mplus = new ArrayList<String>();
    private ArrayList<Integer> IS = new ArrayList<Integer>();
    private ArrayList<Integer> DS = new ArrayList<Integer>();
    private static int STRLEN;
    private int comparisons;
    private int BATCH_SIZE; /**most of the variables here are used so that
calculations are executed only once*/
    private int LAST_ROUND;
    private int INP_SIZE;
    private int LAST_SET_SIZE;
    private boolean ONE_INP_LEFT;

    public Population4(int bs) {
        P.clear();
        INPUT.clear();
        M.clear();
        Mplus.clear();
        IS.clear();
        DS.clear();
        STRLEN = 0;
        comparisons = 0;
        BATCH_SIZE = bs; /**@@@@@@@@@@@@@@@@ CHANGE THIS VALUE TO WORK WITH OTHER
BATCH SIZE @@@@@@@@@@@@@@@@@@*/
        LAST_ROUND = 0;
        INP_SIZE = 0;
        LAST_SET_SIZE = 0;
        ONE_INP_LEFT = false;
    }

    public void addRule(String rule) {
        P.add(rule);
    }

    public void addInput(String input) {
        INPUT.add(input);
    }

    public void setBatchSize(int b) {
        BATCH_SIZE = b;
    }

    public void printSets() {
        System.out.println("P: "+P);
        for (int i=0; i<INP_SIZE; i++) {
            System.out.println("M["+i+"]("+INPUT.get(i)+"): "+M.get(i));
        }
    }

    public int getComparisons() {
        return comparisons;
    }

    public void sortInputs() {
        Collections.sort(INPUT);
    }

    /** The standard (non-batch) way of computing a match set */
    public void makeMatchSets() {
        initializations();

        int index=0;
        for (String in : INPUT) {
            boolean matches;
            for (String r : P) {

```

```

        matches = true;
        for (int i=0; i<STRLEN; i++) {
            if (mismatches(in, r, i)) {
                matches = false;
                break;
            }
        }
        if (matches) M.get(index).add(r);
    }
    index++;
}

/** A minimal batch match set computation: computes [M1] and [M2] for two inputs
*/
public void makeBatchMatchSets() {
    initializations();

    for (int j=0; j<INP_SIZE; j+=BATCH_SIZE) {
        Mplus.clear();
        IS.clear();
        DS.clear();
        int start = j;

        //computes IS-DS for one set of inputs
        computeISDS(start);
        //uncomment for testing
        //System.out.println("IS: "+IS);
        //System.out.println("DS: "+DS);

        for (String r : P)
            findM1AndM1Plus(r, start); //for one set of inputs

        //uncomment for testing
        //System.out.println("M+: "+Mplus);

        for (String r : Mplus)
            findMs(r, start);
    }
}

/**
 * Outputs all possible permutations or rules for the population (P) given a size
 */
private ArrayList<String> allPermutations(int len) {
    ArrayList<Character> c = new ArrayList<Character>();

    c.add('0');
    c.add('1');
    c.add('#');

    int size = c.size();
    int permutations = (int) Math.pow(size, len);
    char[][] entry = new char[permutations][len];
    int x, t2, p1, a1, p2, y;

    for (x = 0, y = len-1; x < len; x++,y--) {
        t2 = (int) Math.pow(size, x);
        for (p1 = 0; p1 < permutations; p1++) {
            for (a1 = 0; a1 < size; a1++) {
                for (p2 = 0; p2 < t2; p2++) {
                    entry[p1][y] = c.get(a1);
                    p1++;
                }
            }
        }
    }

    ArrayList<String> result = new ArrayList<String>();
    for (char[] permutation : entry) {
        result.add(new String(permutation));
    }
    return result;
}

private void resetVariables() {
    STRLEN = INPUT.get(0).length();
    INP_SIZE = INPUT.size();
    LAST_SET_SIZE = INP_SIZE % BATCH_SIZE;
    LAST_ROUND = INP_SIZE - (LAST_SET_SIZE); //calculate value of index in last
round
    ONE_INP_LEFT = (LAST_SET_SIZE == 1);
    M.clear();
    //Mplus.clear();
    //IS.clear(); //only required in batch matching
    //DS.clear();
    comparisons=0;
}

```

```

    }

    public void clearValues() {
        P.clear();
        INPUT.clear();
    }

    private void initializations() {
        //first clear variables and then initialize
        resetVariables();
        for (int i=0; i<INP_SIZE; i+=BATCH_SIZE) {
            int tempBATCH_SIZE = BATCH_SIZE;
            if (i==LAST_ROUND) {
                if (ONE_INP_LEFT) { //only a single input at the end
                    M.add(new ArrayList<String>());
                    return;
                }
                tempBATCH_SIZE = LAST_SET_SIZE;    //how many inputs left for the last
set
            }
            for (int j=0; j<tempBATCH_SIZE; j++)
                M.add(new ArrayList<String>());
        }
    }

    private void findM1AndM1Plus(String rule, int start) {
        int i=start;
        boolean match = true;

        if (i==LAST_ROUND) {
            if (ONE_INP_LEFT) { //in case only one input left
                for (int j=0; j<STRLEN; j++) {
                    match = true;
                    if (mismatches(INPUT.get(i), rule, j)) {
                        match = false;
                        break;
                    }
                }
                if (match) M.get(i).add(rule);
                return;
            }
        }

        final String currentInput = INPUT.get(i);
        for (int j : IS) {
            if (mismatches(currentInput, rule, j))
                return;
        }
        Mplus.add(rule);
        for (int j : DS) {
            if (mismatches(currentInput, rule, j))
                return;
        }
        M.get(i).add(rule);
    }

    public void findMs(String r, int start) {
        int i = start;
        int tempBATCH_SIZE = BATCH_SIZE;

        if (i==LAST_ROUND) {
            if (ONE_INP_LEFT) return; //this scenario is dealt with in findM1AndM1Plus
            tempBATCH_SIZE = LAST_SET_SIZE; //change BATCH SIZE to match the number
of inputs in the last set
        }

        boolean[] match = new boolean[tempBATCH_SIZE];
        for (int k=1; k<tempBATCH_SIZE; k++) {
            match[k] = true;
            for (int j : DS) {
                if (mismatches(INPUT.get(i+k), r, j)) {
                    match[k] = false;
                    break;
                }
            }
            if (match[k]) M.get(i+k).add(r);
        }
    }

    private boolean mismatches(String input, String rule, int index) {
        comparisons++;
        if (rule.charAt(index) != '#' && rule.charAt(index) != input.charAt(index))
            return true;
        return false;
    }

    private void computeISDS(int start) {

```



```

        final int i=start;
        int tempBATCH_SIZE = BATCH_SIZE;

        if (i==LAST_ROUND) {
            if (ONE_INP_LEFT) return; //this scenario is dealt with in findM1AndM1Plus
            tempBATCH_SIZE = LAST_SET_SIZE; //change BATCH SIZE to match the number
of inputs in the last set
        }

        boolean[] match = new boolean[STRLEN];
        final String firstInput = INPUT.get(i);
        for (int k=0; k<STRLEN; k++) {
            match[k] = true;
            final char temp = firstInput.charAt(k);
            for (int iIndex=i+1; iIndex<i+tempBATCH_SIZE; iIndex++) { //iIndex the
input index (which input)
                if (temp != INPUT.get(iIndex).charAt(k)) {
                    match[k] = false;
                    break;
                }
            }
        }
        for (int k=0; k<STRLEN; k++) {
            if (match[k])
                IS.add(k);
            else
                DS.add(k);
        }
    }

    private void test1() {
        //sortInputs(); //uncomment to sort inputs
        System.out.println("Non Batch:");
        makeMatchSets();
        printSets();
        System.out.println("Comparisons: "+getComparisons());
        System.out.println("Batch:");
        makeBatchMatchSets();
        printSets();
        System.out.println("Comparisons: "+getComparisons());
    }

    private void repeatTest() {
        INPUT.add("0100111");
        INPUT.add("0110001");
        INPUT.add("1101001");
        INPUT.add("0111111");
        INPUT.add("0100110");
        INPUT.add("1110000");
        INPUT.add("1011100");
        INPUT.add("1100110");

        /*
        INPUT.add("0000011");
        INPUT.add("0000011");
        INPUT.add("0000011");
        INPUT.add("0000011");
        INPUT.add("0000011");
        INPUT.add("0000011");
        INPUT.add("0000011");
        INPUT.add("0000011");
        */
        //P = allPermutations(input.get(0).length());

        P.add("000#1#1");
        P.add("01010##");
        P.add("0#0#1#0");
        P.add("100##01");
        P.add("1#1#010");
        P.add("1#00#0#");
        P.add("#001##1");
        P.add("#10#101");
        P.add("##01#01");
        P.add("####0#");

        for (int i=0; i<1; i++) {
            test1();
        }
    }

    public static void main(String[] args) {
        Population4 bTest = new Population4(4);

        bTest.repeatTest();
    }

```

Appendix B: Experiment.java

```
import java.util.Random;
import java.util.ArrayList;
import java.text.DecimalFormat;

public class Experiment {
    static Random rand = new Random();

    /**@@@ CHANGE VALUES FOR DIFFERENT TEST CASES @@@@*/
    static final int POP_SIZE = 2500;
    static final float HASH_PROB = 0.50f;
    static final int NUM_INPUTS = 2000;
    static final int NUM_REPS = 50;
    static final int DIMENSIONS = 100;
    static final int BATCH_SIZE = 20;

    public static String randomCondition() {
        StringBuilder buff = new StringBuilder(DIMENSIONS);
        for (int i=0; i<DIMENSIONS; i++) {
            if (rand.nextFloat() < HASH_PROB)
                buff.append('#');
            else {
                if (rand.nextBoolean())
                    buff.append('0');
                else
                    buff.append('1');
            }
        }
        return buff.toString();
    }

    public static String randomInput() {
        StringBuilder buff = new StringBuilder(DIMENSIONS);
        for (int i=0; i<DIMENSIONS; i++) {
            if (rand.nextBoolean())
                buff.append('0');
            else
                buff.append('1');
        }
        return buff.toString();
    }

    public static void repeatNonBatch(Population5 batch) {
        long startTime;
        Averager nonBatchAvg = new Averager();
        //batch.sortInputs();
        //batch.printSets();
        for (int rep=0; rep<NUM_REPS; rep++) {
            System.gc();
            startTime = System.currentTimeMillis();
            batch.makeMatchSets();
            nonBatchAvg.add(System.currentTimeMillis() - startTime);
            nonBatchAvg.addComparisons(batch.getComparisons());
            //uncomment next line if you want differencnt inputs/rules for every
iteration
            //addInputsRules(batch);
        }
        System.out.print("Naive matching (");
        System.out.print("Avg Time: "+nonBatchAvg.average());
        DecimalFormat threeDec = new DecimalFormat("0.000");
        System.out.print(", St.Deviation: " + threeDec.format(nonBatchAvg.stdDev()));
        System.out.print(", Avg Comparisons: " + nonBatchAvg.averageComparisons());
        System.out.println(")");
    };

    public static void repeatBatch(Population5 batch) {
        long startTime;
        Averager batchAvg = new Averager();
        //batch.sortInputs();
        //batch.printSets();
        for (int rep=0; rep<NUM_REPS; rep++) {
            System.gc();
            startTime = System.currentTimeMillis();
            batch.makeBatchMatchSets();
            batchAvg.add(System.currentTimeMillis() - startTime);
            batchAvg.addComparisons(batch.getComparisons());
            //uncomment next line if you want differencnt inputs/rules for every
iteration
            //addInputsRules(batch);
        }
        System.out.print("Batch matching (");
        System.out.print("Avg Time: "+batchAvg.average());
        DecimalFormat threeDec = new DecimalFormat("0.000");
        System.out.print(", St.Deviation: " + threeDec.format(batchAvg.stdDev()));
```

```
        System.out.print(", Avg Comparisons: " + batchAvg.averageComparisons());
        System.out.println("");
    }

    private static void addInputsRules(Population5 batch) {
        batch.clearValues();
        for (int i=0; i< NUM_INPUTS; i++)
            //batch.addInput("1111110000");
            batch.addInput(randomInput());

        for (int i=0; i< POP_SIZE; i++)
            batch.addRule(randomCondition());
    }

    public static void test1() {
        Population5 batch = new Population5(BATCH_SIZE);

        batch.setBatchSize(BATCH_SIZE);
        addInputsRules(batch);
        for (int i=0; i<2; i++) {
            System.out.println("Test for: \t"+NUM_REPS+" repetitions");
            System.out.println("\t\t"+NUM_INPUTS+" inputs");
            System.out.println("\t\t"+DIMENSIONS+" input size");
            System.out.println("\t\t"+POP_SIZE+" Population5 size");
            System.out.println("\t\t"+HASH_PROB*100+"% hash probability");

            repeatNonBatch(batch);
            repeatBatch(batch);
        }
    }

    public static void main(String[] args) {
        test1();
    }
}
```

Appendix C: Averager.java

```
import java.util.ArrayList;
import java.util.Random;
import java.util.Collections;

/** A naive, memory-inefficient version which stores all the
 * data and calculates the average and standard deviation
 * in a batch
 *
 * should be made generic
 */
public class Averager {
    private ArrayList<Long> values;
    private ArrayList<Integer> comparisons;

    public Averager() {
        values = new ArrayList<Long>();
        comparisons = new ArrayList<Integer>();
    };
    public Averager(int size) {
        values = new ArrayList<Long>(size);
        comparisons = new ArrayList<Integer>(size);
    }
    public void add(long next) {
        values.add(next);
    }
    public void addComparisons(int comp) {
        comparisons.add(comp);
    }
    public double average() {
        long sum=0;
        for (long current: values)
            sum += current;
        return sum / (double) values.size();
    }

    public double averageComparisons() {
        long sum=0;
        for (int i=0; i<comparisons.size(); i++)
            sum += comparisons.get(i);

        return sum / (double) values.size();
    }

    /** Sample variance is the sum of the squared deviations from the average */
    public double variance() {
        double avg = average();
        double sumOfSquaredDiff = 0.0;
        for (long current: values) {
            double difference = current - avg;
            sumOfSquaredDiff += difference * difference;
        }
        return sumOfSquaredDiff;
    }
    public double stdDev() {
        return Math.sqrt(variance() / values.size());
    }
    public int size() {
        return values.size();
    };
    public void clear() {
        values.clear();
    }

    /** @return the t statistic for the averages of two samples of equal size drawn
     * from distributions with assumed equal variance.
     * The lower the returned value, the less likely the averages are drawn from the
     same
     * distribution.
     * The usual thresholds for statistical significance are 0.1, 0.05 and 0.01 */
    public double equalSizeEqualVarianceTTest(Averager avg2) {
        return equalSizeEqualVarianceTTestDetails(this.average(), avg2.average(),
                                                    this.variance(), avg2.variance(),
                                                    this.size());
    }

    private double equalSizeEqualVarianceTTestDetails(double avg1, double avg2, double
    var1, double var2, double size) {
        double diffOfAveragers = avg1 - avg2;
        //System.out.println("diff of averagers = " + diffOfAveragers);
        double pooledStdDev = Math.sqrt((var1 + var2) / 2.0);
        //System.out.println("pooled std. dev. = " + pooledStdDev);
        double denominator = pooledStdDev * Math.sqrt(2.0/size);
        //System.out.println("denominator = " + denominator);
    }
}
```

```

    return diffOfAveragers / denominator;
}

/** Test equalSizeEqualVarianceTTestDetails */
public static void testTTest1() {
    Averager avg = new Averager();
    System.out.println(avg.equalSizeEqualVarianceTTestDetails(100, 101,
                                                                5, 6,
                                                                1000)
                      );
}

/** Generate some averages by sampling and compare them */
public static void testTTest2() {
    Random rand = new Random();
    Averager avg1 = new Averager();
    Averager avg2 = new Averager();
    final int SAMPLE_SIZE = 10;
    final int DIFFERENCE = 100000; // difference in means

    // we generate ints here rather than longs so that we
    // can add something to one of them without overflowing
    for (int i=0; i<SAMPLE_SIZE; i++) {
        avg1.add(rand.nextInt() + DIFFERENCE);
        avg2.add(rand.nextInt());
    }
    System.out.println("t = " + avg1.equalSizeEqualVarianceTTest(avg2));
}

// a simple test of the calculations
public static void statsTest() {
    Averager avg = new Averager();
    avg.add(2);
    avg.add(4);
    avg.add(6);
    System.out.println("average " + avg.average()); // sqrt(8/3) = 1.6329...
    System.out.println("size " + avg.size());
    System.out.println("variance " + avg.variance());
    System.out.println("stdDev " + avg.stdDev());
}

/** A worked example from http://en.wikipedia.org/wiki/T-test
 * which unfortunately used real values so the result is different. */
public static void statsTest2() {
    Averager avg1 = new Averager();
    Averager avg2 = new Averager();
    long[] arr1 = {3002, 2999, 3011, 2997, 3001, 2999};
    long[] arr2 = {2989, 2993, 2972, 2998, 3002, 2998};
    for (int i=0; i<arr1.length; i++) {
        avg1.add(arr1[i]);
        avg2.add(arr2[i]);
    }
    System.out.println("t = " + avg1.equalSizeEqualVarianceTTest(avg2));
    System.out.println("should be 0.084");
}

public static void main(String[] args) {
    statsTest2();
}
}

```

Appendix D: TTree.java of UCS

```

package ucs;

import java.util.*;
import dataprocessing.Example;

/**
 * Ternary tree implementation
 *
 * Adjusted and extended so that it can be integrated into the UCS. Keeps a
 * uniformity over the methods of Population.java
 *
 * @author Yiannis Tsentas
 * @version August 2010
 */
public class TTree
{
    // instance variables - replace the example below with your own
    private Node root;
    private double HASH;
    private double ONE;
    private double ZERO;
    private ArrayList correctSet;
    private String macros = "";
    private long SIZE;
    private HashMap macroFitnesses = new HashMap();

    /**
     * Constructor for objects of class TTree
     */
    public TTree()
    {
        root = new Node();
        HASH = -1.0;
        ONE = 1.0;
        ZERO = 0.0;
        correctSet = new ArrayList();
        SIZE = 0;
    }

    public void clear() {
        root = null;
        correctSet.clear();
        SIZE = 0;
    }

    public Node getRoot() {
        return root;
    }

    public ArrayList getCorrectSet() {
        return correctSet;
    }

    public void clearCorrectSet() {
        correctSet.clear();
    }

    public long size() {
        return this.SIZE;
    }

    /**
     * Prints out every unique individual in this Population.
     */
    public void printMacros()
    {
        System.out.println(macros);
    }

    //////////////////////////////////////
    //////////////////////////////////////
    //////////////////////////////////////
    //////////////////////////////////////

    /**
     * Add the Indiv in the tree
     */
    public void add(Indiv ind)
    {
        Node current = root;
        for (int i=0; i<ind.condition.values.length; i++) {

```

```

        if (ind.condition.values[i] == ONE) {
            if (current.getLeft()==null)
                current.setLeft(new Node(current));
            current = current.getLeft();
        }
        else if (ind.condition.values[i] == HASH) {
            if (current.getMiddle()==null)
                current.setMiddle(new Node(current));
            current = current.getMiddle();
        }
        else {
            if (current.getRight()==null)
                current.setRight(new Node(current));
            current = current.getRight();
        }
    }
    if (current.getValue() == null)    { //add if not added already
        //System.out.println("POP SIZE: "+this.SIZE+", ADDING NEW");
        current.setValue(ind);
    }
    else {
        //System.out.println("POP SIZE: "+this.SIZE+", ADDING DUPLICATE");
        current.getValue().numerosity++;
    }
    this.SIZE++;
}

////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////

/**
 * Find if the rule exists in the tree
 *
 * @return returns true the rule is found and false if it was not
 */
public boolean find(double[] input) {
    Node current = root;
    for (int i=0; i<input.length; i++) {
        if (input[i] == ONE) {
            if (current.getLeft() == null)
                return false;
            current = current.getLeft();
        }
        else if (input[i] == HASH) {
            if (current.getMiddle() == null)
                return false;
            current = current.getMiddle();
        }
        else {
            if (current.getRight() == null)
                return false;
            current = current.getRight();
        }
    }
    return true;
}

////////////////////////////////////
////////////////////////////////////
////////////////////////////////////
////////////////////////////////////

/**
 * Removes a given rule from the tree
 *
 * NOT TESTED - NOT USED ANYWHERE
 *
 * @return returns true if the removal was successful and false if it wasn't
 */
public boolean remove(Indiv ind) {
    Node current = root;
    Node previous = null;
    //find
    for (int i=0; i<ind.condition.values.length; i++) {
        if (ind.condition.values[i] == ONE) {
            if (current.getLeft() == null)
                return false;
            previous = current;
            current = current.getLeft();
        }
    }
}

```



```

        this.SIZE--;
    }

    //////////////////////////////////////
    //////////////////////////////////////
    //////////////////////////////////////
    //////////////////////////////////////

    /**
     * Returns the match set (rules that match input) of the given input.
     * It searches through the tree recursively
     * also computes correctSet
     *
     * @param e      Example => input
     * @return matchSet return the set of rules that match the input
     */
    public ArrayList<Indiv> getMatchSet(Example e) { return getMatchSet(this.root,
e.inputs, new ArrayList<Indiv>(), e.target); }
    private ArrayList<Indiv> getMatchSet(Node current, double[] input,
ArrayList<Indiv> matchSet, double target) {
        Indiv ind = current.getValue();
        if (ind != null) {
            matchSet.add(ind);
            if( ind.test(target) ) correctSet.add(ind);
            return null;
        }
        double c = input[0];
        final int newSize = input.length-1;
        double[] newInput = new double[newSize];
        if (c==ONE){
            Node left = current.getLeft();
            if (left!=null) {
                System.arraycopy(input, 1, newInput, 0, newSize);
                getMatchSet(left, newInput, matchSet, target);
            }
            Node middle = current.getMiddle();
            if (middle!=null) {
                System.arraycopy(input, 1, newInput, 0, newSize);
                getMatchSet(middle, newInput, matchSet, target);
            }
        }
        else {
            Node right = current.getRight();
            if (right!=null) {
                System.arraycopy(input, 1, newInput, 0, newSize);
                getMatchSet(right, newInput, matchSet, target);
            }
            Node middle = current.getMiddle();
            if (middle!=null) {
                System.arraycopy(input, 1, newInput, 0, newSize);
                getMatchSet(middle, newInput, matchSet, target);
            }
        }
        return matchSet;
    }

    //////////////////////////////////////
    //////////////////////////////////////
    //////////////////////////////////////
    //////////////////////////////////////

    /**
     * Returns the the whole population; all the rules in the tree.
     *
     * @return values      return the set of rules
     */
    public ArrayList<Indiv> getAll() { return getAll(this.root, new
ArrayList<Indiv>()); }
    private ArrayList<Indiv> getAll(Node current, ArrayList<Indiv> values) {
        Indiv ind = current.getValue();
        if (ind != null) {
            values.add(ind);
            return null;
        }
        final Node right = current.getRight();
        if (right!=null)
            getAll(right, values);
        final Node middle = current.getMiddle();
        if (middle!=null)
            getAll(middle, values);
        final Node left = current.getLeft();
        if (left!=null)

```

```

        getAll(left, values);
    }
    return values;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

/**
 * Calculates the average accuracy of all individuals in this population.
 * @return A double providing the average accuracy.
 */
public double averageAccuracy() { return averageAccuracy(this.root, new
Double(0)); }
private double averageAccuracy(Node current, double total)
{
    Indiv ind = current.getValue();
    if (ind != null) {
        total += ind.accuracy;
        return 0.0;
    }
    final Node right = current.getRight();
    if (right!=null)
        averageAccuracy(right, total);
    final Node middle = current.getMiddle();
    if (middle!=null)
        averageAccuracy(middle, total);
    final Node left = current.getLeft();
    if (left!=null)
        averageAccuracy(left, total);

    return total/this.SIZE;
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

/**
 * Prints out every Individual in this Population.
 */
public void print() { print(this.root, new ArrayList<Indiv>()); }
private void print(Node current, ArrayList<Indiv> values) {
    Indiv ind = current.getValue();
    if (ind != null) {
        String s = ind.toString()+" , Accuracy: "+ind.accuracy+" , Experience:
"+ind.numMatches;
        System.out.println(s);
        return;
    }
    final Node right = current.getRight();
    if (right!=null)
        getAll(right, values);
    final Node middle = current.getMiddle();
    if (middle!=null)
        getAll(middle, values);
    final Node left = current.getLeft();
    if (left!=null)
        getAll(left, values);
}

////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

/**
 * Counts the number of macroclassifiers in this population.
 * @return An integer count of the number of macroclassifiers.
 */
public int numMacroClassifiers()
{
    boolean print = false;

    //macroFitnesses was move to global variable

```

```

        macroFitnesses.clear();
        HashSet macroclassifierList = getMacroClassifiers(this.root, new
HashSet()); //fills both macroclassifierList and macroFitnesses

        int k=0;
        Iterator myIterator = macroclassifierList.iterator();
        macros="";
        while (myIterator.hasNext())
        {
            //I have no idea why calling toString here causes a runtime error....
            //Indiv ind = (Indiv)myIterator.next();
            //String x = ind.toString();
            //macros += x + "\t" + ind.accuracy + "\t" + ind.numMatches + "\n";
            String x = (String)myIterator.next();
            macros += x + "\n";
            /*
            if (print)
            {
                System.out.print((k++)+" : "+x+" - ");
                ArrayList y = (ArrayList)macroFitnesses.get(x);
                Iterator yiter = y.iterator();
                while (yiter.hasNext())
                {
                    System.out.print(yiter.next()+" ");
                }
                System.out.println();
            }
            */
        }

        return macroclassifierList.size();
    }
    //used by numMacroClassifiers() to go through tree and collect macroclassifiers
    private HashSet getMacroClassifiers(Node current, HashSet macroclassifierList) {
        Indiv ind = current.getValue();
        if (ind != null) {
            String x = ind.toString();

            //collect a statistic to print later
            if( !macroclassifierList.add(x) )
            {
                ArrayList y = (ArrayList)macroFitnesses.get(x);
                //y.add(new Double(ind.fitness()));
                y.add(""+ind.lastTimeThisWasInTheGA);
            }
            else
            {
                ArrayList y = new ArrayList();
                //y.add(new Double(ind.fitness()));
                y.add(""+ind.lastTimeThisWasInTheGA);
                macroFitnesses.put(x, y);
            }
            return null;
        }

        final Node left = current.getLeft();
        if (left!=null)
            getMacroClassifiers(left, macroclassifierList);
        final Node middle = current.getMiddle();
        if (middle!=null)
            getMacroClassifiers(middle, macroclassifierList);
        final Node right = current.getRight();
        if (right!=null)
            getMacroClassifiers(right, macroclassifierList);

        return macroclassifierList;
    }
}

```