Abstract

The LCS (Learning Classifier System) is a special approach which has been broadly used on different machine learning problems (Kovacs, January, 2002). This system is a rule ("If-Then Rules") based system. At any one time, there are a large number of rules (hundreds or thousands of rules) stored in it. Those rules are usually evolved by genetic algorithm. Based on this, each time LCS receives an input for training or testing, the input would be matched against all the rules inside the system. Compared with most machine learning methods, LCS has a very sow run-time. And the most time consumption part inside it is "If–Then Rules Matching" which turns out to be a big potential problem while processing inputs.

In this study, the problem would be optimised in different directions using different technics. The best methods (Input Tree+ Ternary Tree) found could even get a stable acceleration of more than 10 times faster than the naive matching (this acceleration is based on the chosen experiments, could be even faster in other situations). We would come up with five main approaches: "HashMatch", "Recursion Optimisation", "Object Pool", "Parallel TTreeN" and "Input Tree + TTreeN". Except for the first method, the other methods would be based a new optimisation approach which is called Ternary Tree (Will be discussed in details in section 3.1). Among them, some methods are quite useful whereas some methods have their own drawbacks, but all of the features would be discussed in this thesis. While discussing the implementation of them, "Input Tree + TTreeN", as the most valuable approach found would be discussed in details. And in section 4, some designed experiments on those methods would be shown to discuss their advantages and drawbacks.

At the end, some possible future works and a conclusion on this issue would be stated.

Keywords

Learning Classification Systems (LCS), Accuracy-Based Systems (XCS), "If-Then rules", High Performance Computing (HPC), Parallel Computing, Object Pool

Contents

1. Introduction

Machine learning is an important area in computer science which focuses on optimizing criterion using existed experience (Alpaydin, 2004). For processing different types of tasks, there are amount of approaches in this area, such as Neural Network, Support Vector Machine and so on. Among them, there is a method called Learning Classifiers Systems (LCS) which is aimed at solving classification and reinforcement learning problems.

1.1 Aims and Objectives

LCS is a stable method in the area of machine learning which has competitive performance. However, it has some drawbacks on its run-time efficiency. There is a part called “If-Then Rules Matching” (it will be discussed in details in the Section2) in the system which responds for pattern matching inputs against existed rules in the system. This part would be executed more than ten-thousand times during a whole process of a normal problem. But unfortunately the efficiency of this part is not as good as needed. This is a big limitation of this system. To solve this problem, this thesis will focus on analysing the problem and trying to make a breakthrough on this point.

1.1.1 Research on optimisation

Some analysis about the cause of this problem would be expressed to show some potential strategies that we could take advantage of to increase the efficiency of this process.

1.1.2 Implementations for optimisation

Several implementations would be done to optimize the “If-Then Rules Matching” part of the system in different directions. Those methods would be described in details in section 3.

1.2 Structure

There would be five sections followed by this “introduction section” totally.

In section 2, some basic background knowledge of this area would be described in details to prepare for the understanding my methods.

In section 3, which is the most important section, I will describe all my methods in details. For each of them I would explain their own related backgrounds, theories, algorithms.

In section 4, some experiment results of the methods described in section 3 will be shown. After each result, the discussion and summary of it would be made and analysed.

In section 5 and 6, a brief conclusion would be made for this project would be come up with. Also some possible future works would be shown.

2. Background

In this section, the basic backgrounds in this area would be introduced in details.

2.1 Genetic Algorithms

Genetic algorithm is an important sub-component for LCS. In this section it will be briefly introduced.

In fact, GA was firstly come up with in the same paper with LCS in 1975 by John Holland (Holland, 1975). And then it was improved by Holland with its full theory in 1988 in cooperate with Goldberg and Booker (L.B. Booker, September 1989). From that point of time, GA has become an individual filed which is not related to LCS anymore, so this thesis would not come up with it in details. Only the basic algorithm used in LCS will be described as follows:

```
While(generation is less than demand)
{
  //population is rules (classifiers) in LCS.
  Select parents from population according to their fitness;
  Do crossover between selected parents and get children;
  Do mutation to the children;
  Replace some of current populations with new children;
  Generation ++;
}
```

Algorithm 2.1-GA

2.2 Reinforcement Learning

The other basic component of LCS is Reinforcement learning algorithm.

RL is an important subfield of AI which contains all the algorithms which could be used to solve RL tasks. A typical RL task aims to learn a concept without any pre-classified examples in training set, and the only thing it needs is the rewards (or punishments) from choosing an action (Kovacs, 2002-06-04). In LCS, a famous RL algorithm called “Q-learning” is used.

2.3 Fundamental about LCS and XCS

In this section, some basic concepts of LCS and XCS would be introduced.

2.3.1 Some history of LCS

The Learning Classifiers Systems (LCS) is first introduced by John Holland (Holland, 1975)in 1975. However, at that time, LCS was not as famous as Holland’s another invitation “Genetic Algorithms (GAs)” although GAs was only a sub-part of LCS (Sigaud & Wilson, 2007).

Several years later, the first implementation of LCS was presented by Judith Reitman and him (Holland & Reitman, 1977) in 1978. And then Holland himself modified the structure of LCS and gave an example of what a standard LCS look like (Bull, 2004).

From that time, the standard LCS has been formed. However, the framework of it is somehow complex and not so easy to be used.

Later, many researchers started to show their interests on this system and soon simplified and modified it in several directions.

And now there are mainly three types of LCS which are famous and used widely:

Types of LCS	Typical Example System
Strength-based systems	ZCS
Accuracy-based systems	XCS
Anticipation-based systems	ALCS

Table 2.1-Types of LCS

In Holland's LCS, the inserted information from environments (outside) is called inputs, the inputs are usually encoded by the binary alphabet {0, 1} while the rules are represented by the ternary alphabet {0, 1, #}. They look like this:

Input (Binary Code)	10101101
Rule (Ternary Code)	IF 1#10##01 THEN class 1

Table 2.2-Inputs and Rules

2.3.2 Inputs

The inputs from environment are usually encoded as binary alphabet {0, 1}, so the length (number of bits) of a single input is decided by how you encode the problem. Recently, more problems are based on real-valued inputs instead of binary ones. However, the procedures of them are similar to each other. In this thesis, examples are based on binary inputs.

Based on this, once the system has been initialised to solve a specific task, the length of input should be decided directly and never be changed.

In practise, the inputs are usually known at the beginning and stored in a file. This file would be iteratively used while training the rule database, so an important feature could be noticed here is that due to this fact, every piece of input in the file would be used several times. This feature is important because this could be a point to be optimised to avoid duplicate inserting inputs.

2.3.3 Rules (classifiers)

Since LCS is a rule-based system, rules are the most important things in the system. In LCS, the rules are also called classifiers because they are used to judge which class an input should belong to.

As can be seen from table 2-2, a simplified rule has two key parts which are "If" part and "Then" part, once the "If" part was matched by an input, the system will output its "Then" part. It is the reason that the rule in LCS is also called "If-Then rules".

A rule is usually an individual object (structure in C++) which contains its own field of attributes as follows:

Attributes	Explanation
Condition (If part)	Condition is the part which is used to do match against inputs. It is usually represented by ternary alphabet {0, 1, #}. "#" here stands for "don't care", if a bit in the condition is "#", which means this bit would match no matter "1" and "0" at the same bit of input.
Action (Then part)	Action stands for the action (e.g. a classification) which would be returned once condition part matches
Fitness	The fitness of this classifier which is propotational to the rewards from outside in LCS

Table 2.3-Attributes in rule of XCS

2.4 Process of LCS

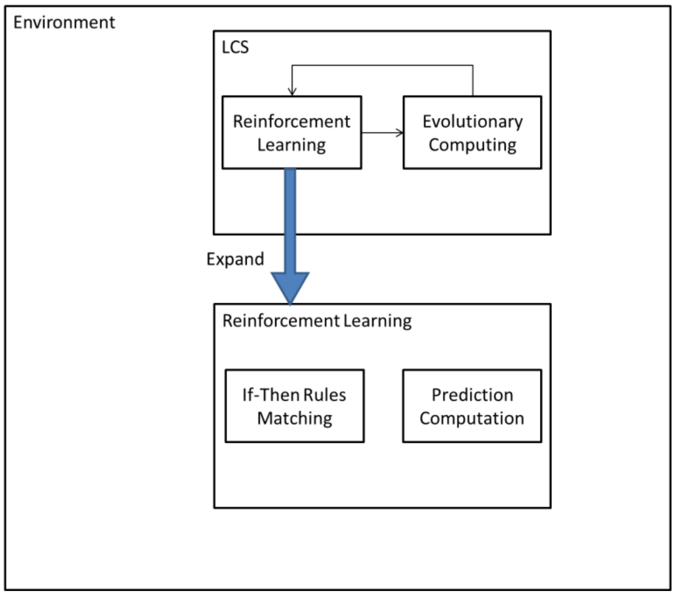


Figure 2-1: Overview of LCS

After understanding the basic data in this system, the whole process of it is clear now.

In overview, as can be seen from this diagram, the system mainly consists of two components: “Reinforcement” and “Evolution Computing” (also discussed in section 2.1 and 2.2). The system communicates with the environment and gets inputs form outside. The inserted inputs would be processed by “Reinforcement” first to predict which class this input belongs to, and get the reward from environment. After this, “Evolutionary Computing” would take the responsibility to update the rules stored in the system according to their fitness which is proportional to the rewards it get.

More precisely, the “Reinforcement” part has two important subparts: “If-Then Rules Matching” and “Prediction Computation”. The following two parts would be about them.

2.4.1 If-Then Rules Matching

This part is the most important part in this system, it responds for doing matching between inputs and rules (in fact only condition part of a rule is used) in the database. The naive matching algorithm of it is as follows:

```
For each input
{
  For each rule
  {
     For each char (bit) in a input and rule database
     {
          Do Compare
     }
  }
}
```

Algorithm 2.2-Naive Matching

The algorithm is quite simple as we can see. For each input and rule, we compare whether every bit in them are the same, if so, they are match, vice versa.

An example of “Matching” process could be shown like this:

Input1	1	0	1	0	1	1	0	1
Rule(Condition)	1	#	1	0	#	#	0	1
Input2	1	1	1	0	0	0	0	1

Table 2.4-Matching Example

Suppose there are two inputs inside the system and they would be matched against the only rule in the system. In this example, both Input1 and Input2 could match this Rule.

2.4.2 Action Selection and Evaluation of Rules (Classifiers)

The set of all classifiers matched in the system at current situation is called the match set [M]. A match set is the source of their corresponding action set [A] ("Then" parts of the chosen classifiers). But for most of time, actions in [M] would not be all the same, which means LCS has to offer a mechanism to choose which action it would finally use (Sigaud & Wilson, 2007). In fact the mechanisms are different when different types of LCS are used. In this thesis, since XCS would be used as the target system, its mechanism would be introduced in its section 2.5.

"Evaluation of Rules" is important for finding a best solution from the LCS. The same with "Action Selection", this part will also depend on the type of LCS used. So an XCS version of this part would be described in detail in its background part as well.

2.5 XCS

There are many different types of LCS as discussed above. Among them, the XCS has been chosen to be optimised in this thesis.

2.5.1 Basic of XCS

As we can see from table 2.1, XCS is an accuracy-based LCS. It is invented by Wilson, Stewart W. in 1995 (Wilson, 1995). It is modified from the traditional LCS. And now, XCS has become the most studied LCS which could reflect its success (Kovacs, 2002-06-04).

In XCS, the most important difference between it and the traditional LCS is about the fitness function in its classifiers. As introduced in section 2.4, the traditional LCS uses a fitness function which is proportional to the rewards it gets. But in XCS, it uses a new fitness function which is proportional to the accuracy of predictions of the expected rewards (Sigaud & Wilson, 2007). In this case, those classifiers which expect big rewards but actually get small rewards would have fairly low fitness. As a result, XCS learns a complete Q-function, whereas a strength based system only learns part of the Q-function, and that is less suitable for backing up Q-values.

2.5.2 Action selection in XCS

In XCS, the action selection could depend on any of the approaches in RL. In Wilson's paper, it uses "Exploration and Exploitation" to select action from action set [A]. In this case, the system has two modes. While in the exploration mode, XCS just choose a random action in the [A]. And in the exploitation mode, XCS uses some strategies like Greedy to choose which action could get the best reward.

2.5.3 Evaluation of Rules in XCS

In XCS, the approach for evaluation of rules is similar to Q-learning in RL which follows the formula (Sigaud & Wilson, 2007):

K stands for the generation, E represents the Expected reward, α is the learning rate and r is the reward.

$$E_{k+1}(s) = E_k(s) + \alpha[r_{k+1} - E_k(s)]$$

And after each time an action was finally chosen, the classifiers in [M] would be modified as follows:

1) The expected reward (p) would be updated according to the real reward (r) and the learning rate β:
$$p = p + \beta(r - p)$$

2) The prediction error (e) would be updated like this:
$$e = e + \beta(|r - p| - e)$$

3) The raw prediction accuracy (k) for each classifier would be like this (e_0 is a threshold):
$$\text{if } (e<e_0),\ k = 1;$$
$$\text{else } k = \alpha(\frac{e}{e_0})^{-v}$$

4) Do normalization to the prediction accuracy:
$$k' = \frac{k}{\sum_{x\in A} k_x}$$

5) Update the Fitness (f) of classifier finally:
$$f = f + \beta(k' - f)$$

2.5.4 The Rules (Classifiers) in XCS

The classifiers have been extended with more fields in XCS with new parameter which would take the respond for evaluation of rules. Here is the new table for rules in XCS:

Attributes	Explanation
Condition (If part)	Condition is the part which is used to do match against inputs. It is usually represented by ternary alphabet {0, 1, #}. "#" here stands for "don't care", if a bit in the condition is "#", which means this bit would match no matter "1" and "0" at the same bit of input.
Action (Then part)	Action stands for the action (e.g. a classification) which would be returned once condition part matches
Prediction	Prediction is an average number which records the payoff estimated if this rule is matched and the action is finally taken by the whole system
Prediction Error	This is the error from prediction
Fitness	The fitness of this classifier, fitness is a concept of GA which would be discussed later, it is used to select good rules.

Table 2.5-XCS Attributes

3. Methods

As discussed in section 2.4.1, the algorithm of naive matching in LCS is quite simple. But one problem it has to face is the efficiency.

In fact, matching has been proved to be the most time consuming part in LCS, about 65%-85% of the overall time would be spent on this (Llor `a & Sastry, 2006). So to speedup this part of system is an important issue in the research of optimising LCS.

This section will describe all the methods I tried to optimise the matching process in LCS (XCS). There are totally five approaches here and they are in different directions. For each of them, the background of them would be explained firstly if needed. And then, the idea and algorithm of each one would be shown in details.

3.1 Common Background (Ternary Tree)

First of all, since most of my approaches are based on a previous method which is called Ternary Tree, this method would be discussed first as a common background.

This is an important optimisation approach because it uses a totally different point of view to optimise the matching process and gets a good enough efficiency finally. This new method was invented by Tim Kovacs and implemented by Tsentas in 2010 (Tsentas, 2010).

Almost all the optimisations towards matching are focusing on the process of matching. It is reasonable to do so because finding a better algorithm for matching would clearly be a direct solution to solve this situation. However, it is not easy to find other suitable algorithms to do match. Most of the new algorithms are actually based on the naive matching algorithm and optimise it with the architecture of hardware such as CUDA parallel match (Lanzi & Loiacono, 2010) and CPU cache efficient match (Chitty, 2012).

Ternary Tree is a new optimisation approach which avoids using the traditional way to optimise matching. Instead, it tries to use a different data structure to store the conditions of rules (other parts of rules are not necessary for matching, so mostly we only store conditions of them in this process). In this case, while doing the match, the traversal of rules in database would be much faster than before.

Ternary Tree is not only the name of this method but also the data structure used in this method. A Ternary Tree has a similar structure to Binary Tree but own three children belong to each node in it. The following figure is a full Ternary Tree.

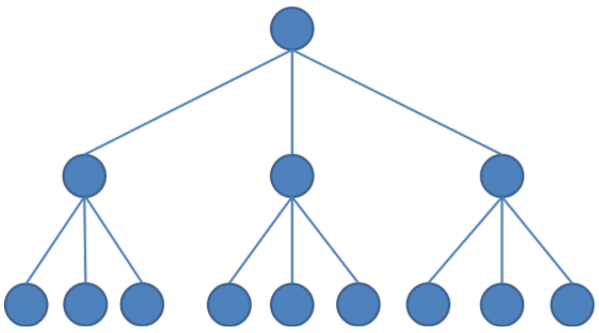


Figure 3-1: Full Ternary Tree

In fact, the first appearance of the concept of Ternary Tree is in 1963. It is used by F.J.M. Barning to represent the set of Primitive Pythagorean triples (Barning, 1963).

There are two types of Ternary Tree introduced by Yiannis Tsentas (Tsentas, 2010). They are single depth Ternary Tree and Multi-level Ternary Tree.

3.1.1 Single depth Ternary Tree

In this idea, only one depth of Ternary Tree would be used for storing the rules. The following example could describe the whole idea clearly.

> ***There are four rules stored in this system, they(condition part) are "#1" , "11" , "01" , "##". In the normal system, they would be usually stored in an ArrayList like:***
>
> ***#1 ->11->01->##***
>
> ***But in a single depth Ternary Tree, they would be stored like this:***

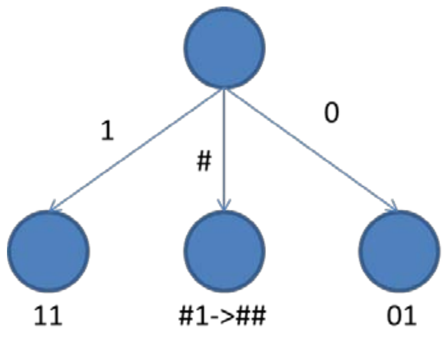


Figure 3-2: Single Depth Ternary Tree

As can be seen from this figure, there are three paths from a father node to its children. According to the construction law of the single depth Ternary Tree, the paths have different meanings:

1. Left path means rules in this path begin with '1'.

2. Middle path means rules in this path begin with '#'.

3. Right path means rules in this path begin with ‘0’.

Benefit from this mechanism, while doing the match, we look at the first bit of each input. If it is ‘1’, we just do match in the left node and the middle node (‘#’), and if it is ‘0’, we only do match in the right node and the middle node (‘#’).

> ***For example, suppose an input is “10”, in this system above, we would choose to use left node and middle node to process it. The left node would fail to find anything match whereas the middle node would return “##” as an answer.***

In this case, the database which the inputs need to faces would be smaller than the original database. This structure in theory could save about 1/3 of the total time.

3.1.2 Multi-level Ternary Tree

In fact, the idea in the single depth Ternary does not necessarily need to use a ternary tree. Instead, we can just use three arrays to store rules separately by their first bit. And then we can just return the arrays as the current database corresponding to current input. However, the idea of Ternary Tree is quite important, because it is the basic theory for this multi-level Ternary Tree.

In a multi-level Ternary Tree, the construction law is similar to a single depth Ternary Tree except that in this tree, instead of relying on the first bit, we build the tree by every bit in the rules. An example would describe the situation:

> ***There are four rules (condition part) in the database which are “#1” , “11” , “01” , “##”. The structure of the multi-level Ternary Tree built on this is like:***

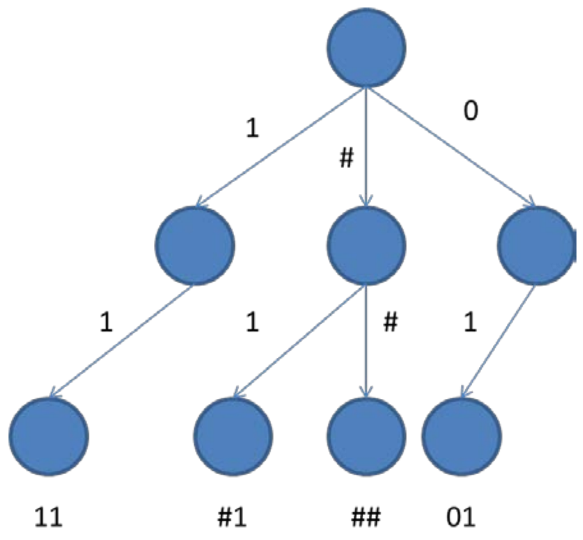


Figure 3-3: Multi-Level Ternary Tree

This example is the same one in the single-depth, so the difference between the two trees could be seen clearly from their figures. For every time it reaches a new node, the next bit of the rule would be considered to choose the next direction (‘1’ means left, ‘0’ means right, ‘#’, means middle). Finally when it reaches the node of last bit in the rule, the value of the whole rule would be stored in this node. At the end, all rules (conditions) would be in the leaves of the tree.

Because of this new data structure, the algorithm for matching would be changed as well. Instead of using naive method, we do match in Ternary Tree like walking in existed paths. This approach has been proved to be faster than the naive one for most of time.

Base on the multi-level Ternary Tree above, the matching algorithm could be explained by the figure below:

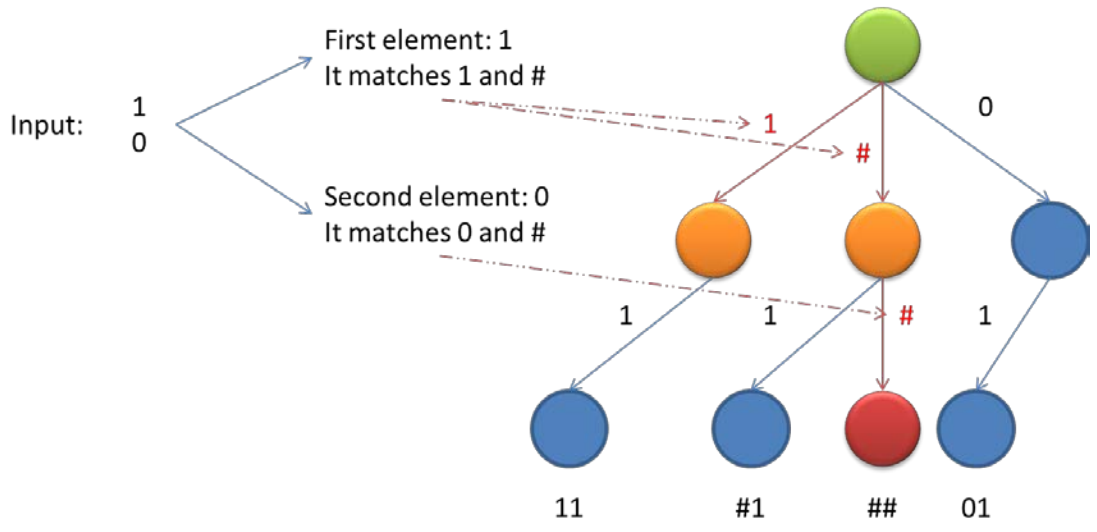


Figure 3-4: Matching Algorithm in Multi-Level Ternary Tree

In this figure, there are totally three steps cost to get the answer. Firstly, the algorithm starts from the root. After that, because the first element of the input is ‘1’, this matches ‘1’ and ‘#’, so the algorithm choose both ‘1’path and ‘#’ path. And then, the second element ‘0’ decides the next path should be ‘0’ or ‘#’, in this example, only one path could be found. So finally we get the value stored in the leaf which is “##”.

3.1.3 Modifications on Ternary Tree

In the XCS system, the GAs control the updating of the rules in this system, so it is normal to delete rules during the process.

In Ternary Tree implemented in Tsentas' thesis, there is a method which is used for deleting rules inside it. However, according to the investigation of this thesis, there is a potential problem with the delete method in this system. In the original method, the system is just deleting rules by simply removing the responding leaves. This seems to be correct, but in fact this process is producing useless fragments all over the trees. The following figure may be easier to understand.

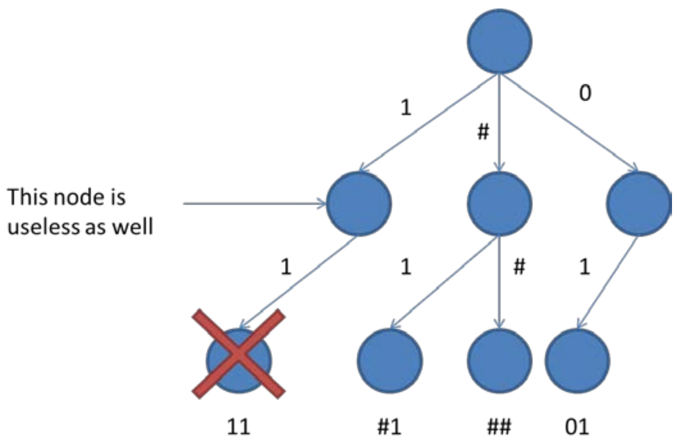


Figure 3-5: Only Removing Leaves Would Cause Problems

In this figure, if the leaf “11” has been removed, then its father would be useless as well because it doesn’t have any other children. If we don't delete the father node, next when the input begins with ‘1’, we would search the root’s left child as usual. But in fact at that time there is nothing valid in root’s left child any more. At the very beginning, this problem would not be big. But after the system has deleted rules for several times, there would be more and more useless nodes in the system which would affect the efficiency of matching obviously.

So, the algorithm of correct deleting method should be like this (In order to make this work, a father pointer has been added to the attribute of node):

```
Delete leaf;
Current_node = leaf -> father;
while(Current_node doesn’t have other children)
{
Current _node = Current_node -> father;
Delete where the Current_node comes from;
}
Return;
```

Algorithm 3.1-Updated Deleting Algorithm in Ternary Tree

However, whether this algorithm would really improve the efficiency of the system would rely on more investigations in the future. This project implemented this new deleting method but would not offer an experiment on it because this is not the main purpose of this thesis.

3.2 Data Pre-processing (Hash Matching)

In this section, the first method “Hash Matching” would be introduced. It is the new invention of this thesis.

3.2.1 Basic Idea

This method is focusing on pre-processing rule database. In an XCS system, there are big amounts of rules stored. If we pre-process the rules and build some links to potential inputs before the matching process, once an input comes, the matched rules would be found directly.

3.2.2 Implementation

To implement this, two basic components should be described.

(1) Possible Input Table

The first one is a table of possible inputs. This structure is implemented using a hash table and this is the reason why this method is called “HashMatch”. As discussed in the basic idea, the role of this table is to store all the possible inputs which would appear in the future.

The process for finding all possible inputs according to rules inside the system could be described as the following table:

Rules in the system	Possible Inputs they respond to
#10	010,110
##1	111,011,101,001
110	110

Table 3.1-Finding Possible Inputs

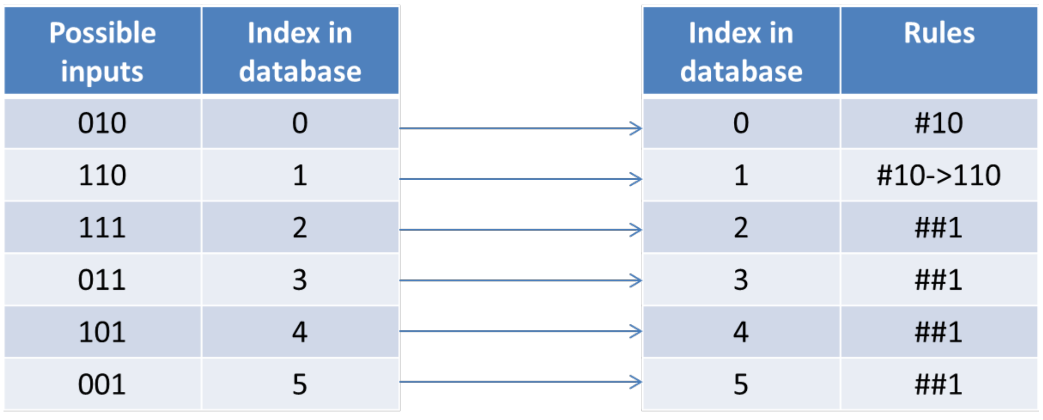
So in the hashtable, we would build six places totally (101,110,111,011,101,001).

(2) Linked Rule Data Base

After finding the possible inputs for each rule we would link the possible input with the rule. In this case, once an input comes, we can find its corresponding rules directly. The structure of Linked rule database is a 2D-array, and the indexes of them could be retrieved easily from the “possible input table”.

Overview

An overview of this system would be like the situation in the following figure:



Possible inputs	Index in database
010	0
110	1
111	2
011	3
101	4
001	5

Index in database	Rules
0	#10
1	#10->110
2	##1
3	##1
4	##1
5	##1

Possible Inputs Table

Linked Rule Data Base

Figure 3-6: Overview of HashMatch

This situation of the system is based on the Table 3.1.

(3) Description of Algorithm

The system is quite clear from the figure above. A description of its algorithm based on the figure could be made as follows:

```
Current Input CI;
Possible Inputs Table PIT;
Linked Rule Data Base LRDB

If(cannot find CI in PIT)
 then output null;
else
{
  Get index i from the table;
  Find the corresponding rule list in the LRDB following the I;
  Output the rule list found; }
```

Algorithm 3.2-HashMatch Algorithm

(4) Experiments

The experiment of this method would be shown in section 4.1.

3.3 Recursion Replacement in Ternary Tree

3.3.1 Basic Idea

In the algorithm of Ternary Tree, there is a function called “getmatchset()” which is responds for finding the match sets for a given input. This function is using a recursive algorithm for finding match sets.

```
getmatchset(current_node, input, matchset )
{
  If(current_node is a leaf node)
  {
    add the value in this leaf into the matchset;
    return null;
  }
  current bit = first bit in the input;
  If(current bit == ‘1’)
  {
    //recursive call the function to search for the rest part of input
    getmatchset(current_node.leftchild, input(1:n-1), matchset );// ‘1’
    getmatchset(current_node.middlechild, input(1:n-1), matchset );// ‘#’
  }
  If(current bit == ‘0’)
  {
    //recursive call the function to search for the rest part of input
    getmatchset(current_node.rightchild, input(1:n-1), matchset );// ‘0’
    getmatchset(current_node.middlechild, input(1:n-1), matchset );// ‘#’
  }
  return matchset;
}
```

Algorithm 3.3-GetMatchSet Algorithm in Ternary Tree

For simple problems, the performance of a recursion function is similar to an iterative algorithm. But if the data processed is big, in theory, an iterative algorithm would bring better performance than a recursive one.

In theory, every recursive function can be transformed into an iterative function. The trick is to use data structures like stack, queue instead of the normal one.

In this project, both of the data structures have been used to optimise this recursive function.

3.3.2 Implementation

(1) Implementation of Stack and Queue

The data structures “stack” and “queue” are already exist in JAVA, but they are for general use which means they would waste some time on several functions which are useless for the project. Especially for stack in JAVA, it is just an extension of the data structure “Vector” which is not so efficient.

So in this thesis, a specified stack and queue would be implemented.

It is easy to implement those. There would be three components for each of them: element, push (offer in queue) function and pop (poll in queue) function.

(2) Implementation of “getmatchset” (stack version and queue version)

It is a general method for transferring from tree traversal function into iterative one. Those two figures could show the method clearly.

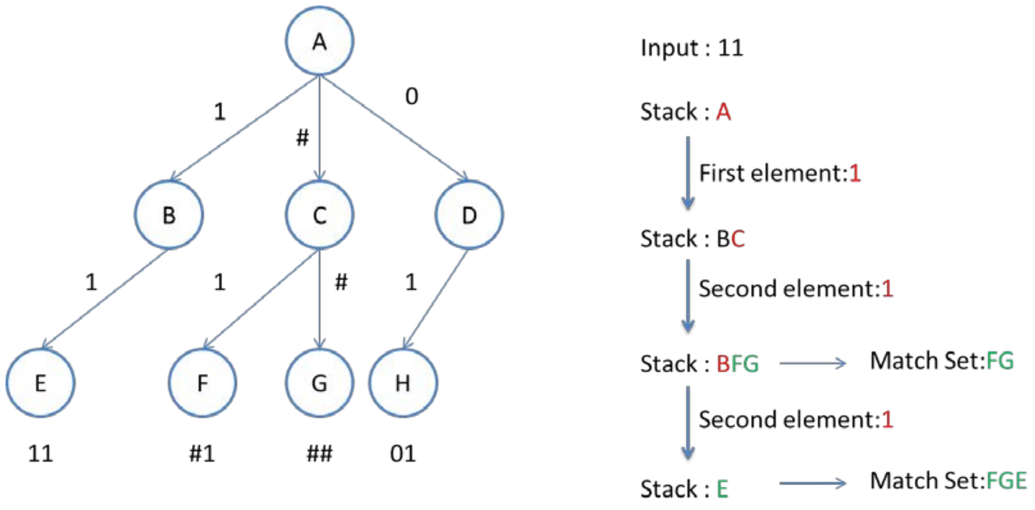


Figure 3-7: Transfer Recursion to Stack

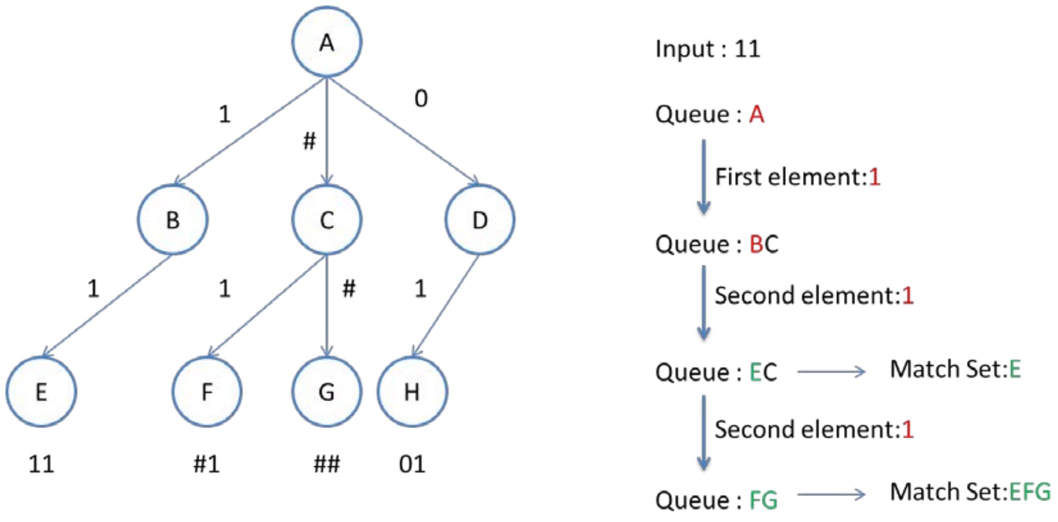


Figure 3-8: Transfer Recursion to Queue

(3) Experiments

The experiment of this method would be shown in section 4.2.

3.4 Object Pool

3.4.1 Background

Design Patterns and Creational Design Patterns

There are many design patterns in object-oriented programming. They are reusable common solutions for solving programming problems and optimising the efficiency of programs (PREE, 1995). Among the patterns, there is a class which deals with the object creating and deleting. This class is called creational design patterns. Object pool is one useful pattern in this class. As a creational design pattern, object pool concerns on managing the objects used in the system.

(1) Object Pool

In JAVA, there exist methods for creating and deleting. But the process of creating an instance of a class equals to calling the constructor of this class. Once the constructor of a class called, all of the attributes in it would be initialized, this would cost much time in runtime. The deleting process in the JAVA is easy. After removing an instance from the system, the garbage collection would clear the removed instance automatically (in JAVA).

While using object pool, before using, we create some objects in the object pool. This is the only place we would call the constructor of the class. Then, when a new instance is needed in the runtime, we borrow an object from the object pool. This process is just like using a pointer to point at this borrowed object which is efficient. And the big difference in the idea of object pool is the deleting operation. While deleting objects using object pool, we do not really remove them, instead, the objects are recycled back to the object pool. In this case, the objects in the object pool could be reused many times during the runtime without calling constructors any more.

(2) Implementation of General Purpose Object Pool

The simplified flow diagram of general purpose object pool is like this:

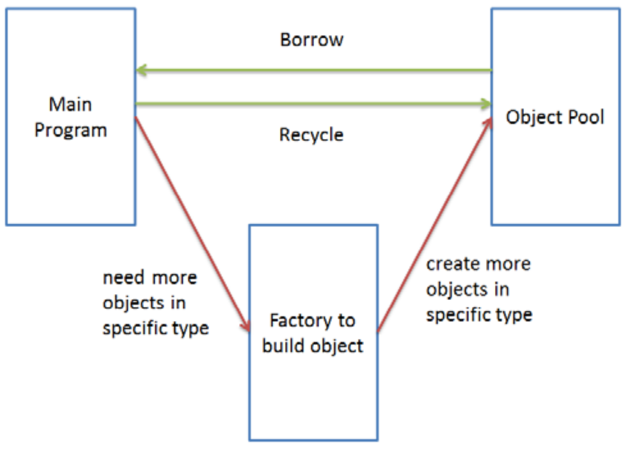

Borrow
Main Program
Recycle
Object Pool
need more objects in specific type
Factory to build object
create more objects in specific type

Figure 3-9: General Purpose Object Pool

As can be seen from the figure, there are two important parts in a general purpose object pool system: Object Pool and its Factory.

The object pool is the place for storing different types of objects. It controls the process of borrowing and recycling.

The factory here is used for creating different types of new objects according to different needs from main program. The idea of factory comes from the Factory Pattern which is another important design patter in object-oriented programming. Factory Pattern is usually used to control the implementations of instances during the runtime.

(3) Implementation of Specified Object Pool

Sometimes, there is just one type which is used a lot in a system. In this case, a general purpose object pool is not really needed for processing. Instead, a specified object pool could be more flexible and offer better performance.

The flow diagram of a specified object pool is like this:

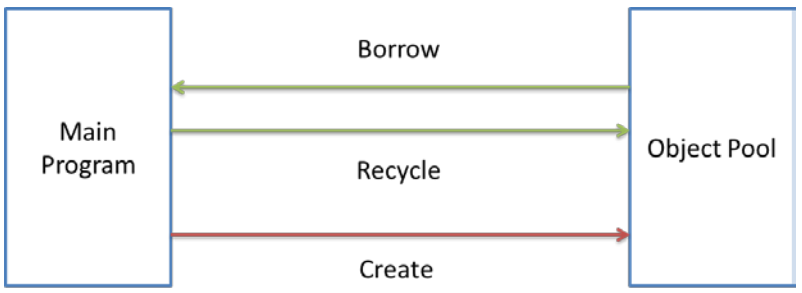


Figure 3-10: Specified Object Pool

In fact, the theory of specified object pool is the same with general purpose object pool. The only difference is that a specified object pool does not need a factory to create objects anymore because it is focused on a specified type.

3.4.2 Basic Idea

This approach is based on the multi-level Ternary Tree. In a multi-level Ternary Tree, the basic elements are nodes in it. And in XCS system, while the rules (nodes in Ternary Tree) are updated by GA, many of them would be added or deleted from the Tree. Due to this, this approach intends to manage the nodes in Ternary Tree by the specified object pool.

3.4.3 Implementation

(1) Data Structure

In order to implement a specified object pool, firstly the mechanism of borrowing and recycling should be decided. In this thesis, LIFO is used for managing the objects (nodes in this case). LIFO stands for “Last In First Out” which means the latest object added into the database would be the first one to be processed later. A classical basic data-structure using LIFO is the stack. In stack, adding element is called "push" and deleting element is called "pop". While pushing, the element would be pushed to the top of the stack. And while popping, the top element would be popped out. This theory is suitable for managing an object pool. The imagination diagram would be like this:

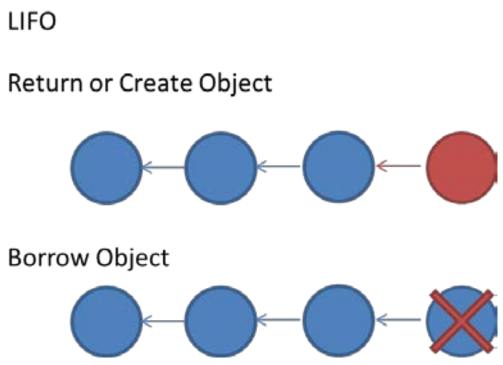


Figure 3-11: Object Pool with LIFO

In this figure, the red node is the one returned to the object pool whereas the node with a red crossing on it is the one borrowed from the object pool.

In this thesis, there are totally four data structures (all of them are in JAVA because the XCS system is in JAVA) tried to implement the LIFO Object Pool, they are: ArrayList, LinkedList, Vector and Stack.

Data Structure	Description	Suitable for LIFO
ArrayList	ArrayList is similar to array which uses continuous spaces in memory. It is a dynamic array, so it is possible to change its length.	Does not have a dedicated operation for removing last element in it.
Vector	This data structure is a thread safe version of ArrayList.	Same with ArrayList
LinkedList	A LinkedList is a list which connects all its elements by pointers. It does not use continuous space in the memory	Has an original method for "push" and "pop" in its list because of its structure.
Stack	The stack in Java is actually an extension of Vector which is not so efficient for LIFO	Has an original method for "push" and "pop" in its list because of its structure. But it is not efficient for the structure of Vector

Table 3.2- Descriptions of Four Data Structures

(2) Object Pool Ternary Tree

After choosing the data structure for Object Pool, it would be easy to combine it with Ternary Tree. The Implementation diagram would be like this:

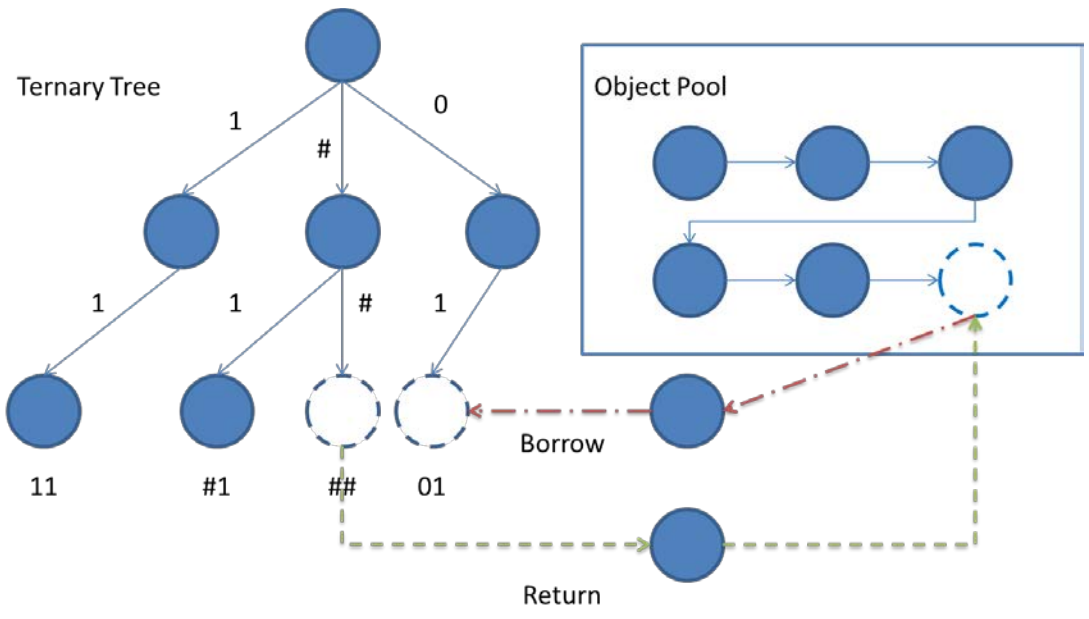


Figure 3-12: Add new leaf "01" and remove leaf "##" in a Ternary Tree with Object Pool

As can be seen from the figure, when a new object is needed in the tree (adding a new rule), we borrow one from the object pool. And when an object is removed from the tree (deleting a rule), we return it back to the object pool for reuse.

(3) Experiments

The experiment of this method would be shown in section 4.3.

3.5 Parallel Computing on XCS

In this section, the classical optimising method "Parallel Computing" would be introduced.

Firstly, some backgrounds of "Parallel Computing" would be introduced.

Secondly, this thesis would come up with two approaches based on GPU and CPU. GPU algorithm is based on the paper by Lanzi & Loiacono (Lanzi & Loiacono, 2010) whereas the CPU algorithm is an invention of this thesis.

3.5.1 Background

(1) Parallel Computing Models

> ***"Parallel processing involves taking a large task, dividing it into several smaller tasks, and then working on each of those smaller tasks simultaneously" (Tushar Mahapatra & Sanjay Mishra, August 2000).***

During this progress, all processors which took part in computing will interact with each other to finish the whole task (Grama, et al., January 16, 2003). The main aim of parallel computing is to increase the execution speed of applications and enhance the ability to process some special tasks which have a great deal of data.

However, not every problem is suitable for being processed by parallel computing. Only those tasks which could be divided into parallel sub-tasks can be successfully refactored by parallel computing.

In fact, the idea of using parallel computing with multiple processors dates back to the earliest electronic computers (John L. Hennessy & David A. Patterson;, 2006). In 1996, Flynn introduced a simple model which could be used to classify computers. This model has been updated into four parts today which are:

1) SISD (Single instruction stream, single data stream)

This one is the uniprocessor. The only processor gets the whole data and processes it by its own sequential instruction.

2) SIMD (Single instruction stream, multiple data streams)

This model aims at executing same instructions on different datasets at the same time. It involves splitting database into several datasets and assigns them into different processors/cores. Several decades before, this model could only be used by devices with multiple processors. But now, since the architectures of chips have changed a lot, this model becomes the most popular one to be used. This thesis would be based on this model as well.

3) MISD (Multiple instruction streams, single data stream)

No commercial of this type of model till now.

4) MIMD (Multiple instruction stream, multiple data streams)

Each processor runs its own instructions using its own data. Distributed computing could be seen as an example of this (it could be SIMD as well). In general, this progress is more flexible than data-level parallelism because each processor is executing its own instruction.

(2) Shared Memory Model

Except for processors, parallel computing also relies on memories. There are mainly two kinds of memory used in parallel computing: distributed-memory and shared -memory. This project is going to work with shared-memory structure.

The following figure is the basic structure of centralized shared-memory multiprocessor (multicores):

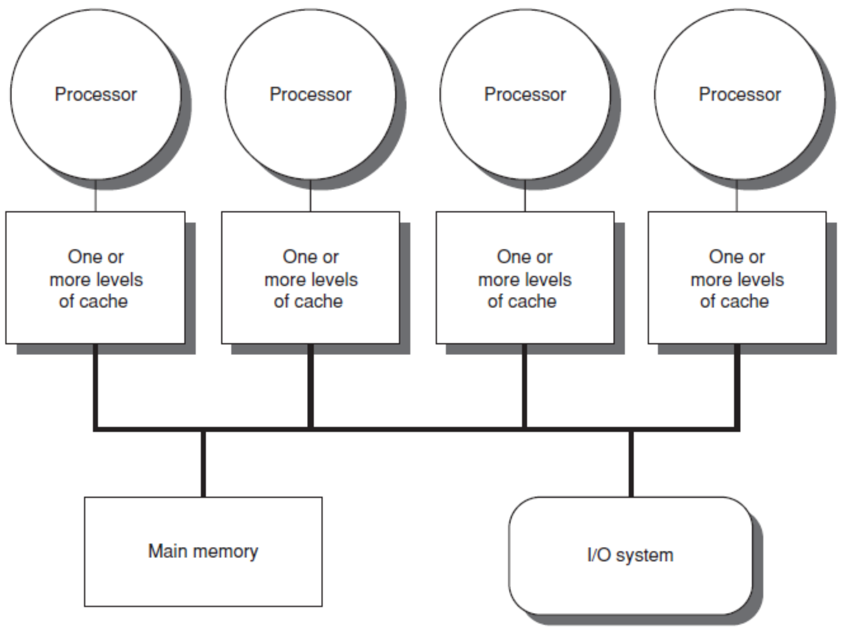

Processor
Processor
Processor
Processor
One or more levels of cache
One or more levels of cache
One or more levels of cache
One or more levels of cache
Main memory
I/O system

Figure 3-13: Basic Structure of Centralized Shared-Memory Multiprocessor (John L. Hennessy & David A. Patterson;, 2006)

(3) Approaches for Parallel Computing

There is a great deal of approaches for taking advantages of parallel computing, and they could be classified by the chip they are based on. The most used chips are CPU and GPU.

The second one is the application area of GPU. GPU was seen as an expert who could only help to process images. But now, since more instruction sets have been added to it, GPU has become a totally different role which could be used for general computing just like CPU.

a) CPU based approaches

Several decades ago, the architecture of CPU just own single core (e.g. Pentium series in Intel) which could only be used for sequential computing. Parallel computing at that time required several CPUs which is quite expensive.

But nowadays, the CPUs are quite different from decades ago. The latest generation of CPUS such as “Intel’s i7 series” and “AMD’s Phenom™ II Quad-Core Processors” have four cores for computing. Even some products which are cheaper like “Intel i5” or “AMD’s Phenom™ II triple” have two or three cores which could be used for parallel programming.

In a multicore CPU, each core could work separately as a processor. In this case, an i7 CPU (four cores) would offer four processors for parallel computing. This means only one CPU would be needed for parallel computing to enhance the performance of programs.

However, multicore technology does not equal to multiprocessors because multicore technology could only use SIMD model whereas a multiprocessors could use both SIMD model and MIMD model. Anyway, due to the reason that multicore technology is much easier and cheaper for general users, most user-oriented programs are now based on it.

To take advantage of parallel computing based on CPU, some APIs (application programming interface) and libraries would be used to control the cores/processors.

i. **MPI**

MPI stands for Message Passing Interface. It is a library specification for message passing among parallel computers and standard defined by committee of vendors, programmers, and users (Snir, et al., 1995). The original purpose of MPI is aimed at severing several multiprocessors (multiple computers). But now it could also be used for the multicore computing as well. The languages supported by it: FORTRAN, C/C++.

ii. **OpenMP**

The OpenMP is an API which focuses on parallel computing on shared-memory platforms (e.g. single processor multiple cores) (Quinn, 2004). Instead of controlling communications among processors, it takes the advantage of multicores by using multithreads. The languages supported by it: FORTRAN, C/C++.

iii. **TBB**

The TBB stands for Intel® Threading Building Blocks. Similar to OpenMP, it is a library which includes many functions for taking advantages for multicore CPU. TBB is based on multithreads as well. The language supported by it: C++.

iv. JAVA Fork/Join

All of the approaches introduced above are based on C/C++ and FORTRAN, they don't support JAVA. In fact there are few approaches about parallel computing which are supporting JAVA. There are some existed design patterns which are for concurrent programming, but most of them have some drawbacks on efficiency while doing synchronization. Programmers of JAVA have kept taking the advantage of multicores by manually writing multithreads codes for a long time.

However, recently, in JAVA SE7, a new design pattern called "Fork/Join" has been introduced for solving this problem. "Fork/Join" is a new framework invented by Doug Lea which concerns on using parallel computing in a convenient and efficient way (Lea, 2000). Since the most prominent XCS is based on JAVA, this is an important approach for optimising it.

Actually, the idea of Fork/Join is easy and here is a simplified algorithm of it.

```
Compute(task)
{
    If (size of task  <= threshold)
   {
      Solve the task in normal way;
   }
   Else
   {
      Split task into small sub-tasks;
      Fork every sub-task, run compute(sub-task);
      Join every sub-task;
      Get result from each sub-task;
   }
}
```

Algorithm 3.4- Simple Description of the Algorithm of Fork/Join

In Fork/Join, the first thing is to split the current task into sub-tasks. And when the size of a sub-task is small enough, it will be sent to be solved by a general algorithm. The Fork part in the algorithm is for giving threads for all the current sub-tasks and parallel call the compute function. The Join part in the algorithm is used for waiting all the sub-tasks to finish their works. And at the end, all the results from sub-tasks would be collected.

b) GPU based approaches

GPU stands for graphic processing unit. It is a dedicated parallel processor which optimised for processing graphical computations. However, the definition is not enough for explaining the modern GPU. A modern GPU is massively parallel and fully programmable. Also, the parallel floating point computing power of it is orders of magnitude higher than a CPU (McClanahan, 2010).

For graphical computing, GPU receives graphic data from CPU software (OpenGL, DirectX) and process them. After that, the processed data would be used for displaying.

In 2006, with the release of NVIDIA’s GeForce 8 series, the GPUs have made a big step to expose its ability of being massively parallel processors. At that time, an important language CUDA was introduced by NVIDIA.

In 2010, the Fermi GPU was introduced and it is the first GPU which designed originally for GPGPU (general purpose-GPU).

From the history introduced above, it is clear that the evolution of GPU’s architecture has transformed from a specific purpose based processing unit to a set of highly parallel and programmable cores for general purpose computation.

There are mainly two famous APIs/Libs which aim at supporting parallel programming with GPU: CUDA C and OpenCL.

i. CUDA C

CUDA is a general purpose parallel computing architecture which has been introduced by NVIDIA in November 2006. The main purpose of this architecture is to solve complex computational problems in a more efficient way than on a CPU (NVIDIA, 4/16/2012). The modern structure of GPU is quite different from CPU and here is a figure which compared them with each other:

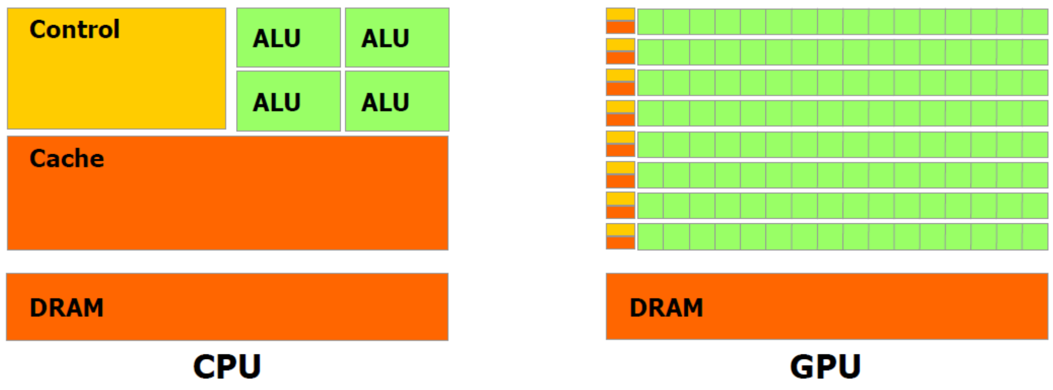


Figure 3-14: Structure of CPU and GPU (NVIDIA, 4/16/2012)

CUDA C is ANSI C extended by several specific keywords and constructs (Shane Ryoo, et al., 2008). CUDA C enables programmers to create a new type of C functions, called kernels. When the kernels functions are called, it will run N times in parallel by N different CUDA threads.

ii. OpenCL

OpenCL is an API for GPU programming. The main feature of it is that it could allow programmer to compile at run-time. This feature means programmer can get a better control of target GPU. While using OpenCL, unlike CUDA C, the programmers need to control the memory manually including avoiding dead locking and so on (Kamran Karimi, et al., 2010).

3.5.2 Implementation

(1) CUDA Parallel Match

This implementation is based on the algorithm referred to by Lanzi and Loiacono in 2010.(Lanzi & Loiacono, 2010). In this algorithm, there are two big improvements over the naive matching algorithm: Bitwise Operation Rules and CUDA Kernel Function. Bitwise Operation Rules is a classical approach for optimising the representation of rules in the database whereas CUDA Kernel Function is an invention in Lanzi and Loiacono's paper.

a) Bitwise Operation Rules

The first one is the different representation method of rules (condition part) in the system.

In original system rules are stores like "1010##" and the judgement sentence in the naive matching algorithm is:

```
if(rules[i].charAt(j)!=input.charAt(j))&&(rules[i].charAt(j)!='#')
```

As can been from above, this sentence is doing two "not equal" judgements.

In Lanzi and Loiacono's idea, if bitwise operation could be used for the matching, a better performance could be brought. As we already know, the inputs of the system are in binary code whereas the rules in the system are ternary codes. If it is possible to transfer the rules in to binary codes as well, the bitwise could be used.

Based on this point of view, Lanzi and Loiacono came up with a new coding method for rules. It uses two binary codes to stand for a single original ternary code. The two binary codes are called FP (first parameter), SP (second parameter). The following figure is an example for this:

Input	110001	101000
Normal Rule	1100##	101#1#
FP	110011	101111
SP	111100	111010

Table 3.3- New Way to Encode Rules

As can be seen from this table, FP is used to stand for the binary values at the positions like before (while facing the position which is #, whether 0 or 1 would be accepted). SP is used to stand for whether a position is a general bit (#), the value equals to 1 if the position is not a general bit and 0 if the position is a general bit. Based on this coding method, the match algorithm would be changed like this:

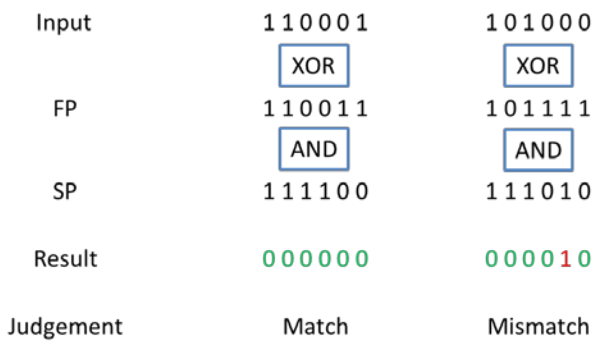


Figure 3-15: New Matching Algorithm for New Codes

The judgements in this algorithm are depending on the results. If results of all bits are “0”, the judgement should be “Match”, vice versa.

b) *CUDA Kernel Function*

The key modification in Lanzi & Loiacono's algorithm is to transfer the naive algorithm to a kernel function in CUDA. First of all, the essential characteristic of the matching process is found. As shown in the figure below, the process of whole matching is similar to matrix multiply.

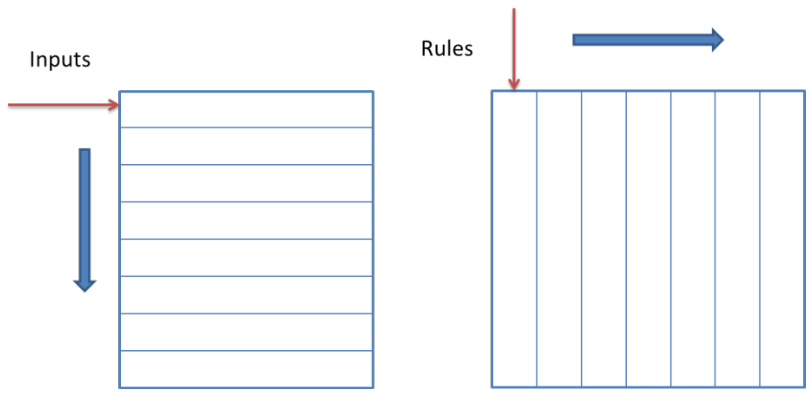


Figure 3-16: The Situation While Doing Matching

Since there are many cores and threads could be applied by a GPU, each sub-match could be assigned in a single core/thread. For example, the inputs in a system is a 2*2 matrix (2 inputs and length of each is 2), the rules in a system is a 3*2 matrix (3 rules and length of each is 2). In this Lanzi & Loiacono's algorithm, each core only need to process one match process (it is acceptable because there is huge number of cores in GPU compared with CPU). In this case, core1 will only be respond for doing match for input1 and rule1 while core2 will only be respond for input1 and rule2 and so on.

c) CUDA Kernel Function Conclusion

According to Lanzi & Loiacono's paper, the conclusion they get is for small problems, this method would be slower than naive matching due to the transfer overhead of GPUs. However, this approach is quite useful for processing big problems. Accordingly, this method could offer an acceleration of 3-12 for interval-based representations and 20-50 for ternary-based representations.

(2) JAVA Fork/Join Parallel based on Ternary Tree

In this section, an approach of parallel computing which is based on CPU would be explained. This method is an invention of this project which aims at optimising Ternary Tree using “Parallel Computing”.

Most of implementations which use parallel computing are based on C/C++. This is because there are few APIs which support JAVA for parallel computing. However, as described in the section 3.4.1, Fork/Join is a new design pattern in JAVA SE7 which aims at processing parallel computing problems in a convenient way.

Since Fork/Join still relies on multi-threads, it would only support using multi-cores CPU for parallel computing. The architecture of CPU is quite different from GPU’s. In a CPU, there are usually less than four cores. So if more than four threads are applied, some of them would not executed parallel. Instead, a core which is assigned by more than two threads would have to frequently switch between them which would also cost time. And it is also the reason why the algorithm introduced by Lanzi and Loiacono is not suitble for CPU parallel computing.

Because there are less nodes in a Ternary Tree than the number of rules in a naive system, a CPU based parallel algorithm would be more suitbale for optimising it.

The idea of parallel using Ternary Tree follows the traditional way of using Fork/Join algorithm (described in Algorithm 3.3). The following figure shows the whole system for parallel Ternary Tree.

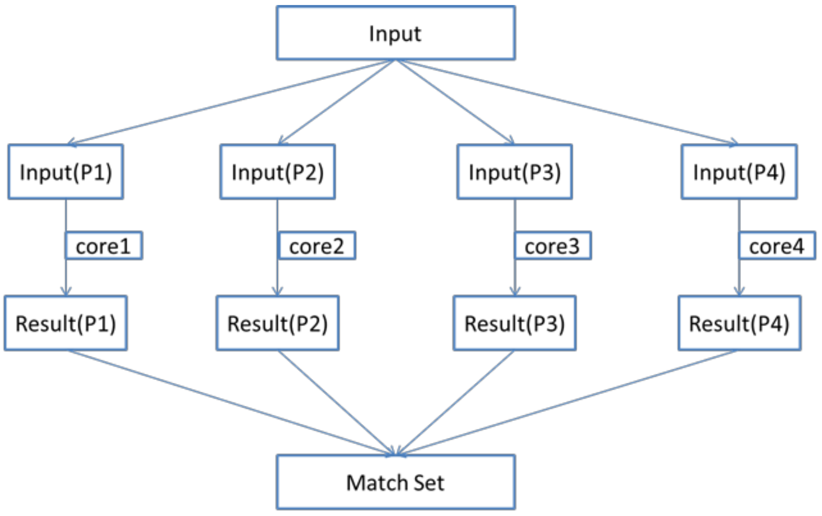


Figure 3-17: The Whole System of Parallel Ternary Tree

The purpose of this algorithm is clear. The whole input set is spilt into several parts for finding their local match set. And at the end of the system, all the results would be collected totally as match set. The simplified algorithm of it is as follows:

```
Compute(input set)
{
    // threshold could be decided by the number of cores
    If (number of inputs  <= threshold)
    {
       find local match set method in Ternary Tree using given subset;
    }
    Else
    {
       split list of inputs into subsets;
       Fork every subsets, run compute (on subset);
       Join every subsets;
       Get local match set from all subset;
    }
}
```

Algorithm 3.5- Algorithm of Parallel Ternary Tree Based on Fork/Join

(3) Experiments

The experiment of this method would be shown in section 4.4.

3.6 Input Tree

3.6.1 Basic Idea

As discussed in the background section, the basic and key problem with the naive matching is that it has to use for loops to do the traversal thorough all the inputs one by one. So many approaches for optimising this system would concern on how to decrease visiting time of inputs. Those methods which could do several matches in a single step are called "batch matching" algorithms. Tsentas also investigated some ideas of "batch matching" in his thesis (Tsentas, 2010).

The idea of input tree comes from a "batch matching" imagination. The original thought is to find the relationships among inputs and summarise the common features of them. Based on this, once an input has been tried, the similar ones would only needed to be partly matched later. For example, the situation is like the follow figure:

Rule1:	10011#	Rule2:	11011#
Input1:	100111	Input1:	100111
Input2:	100111	Input2:	100111
Input3:	100110	Input3:	100110
Input4:	100101	Input4:	100101
Input5:	100100	Input5:	100100
Input6:	100000	Input6:	100000
Input7:	101001	Input7:	101001
Input8:	110101	Input8:	110101

Figure 3-18: Situation of "batch matching" in Imagination

As can be seen from above, for Rule1, if Input1 matches it, we can just ignore the same parts in other inputs (e.g. Input2 would be totally skipped because it is the same with Input1. In Input3, the only bit needed to be match is the last bit). For Rule2, if Input1 does not match it, the first mismatch bit would be recorded and any other inputs with it would not match as well (e.g. Input2-7 would not match because the 2nd bits of them are the same and it is a mismatch).

However, to implement a program which satisfies this would require a data pre-processing before doing match. There are three big problems with this implementation:

(1) It is hard to decide which input should be the base input (Input 1 in the example).

(2) This process would cost much time for finding a correct order for all the inputs because for each input we would have to compare how similar it is to the base input. This means we have look at the whole input instead of part of it.

(3) There has to be a mechanism to record the similarities between every input and the base input. If a big table would be made for this, it would cost too many spaces.

Fortunately, there is a data structure which satisfies all of the features needed above and has a good efficiency for processing data. It is a Binary Tree.

A binary tree is a tree in which each node has two children. Similar to the Ternary Tree algorithm, this tree stores all the inputs in the leaves of the tree. The following figure is a comparison between naive storage method and Binary Tree:

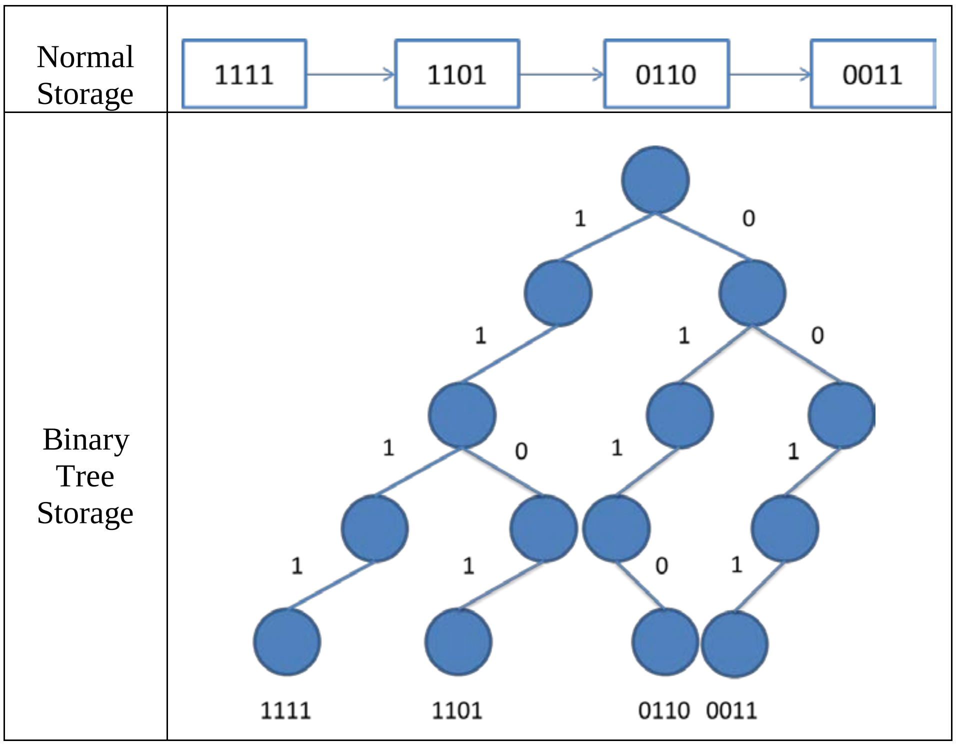


Figure 3-19: Comparison Between Naive Storage and Binary Tree

As shown in the figure, the input tree (Binary Tree) follows the similar rule of Ternary to store things. For processing each input, the algorithm translates each bit of input into path in the tree and follows this path to find which leaf the input should be stored in.

This data structure would solve the three problems discussed before.

(1) For the problem 1, there is no need to find a base input now in this data structure because in this tree all relationships among branches are clear. Any rules stored in a same branch would share a same part, for example. “1111” and “1101” are sharing the same path from depth0 to depth2.

(2) For the problem 2, it is easy to find all the similarities between any two inputs. This is because the similarity is just how far an input is from another in input tree. For example. “1111” and “1110” would be quite

similar to each other because the first three parts of path are the same. This is also the reason that they would be neighbours in input tree.

(3) There is no need to record the similarities among them because they are already shown by the structure of tree.

This method could be used with normal rule database as well as Ternary Tree. However, to make it work with Ternary Tree would be more valuable for the research. It is because Ternary Tree has been proved to have a better performance than the normal rule database, if combing Input tree and Ternary Tree would get even better speed up, this approach could be useful in this area. Also, the structures of Input Tree and Ternary Tree are similar to each other, so it would be interesting and challenging to find a way to combine them together.

3.6.2 Implementation

(1) Input Tree

The first and basic implementation would be the input tree itself. Similar to the Ternary Tree, the building process of Input Tree also involves following the path of every bit in inputs. Here is an example of building an input tree:

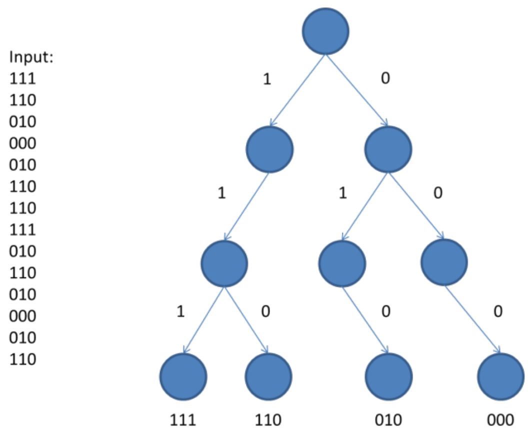


Figure 3-20: Example of Building Input Tree

As shown in this figure, this input tree is a product of the all the inputs to the right of it.

There is a phenomenon here about the inputs. There seems to be many inputs, but in fact, there are just four distinguish ones. This is normal in XCS because the input file would be used several times for training. As we can see, the input tree has taken advantage of this phenomenon and builds the tree with only four leaves. During the building process of the Input Tree, the repeating ones are ignored automatically by its data structure.

There are more good features this Input Tree could bring to us which would be discussed in the next section.

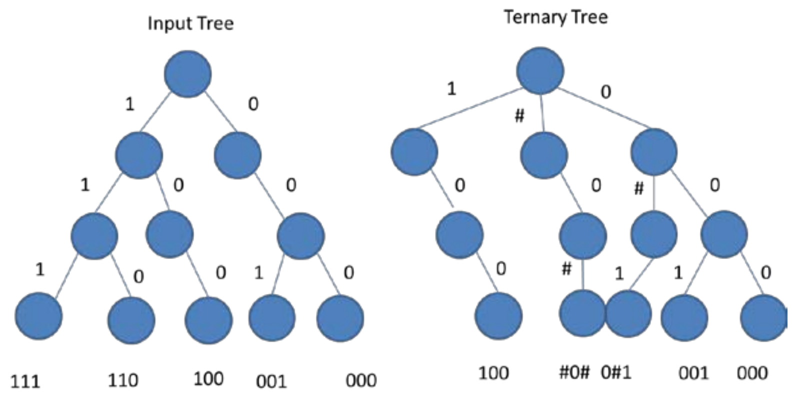
(2) Combine Two Trees

As discussed above, it is valuable to try combining Input Tree with Ternary Tree. Also this approach is the most important one in this thesis which has speeded up the matching part a lot.

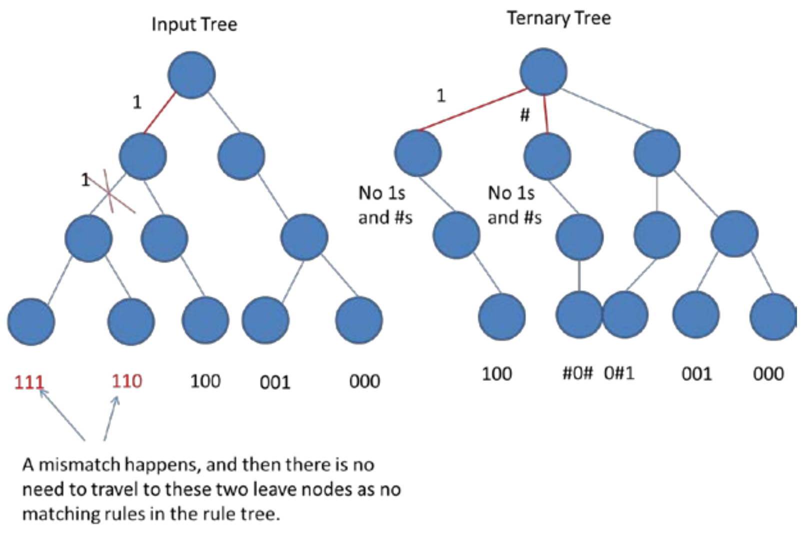
First of all, to combine them together, an analysis of their algorithms in the process would be needed.

For the Ternary Tree, the algorithm for finding the match set is based on a recursion process which has been discussed in the Algorithm 3.3. For the Input Tree, the algorithm for finding the match set is doing recursively traversal through the whole tree and finds matched rules. So they are both using similar recursive traversal in the process of finding match set, it would be possible to combine those two trees if the traversal could be mixed together.

The hardest thing for implementing this idea is how to synchronize the process of algorithms in both of them. In this design, there would be two separate recursive functions for the two trees and one of them would call another one at the runtime. Those pictures below would show how this complex algorithm works and try to prove this idea is reasonable.



For example, the two trees in the system are like this.



Input Tree
Ternary Tree
1
#
No 1s and #s
No 1s and #s
111
110
100
001
000
100
#0#
0#1
001
000
A mismatch happens, and then there is no need to travel to these two leave nodes as no matching rules in the rule tree.

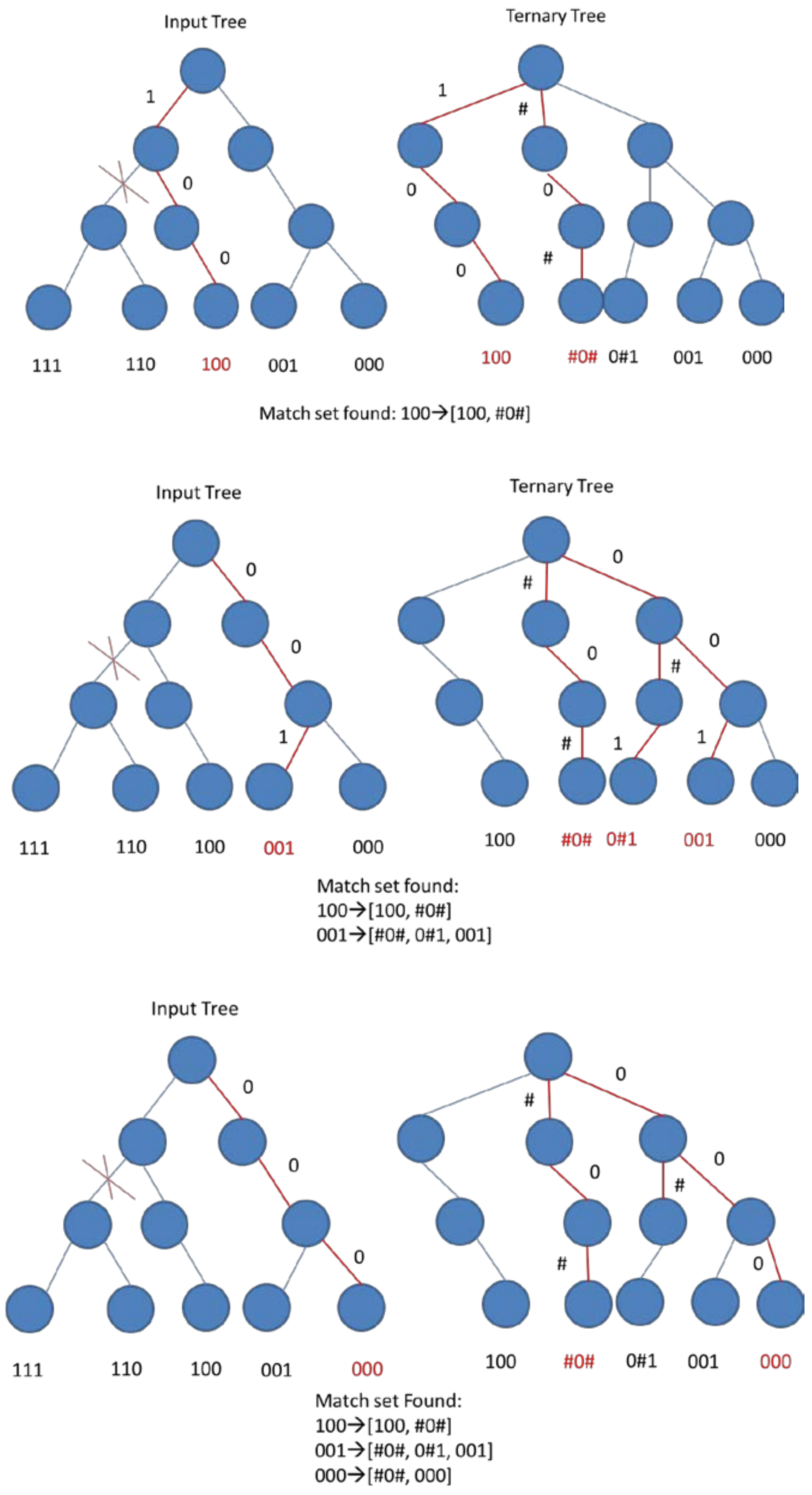


Figure 3-21: Algorithm of Matching in Combining Input Tree and Ternary Tree

Through this algorithm for matching, the efficiency would be speeded up a lot. The improvements would basically come from three features which would be discussed later in the discussion section.

Here is the simplified algorithm for Input Tree in words:

```
Current_in_node = inputroot;
Current_tt_node = ttreeroot;


GetInput(current_ in_node, current_tt_node)//Recursion in Input Tree
{
   If(current_tt_node is a leaf)
   {
       Add it into the matchset;
   }
   Else
   {
     If(current_tt_node has a left child)
     {
          //Inform the Ternary Tree, the node in  Input Tree has chosen to go left
        GetMatchRules(current_in_node's left child, current_tt_node, 1 )
      }
        If(current_tt_node has a right child)
     {
          //Inform the Ternary Tree, the node in  Input Tree has chosen to go right
        GetMatchRules(current_in_node's right child, current_tt_node, 1 )
      }
   }
}

GetMatchRules (current_ in_node, current_tt_node, flag)
 //Recursion in TTree, the flag is to record the choice in Input Tree
{
   If(flag==1)
   {
     Left = current_tt_node's left child;
     Middle = current_tt_node's middle child;
     If(Left and Middle are null)
     {
       This is a mismatch
     }
     If(Left is not null)
      {
        GetInput(current_in_node, Left);
      }
     If(Middle is not null)
      {
        GetInput(current_in_node, Middle);
      }
   }
   Do the same thing for flag =0;
}
```

4. Results and Discussions

This section would show all the test results of the former approaches in section 3. For different methods, different experiments were designed to show whether the methods could actually improve the performance of the matching process. And for each approach, some discussions and analysis would be made for the results of it.

All the experiments are based on only matching process of XCS instead of the whole system, more investigations on applying them to the whole system could be done in the future.

There are two separate algorithms which are responding for generating random rules and inputs in the system. For generating random inputs, we choose every bit from “1” and “0” with the same chances (50%). For generating random rules, we could control the probability of choosing “#” (which is called general percentage), beside it, ‘1’ and ‘0’ would have a same chance to be chosen from.

Also, all the experiments are done on the same computer. Here are the main details of the computer:

CPU	Intel i7-2670QM, 2.20GHZ
Memory	6GB
GPU	NVIDIA GeForce GT 540M

Every result comes from the average of 10 times running.

4.1 Data Pre-processing (Hash Matching)

4.1.1 Experiment Design

The experiment settings are as follows:

Methods in Comparison	Naive Matching, Hash Matching
Rule Population	1000
Rule Producing Strategy	Random
Input Population	10000
Input Producing Strategy	Random
General (“#”) Percentage	0%, 33%, 66%, 99%
Length	10,15, 20

Table 4-1: Experiment Setting of HashMatch

4.1.2 Results of Experiment

The test results are shown here:

Length	General Percentage	Naive Matching (ns)/Ratio1	Total HashMatch (ns)/ Ratio1	Match Part in HashMatch (ns)/Ratio2	Data Pre-Processing in HashMatch (ns)/Ratio2
10	0%	2.43E+08/1	1.43E+07/0.059	1.04E+07/1	3.92E+06/0.377
10	33%	3.37E+08/1	1.76E+08/0.522	9.97E+06/1	1.66E+08/16.633
10	66%	3.48E+08/1	2.64E+08/0.758	3.17E+06/1	2.61E+08/82.332
10	99%	3.68E+08/1	6.77E+08/1.842	2.52E+06/1	6.75E+08/267.934
15	0%	2.29E+08/1	9.41E+06/0.0411	4.99E+06/1	4.43E+06/0.888
15	33%	3.36E+08/1	2.06E+08/0.613	8.73E+06/1	1.97E+08/22.597
15	66%	3.56E+08/1	1.01E+09/2.850	5.06E+06/1	1.01E+09/199.373
15	99%	4.90E+08/1	1.76E+10/35.994	1.13E+07/1	1.76E+10/1552.883
20	0%	2.51E+08	Fail to Run	Fail to Run	Fail to Run

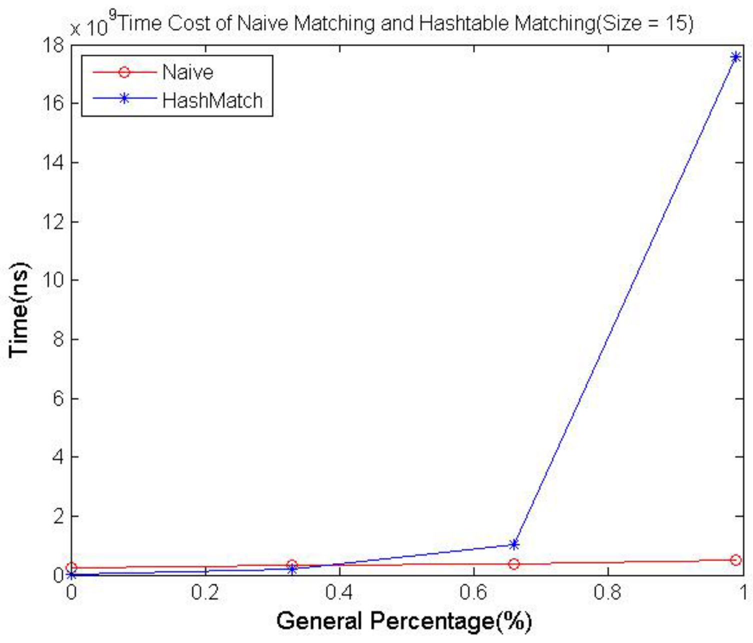
Table 4-2: Test Result of Experiment for HashMatch

Length	General Percentage	Standard Deviation for Naive Matching data(ns)	Standard Deviation for Total Hash Match data(ns)	Standard Deviation for Match part data(ns)	Standard Deviation for data Pre-Processing data(ns)
10	0%	3.1667E+07	4.8263E+06	3.7342E+06	1.1026E+06
10	33%	5.8998E+07	5.7588E+07	1.2805E+06	5.6485E+07
10	66%	3.6736E+07	1.1202E+08	7.5786E+05	1.1130E+08
10	99%	1.1224E+08	1.5029E+08	7.5655E+05	1.4963E+08
15	0%	2.5590E+07	2.1231E+06	1.2057E+06	9.5507E+05
15	33%	3.0411E+07	1.0598E+08	8.6885E+05	1.0578E+08
15	66%	2.6576E+07	1.8101E+08	1.1558E+06	1.8023E+08
15	99%	7.4305E+07	1.9405E+09	1.4887E+06	1.9395E+09
20	0%	N/A	N/A	N/A	N/A

Table 4-2-1: Standard Deviation table of the result

All the results above are from average of results from ten times.

The first column of the table shows the length of a single rule. And the “General Percentage” in the second column shows the percentage of ‘#’ in each rule. After that, columns three and four are the execution time of “naive matching” and “HashMatch”. Finally, the last two columns are the execution time of the two sub-parts of “HashMatch” which are “Match” and “Data Pre-Processing”. The part “Match” stands for the time cost by finding match set while the part “Data Pre-Processing” stands for the time cost by data compilation. There are two ratios in this table, “Ratio1” and “Ratio2”. “Ratio1” is for the comparison between “Naive Match” and “Total Hash Match” and “Ratio2” is for telling how the time proportion is spent by the two parts in “Hash Match”. The following figures are showing the same situation in the table:



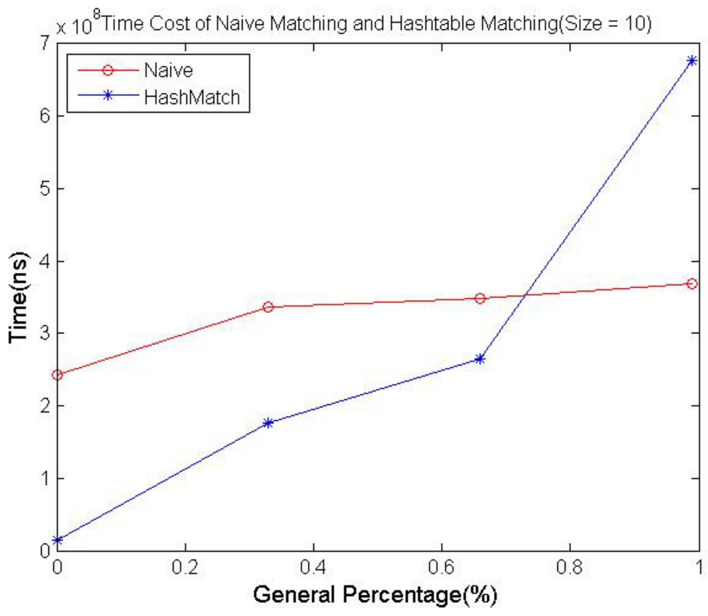


Figure 4-1: Test Result of Experiment for HashMatch

4.1.3 Analysis of Results

In this section, the results of this algorithm would be evaluated and discussed.

Phenomenon:

The algorithm “Hash Match” wins the naive one easily while the “General Percentage” is low, but with the increase of complexity of the rules, this method becomes worse and worse. And for the experiment with size of 20, the “Hash Match” would not work.

As can be seen from the table, actually, the “Match” part in “HashMatch” always cost little time whereas the time cost by “Data Pre-Processing” increases obviously with the increase of “General Percentage”. And when the “General Percentage” is high, the process of “Data Pre-Processing” would cost more than a thousand folds of time over the “Match” part.

Analysis:

It is easy to understand that the “Match” part is always good because this process in “HashMatch” algorithm is just for finding the correct place of match set which has already been prepared by the “Data Pre-Processing”. As the most time consuming part, “Data Pre-Processing” is for finding all possible inputs for all the rules stores in the system. In this case, it is clear that it would increase a lot with the increase of “General Percentage”. For example, a rule “10#” would create possible inputs “100” and “111” whereas a rule “1##” would create “100”, “111”, “101” and “110”. This is the deadly drawback of this algorithm. Because of this mechanism, it would cost too much time and too many spaces for preparing for the match sets. That is the reason why this algorithm become very slow while facing complex rules. Also, it is easy to understand why this algorithm would not run with a rule whose size is more than 20, because there would be a very big space needed for storing all the possible inputs form the system.

Conclusion:

This algorithm is easy to use and implement, but there are two deadly drawbacks for it. Those drawbacks have decided this method could not be used in XCS. However, this method could be possibly used in another system which does not usually change the rules (XCS usually changes its rules), because in that case, the algorithm would only need to do the “Data Pre-Processing” once and use the “Match” part forever.

4.2 Recursion Optimisation in Ternary Tree

In this section, the optimisation towards the recursively method “getmatchset()” in Ternary Tree would be tested.

4.2.1 Experiment Design

Methods in Comparison	Naive Matching, Hash Matching
Rule Population	1000
Rule Producing Strategy	Random
Input Population	10000
Input Producing Strategy	Random
General (“#”) Percentage	33% (The less repeat rule databse)
Length	20
Strategies to replace recursive function	OwnQueue, OwnStack (Array Version), OwnStack (LinkedList Version), JAVA Queue, JAVA LinkedList

Table 4-3: Experiment Setting of Recursion Replacement

4.2.2 Results of Experiment

Here is the result of Recursion Replacement:

Methods for finding match set	Ternary Tree Matching (ns)/Ratio	Standard Deviation
Recursion	1.27E+08/1	1.5584E+07
OwnQueue	6.45E+07/0.506	6.0069E+06
OwnStack (Array Version)	6.24E+07/0.489	1.0322E+07
OwnStack (LinkedList Version)	7.33E+07/0.575	8.6030E+06
JAVA Queue	7.52E+07/0.590	1.2472E+07
JAVA LinkedList	7.76E+07/0.609	1.0698E+07

Table 4-4: Test Result of Experiment for Recursion Replacement

This is the results of experiment of Recursion Replacement, the first column shows the approaches used for comparison and the second column is the time cost by those methods to find match set.

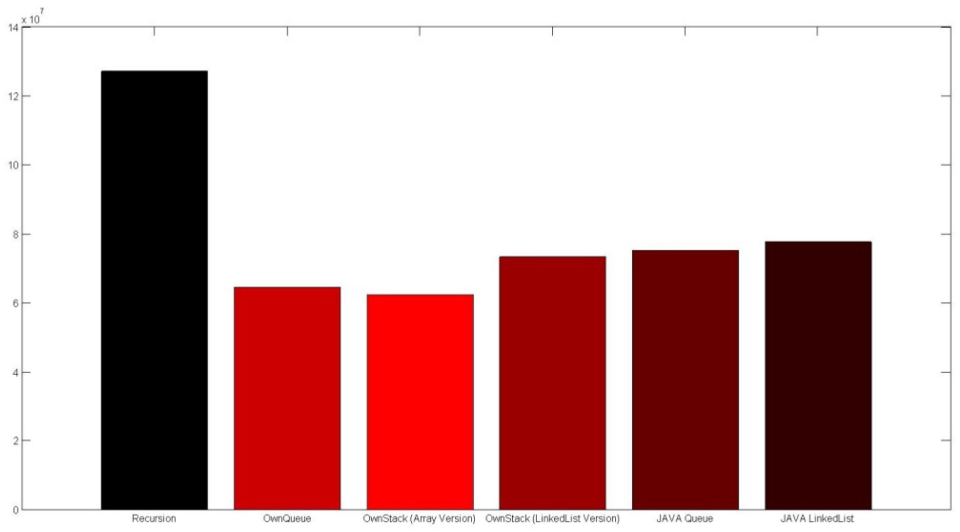


Figure 4-2: Test Result of Experiment for HashMatch

4.2.3 Analysis of Results

Phenomenon:

As could be seen from the table and figure, it is clear that all of the approaches could accelerate the method “getmatchset()”. Among them, the best one is “Own Stack (Array Version)” and a similar one is “Own Queue”.

Also a phenomenon could be seen is that the original data structures in JAVA didn’t perform so well. The reason for this has been discussed in section 3.3.2.

Analysis:

Due to the fact that a recursion function does not usually have a good efficiency, it is clear that all of the replacements could be helpful. The two best approaches are “Own Stack (Array Version)” and “Own Queue” which may because their data structures are both based on array which is the most efficient data structure.

Conclusion:

After the experiment, it is easy to see all the new replacements could perform better than the original recursive method in Ternary Tree. Among them, the nest method is the “Own Stack” (Array Version). One advantage of this approach is that this approach could be easily applied to the Ternary Tree directly to optimise its process of finding match set.

4.3 Object Pool

In this section, some experiments on testing object pool would be designed. An important thing to mention is that since only the matching parts of XCS has been implemented for all the experiments in this thesis, the process of updating (generating new rules and deleting bad rules) won't be tested. So the "recycle" function won't be used because deleting only happens in updating process. Instead, to test the object pool, this experiment aims at testing the time of building a tree. At the beginning, 50000 objects are pre-built in the object pool (time included), and then the system borrows objects from it to build the tree.

4.3.1 Experiment Design

The experiment of object pool is for building the tree, here is the design setting of it.

Methods in Comparison	Ternary Tree and Object Pool Ternary Tree
Rule Population	1000
Rule Producing Strategy	Random
Input Population	10000
Input Producing Strategy	Random
General ("#") Percentage	0%, 33%, 66%
Length	10, 30

Table 4-5: Experiment Setting of HashMatch

One thing should be noticed is that the experiment doesn't choose to analyse the situation when "General Percentage = 99%". It is because that situation is too easy for building a Ternary Tree (Most of the rules are all '#', which means a Ternary Tree need only to have a middle line) which is like a special case.

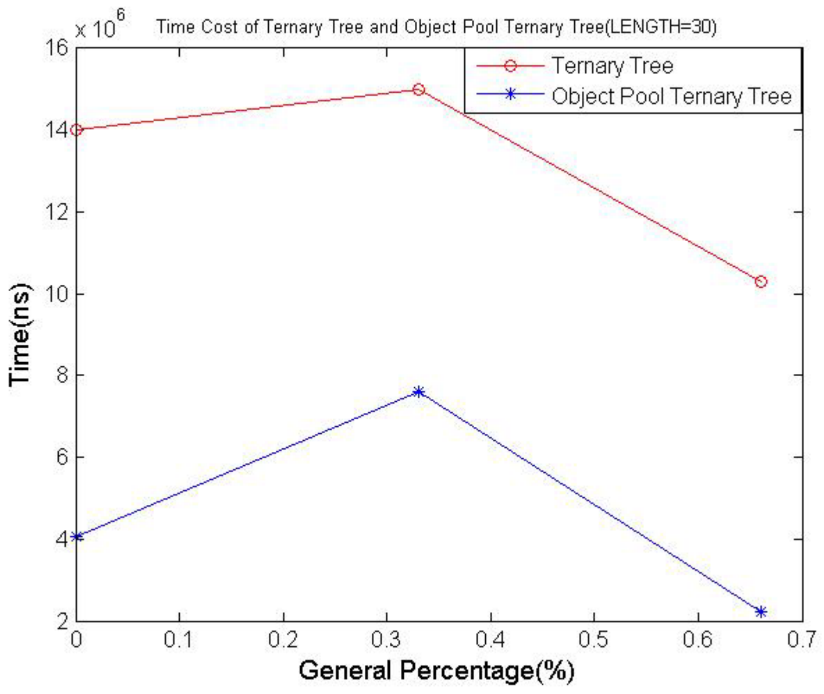
4.3.2 Results of Experiment

Length	General Percentage	Ternary Tree Building (ns)/Ratio	Standard Deviation of Ternary Tree Building	Object Pool Ternary Tree Building (ns)/ Ratio	Standard Deviation of Object Pool Ternary Tree Building
10	0%	1.40E+07/1	3.3932E+06	4.05E+06/0.289	1.0666E+06
10	33%	1.50E+07/1	3.4186E+06	7.59E+06/0.507	3.1632E+06
10	66%	1.03E+07/1	3.8835E+06	2.22E+06/0.215	9.6024E+05
30	0%	2.05E+07/1	3.6951E+06	1.79E+07/0.873	3.3771E+06
30	33%	2.22E+07/1	4.2186E+06	1.96E+07/0.884	3.7450E+06
30	66%	2.13E+07/1	4.3851E+06	1.83E+07/0.861	4.4318E+06

Table 4-6: Test Result of Experiment for Object Pool

All the results above are from average of results from ten times. The first column is the length of each rule in the database. And the second column is the General ('#') Percentage. The columns three and four are the comparison between traditional Ternary Tree Building time cost and Object Pool Ternary Tree Building time cost. Also the ratio in the table show the proportion between them which is clearer.

The following figures are the same situation in the table which is clearer for the analysis.



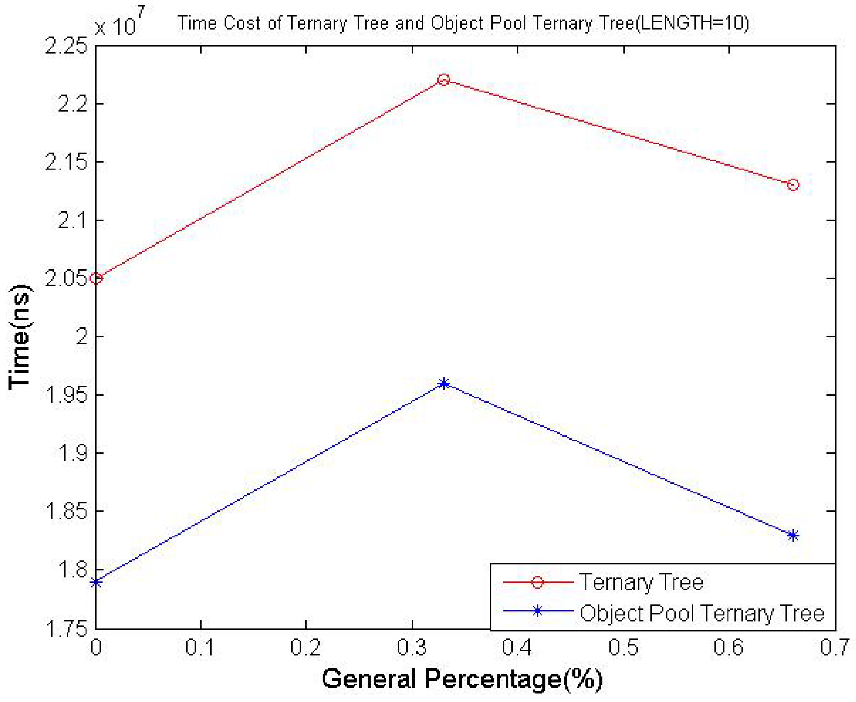


Figure 4-3: Test Result of Experiment for Object Pool

4.3.3 Analysis of Results

In this section, the results from the experiment of Object Pool would be analysed.

Phenomenon:

It is clear from the figure that the object pool could offer some improvements for building the Ternary Tree. But the acceleration is not so obvious when longer rules are used (only a speedup of 20%).

Also there is a special phenomenon found during the test which is the highest time consumption is always at “General Percentage =33%”.

Analysis:

As shown in the figure and table above, the improvement of object pool could be found but not too much. This problem may because that the node in a Ternary Tree is too simple (only contains two pointers and a String actually), so it won’t cost too much time on initializing a new node. Out of the project, some other tests towards object pool shows that the more complex a node is, the better performance the object pool could offer.

For the phenomenon about the highest point in the figure, it is because while “General Percentage = 33%”, the system would be the most random one. In this situation, ‘1’, ‘0’ and ‘#’ would have the same chance (33.33%) to appear in a rule. So the Ternary Tree would get few repeat rules while building which means having a larger chance to add new paths.

Conclusion:

As discussed above, the Object Pool could bring some acceleration to the process of building tree but not too much because of the nodes in Ternary Tree is too simple. Also, more investigations should be done on the recycle process in object pool. Since building a tree is just using “borrowing” operation in Object Pool, it is not clear whether the “recycling” operation would affect the efficiency of the system or not.

4.4 Parallel Computing in XCS

4.4.1 Experiment Design

In this thesis, the idea of using CUDA to optimise XCS would not be tested because that idea is based on the paper written by Lanzi and Loiacono and already tested by them in that paper (Lanzi & Loiacono, 2010). The experiment in this section would be focused on the Fork/Join Parallel Ternary Tree.

As the "Parallel Ternary Tree" is an optimisation based on "Ternary Tree", in the experiment it would be only needed to be compared with traditional "Ternary Tree".

The experiment settings are as follows:

Methods in Comparison	Ternary Tree, Parallel Ternary Tree
Rule Population	1000
Rule Producing Strategy	Random
Input Population	1000,10000
Input Producing Strategy	Random
General ("#") Percentage	0%, 33%, 66%, 99%
Length	100

Table 4-7: Experiment Setting of HashMatch

4.4.2 Results of Experiment

The test results are shown here:

Input Population	General Percentage	Ternary Tree (ns)/Ratio	Standard Deviation for Ternary Tree	Parallel Ternary Tree (ns)/Ratio	Standard Deviation for Parallel Ternary Tree
1000	0%	7.00E+06/1	1.6730E+06	4.79E+06/0.683	1.1787E+06
1000	33%	4.28E+07/1	1.0928E+07	5.70E+06/0.133	1.5598E+06
1000	66%	4.59E+07/1	8.9743E+06	5.89E+06/0.128	1.1971E+06
1000	99%	4.02E+07/1	8.1607E+06	7.47E+06/0.186	2.0628E+06
10000	0%	2.42E+07/1	4.7725E+06	1.62E+07/0.670	2.4260E+06
10000	33%	8.87E+07/1	1.6906E+07	2.23E+07/0.251	2.5818E+06
10000	66%	1.00E+08/1	1.1888E+07	2.45E+07/0.245	3.5393E+06
10000	99%	1.57E+08/1	1.3467E+07	5.00E+07/0.319	1.0254E+07

Table 4-8: Experiment Results of Parallel Ternary Tree

All the results above are from average of results from ten times. The first column of the table shows the number of inputs. And the “General Percentage” in the second column shows the percentage of ‘#’ in each rule. Finally, columns three and four are the execution time of “Ternary Tree” and “Parallel Ternary Tree”. The “Ratio” in this table shows the difference between them.

The following figures are showing the same situation in the table:

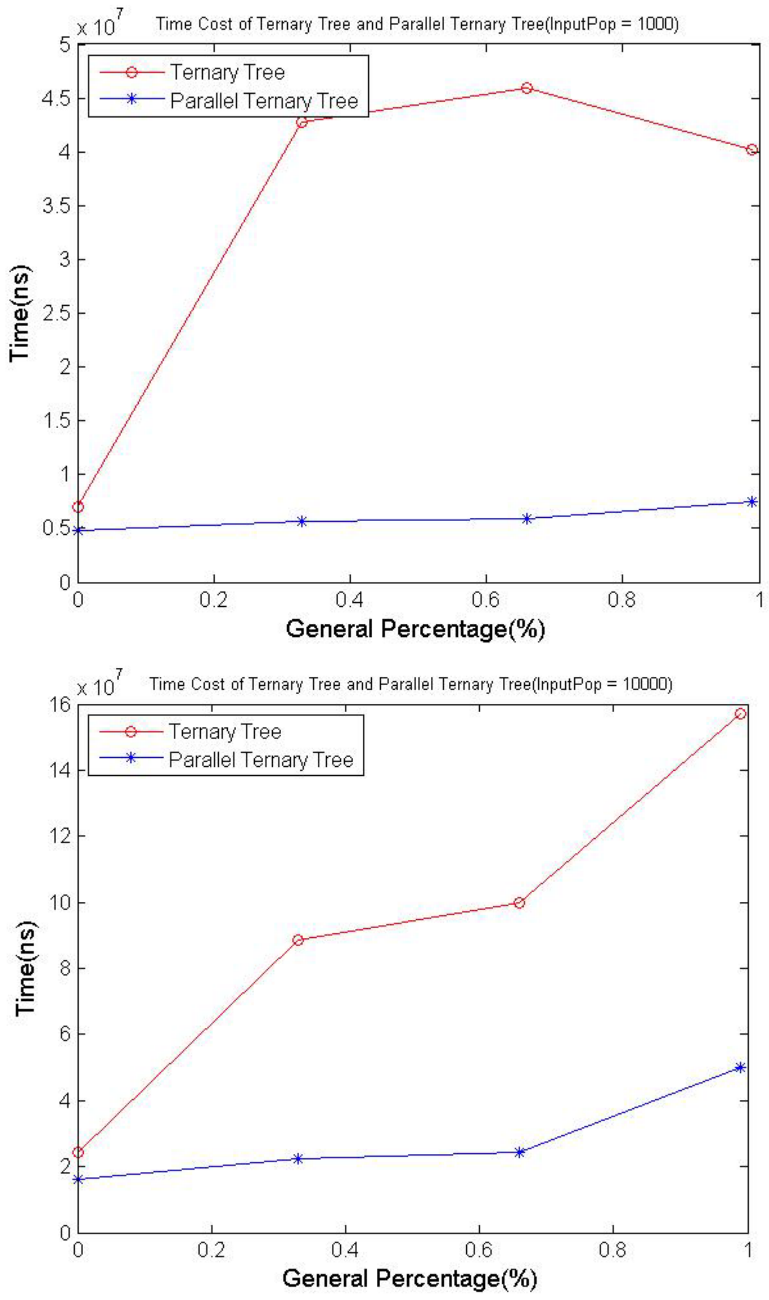


Figure 4-4: Test Result of Experiment for Parallel Ternary Tree

4.4.3 Analysis of Results

In this section, the results of Parallel Ternary Tree would be evaluated and discussed.

Phenomenon:

This table shows clearly that all the results from "Parallel Ternary Tree" are better than the traditional "Ternary Tree". Also a feature could be seen is that it is hard to find a stable speed up among the experiments. Most of the speedup is around four times.

Analysis:

This is a successful method for improving the performance of Ternary Tree. As could be seen from the results, all of the situations have shown 2-10 times of speed up. An important thing should be highlighted here is that it would be hard to completely evaluate a parallel computing algorithm. It is because usually a parallel computing algorithm would depend on so many factors at the runtime like "CPU Load", "Number of Cores in CPU", "Memory Spaces". So there would be hardly to get stable results from experiments. And the reason for the most of the speedup is around four is because the test machine used has four cores. This could also be a proof for explaining how parallel computing would depend on the architecture of CPU.

Conclusion:

This method is a useful solution for optimising Ternary Tree using parallel computing. Before this, most of parallel computing approaches for XCS are based on C/C++, it is an attempt to optimise XCS with parallel computing JAVA. It is valuable to do so because the promoted XCS called JavaXCSF is based on JAVA (Stalph & Butz, 2009). What should be investigated about this algorithm is how to find a stable situation to get an accurate stable improvement.

4.5 Input Tree

This is the most important approach tried in this thesis. In this section, the test result of this method would be evaluated and discussed.

4.5.1 Experiment Design

Because this approach would be the most important one in this project, the experiment for it would do comparison among three different matching methods.

Methods in Comparison	Naive Match, Ternary Tree, Input Tree
Rule Population	5000
Rule Producing Strategy	Random
Input Population	2000,10000,20000,50000
Input Producing Strategy	Random
General (“#”) Percentage	33%, 66%
Length	50

Table 4-5: Experiment Setting of Input Tree

4.5.2 Results of Experiment

The test results are shown here:

Input Population	General Percentage	Naive Matching (ns)/Ratio	Ternary Tree (ns)/Ratio	Input Tree (Including Tree building time)+ Ternary Tree (ns)/Ratio
2000	33%	1.01E+08/1	4.46E+07/0.440	3.52E+07/0.348
2000	66%	3.72E+08/1	1.44E+08/0.386	7.65E+07/0.205
10000	33%	3.94E+08/1	1.49E+08/0.377	8.91E+07/0.226
10000	66%	4.12E+08/1	1.60E+08/0.389	9.38E+07/0.227
20000	33%	7.81E+08/1	1.64E+08/0.210	1.01E+08/0.129
20000	66%	8.24E+08/1	2.03E+08/0.246	1.23E+08/0.149
50000	33%	1.86E+09/1	2.74E+08/0.147	1.69E+08/0.091
50000	66%	1.95E+09/1	3.10E+08/0.159	1.86E+08/0.095

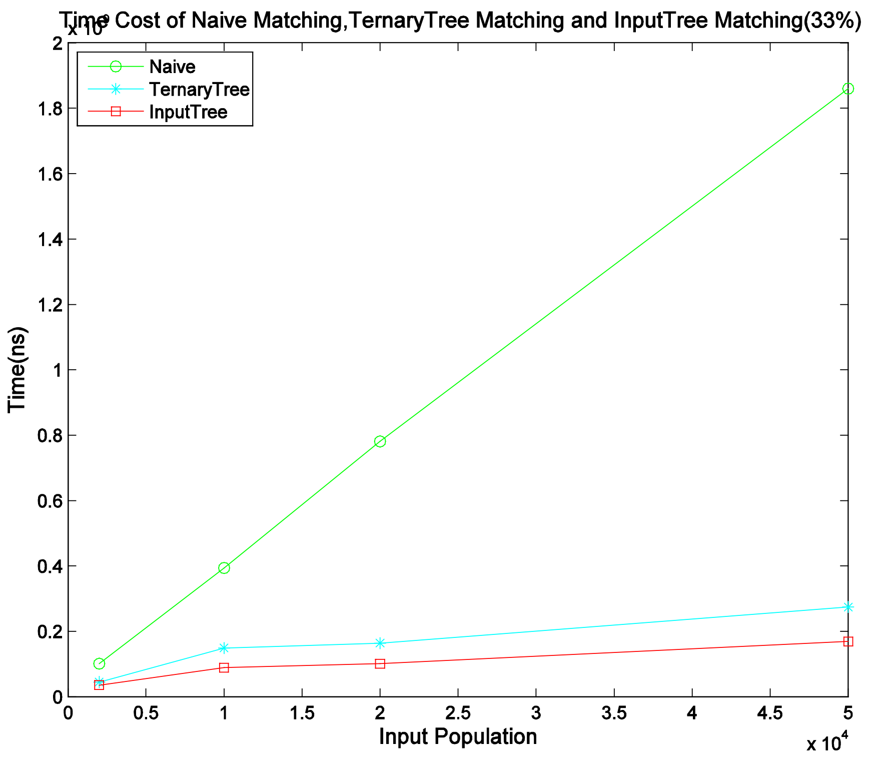
Table 4-6: Experiment Results of Input Tree

Input Population	General Percentage	Standard Deviation for Naive Matching	Standard Deviation for Ternary Tree	Standard Deviation for Input Tree and Ternary Tree
2000	33%	1.4209e+007	1.0322e+007	1.0286e+007
2000	66%	3.0909e+007	1.8321e+007	9.5820e+006
10000	33%	4.9090e+007	4.0942e+007	1.4628e+007
10000	66%	2.1382e+007	3.0878e+007	2.7973e+007
20000	33%	6.6926e+007	1.3241e+008	7.1430e+007
20000	66%	1.0135e+008	1.3284e+008	2.4612e+007
50000	33%	2.1926e+008	9.0525e+008	3.3462e+007
50000	66%	1.7922e+008	9.3393e+008	5.0967e+007

Table 4-6-1: Standard Deviation Table of The Experiment

All the results above are from average of results from ten times. The first column of the table shows the number of inputs. And the “General Percentage” in the second column shows the percentage of ‘#’ in each rule. Finally, columns three, four and five are the execution time of “Naive Matching”, “Ternary Tree” and “Input Tree”. The “Ratio” in this table shows the difference among them.

Here are the figures for all experiments above which would show the results in a clearer way.



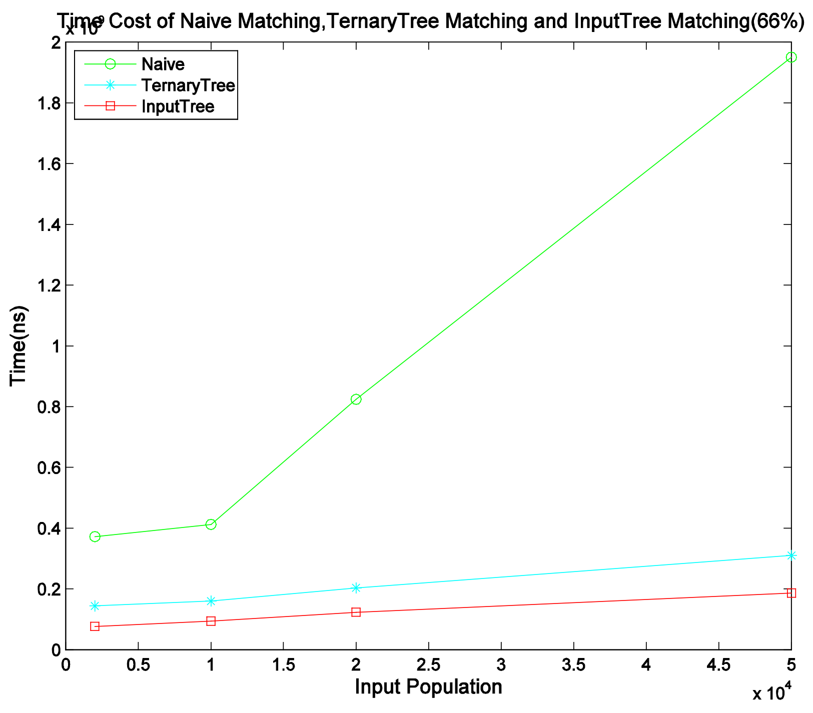


Figure 4-5: Test Result of Experiment for Input Tree

4.5.3 Analysis of Results

In this section, the results of the experiment on "Input Tree" would be evaluated and discussed.

Phenomenon:

According to the results in the table, this approach is quite useful for accelerating the matching process especially compared with the naive matching algorithm. It even gets an acceleration of more than 20 times in some cases.

Analysis:

As discussed before in the method section, it is know that this approach has a good efficiency mostly because its data structure. There are three features which would explain why this method has a good efficiency.

(1) Unchangeable Tree's Level. Usually, there would be several ten-thousands of inputs in XCS. If the normal Naive Match was used, there would be the same number of loops required which is inefficient. But in an "Input Tree", the repeated inputs would be ignored because of its data structure. No matter how many inputs there would be in XCS, the number of leaves could only depend on the length a rule. For example, if the length of input is 3, there would at most be 2^3 leaves in a tree. No

matter how many inputs inserted, the Input Tree won't change any more after it has become a full tree.

(2) Batch Matching. This is the original thought at the beginning and has been achieved by this structure. For any paths in the input tree, they may not only stand for one bit in a single rule but also stand for the same bits in all the rules which pass through this path. An example could be like this:

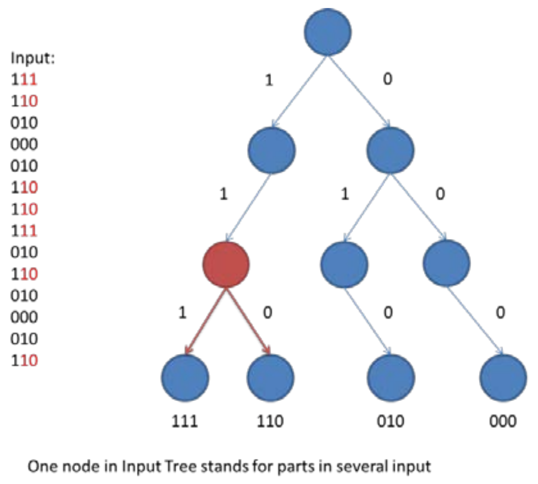


Figure 4-6: Each node in Input Tree stands for several inputs

So, doing every step of match in Input Tree equals to doing several for loops in naive matches at the same time.

(3) Automatic Prune. This feature also reflects the good design of Input Tree. As the algorithm discussed before in section 3.6.2, if a mismatch found in a node in Input Tree, then all of its sub-nodes would be ignored in the later match. This is efficient because each node in the Input tree stands for several rules (same theory in Batch Matching), if one node was deleted, it equals to deleting several rules at the same time in Input Tree.

Conclusion:

Input Tree is a suitable approach to be combined with Ternary Tree to accelerate the matching process a lot. This method could also be possible used with naive matching, but more investigation would be needed.

5. Conclusion

In this section, a summary of this project would be made.

In this project, there have been mainly five methods tried to optimise the match process in XCS. Among them, “HashMatch” and “Input tree” try to find different algorithms for matching whereas the other three methods are based on optimising Ternary Tree.

“HashMatch” has been proved not so useful for the XCS because it needs too many spaces and have a bad efficiency for updating rules.

“Recursion replacement” has turned out to be useful for improving the matching method in Ternary Tree. Although the acceleration is not so big, the advantage of it is that this optimisation could be easily applied the system without any changes of the interfaces.

“Object pool” is a classical way to optimise data structures and also useful for optimising Ternary Tree. This decreases the cost of building a Ternary Tree and updating of the rules.

“Parallel Computing” is also a classical way to optimise algorithms. In this project, a new parallel computing method in JAVA was invented for optimising Ternary Tree. It transfers the method of “getmatchset()” into a parallel one. As shown in the test result, this method is quite successful for optimising Ternary Tree.

“Input Tree” is the invention from this project which aims at implementing batch matching algorithm. It could be both used with traditional XCS and Ternary. This thesis also tried to combine it with Ternary Tree and got a good result which could be seen from the test result.

6. Future Work

In this section, some possible future work from this thesis will be discussed.

For the “Object Pool” approach, although it has a good performance in the test, the operation of recycling objects has not been included. More works on the recycle part should be needed to ensure it works well.

For the “Parallel Computing”, it is hard to get an average acceleration from it. Maybe more tests would be needed to calculate a stable acceleration of it.

For the “Input Tree”, this idea is successful in Ternary Tree. However it has not been tested to work with the traditional rule database, some future works could be about this. Also this idea could be possibly applied to other areas in machine learning as well, but it needs more investigations.

A future work would be combine any of those approaches together, it is like combining “Input Tree” and “Ternary Tree” could offer a better performance.

This project has come up with several successful methods for optimising the system. Although it would be better to match faster, for the whole system, it would be hard to fully solve the efficiency problem because the other parts of system would become bottleneck after this part has been fully optimised. So it would be better to optimise other parts in XCS as well in the future’s work.

Bibliography

Alpaydin, E., 2004. *Introduction to Machine Learning.* Cambridge,MA 02142: Massachusetts Institute of Technology.

Barning, F., 1963. *On Pythagorean and quasi - Pythagorean triangles and a generation process with the help of unimodular matrices,* Amsterdam Afd. Zuivere Wisk.: Math. Centrum .

Bull, L., 2004. Learning Classifier Systems: A Brief Introduction. In: *Applications of Learning Classifier Systems.* Bristol: Springer Berlin Heidelberg, pp. 1-12.

Butz, M. V., Lanzi, P. L., Llor `a, X. & Loiacono, D., 2008. An Analysis of Matching in Learning Classifier Systems. In: *Proceedings of the 10th annual conference on Genetic and evolutionary computation.* Atlanta, GA, USA: ACM, pp. 1349--1356.

Chitty, D. M., 2012. Fast parallel genetic programming: multi-core CPU versus many-core GPU. *Soft Computing - A Fusion of Foundations, Methodologies and Applications,* 16(10), pp. 1795-1814.

Grama, A., Gupta, A., Karypis, G. & Kumar, V., January 16, 2003. *Introduction to Parallel Computing, Second Edition.* s.l.:Addison Wesley.

Holland, J. H., 1975. *Adaptation in Natural and Artificial Systems.* University of Michigan: Massachusetts Institute of Technology.

Holland, J. H. & Reitman, J. S., 1977. Cognitive systems based on adaptive algorithms. *ACM SIGART Bulletin*, June, pp. 49-49.

John L. Hennessy & David A. Patterson;, 2006. *Computer Architecture: A Quantitative Approach.* U.S.A: ELSEVIER.

Kamran Karimi, Neil G. Dickson & Firas Hamze, 2010. *A Performance Comparison of CUDA and OpenCL,* Canada: D-Wave Systems Inc..

Kovacs, T., 2002-06-04. Learning classifier systems resources. *Soft Computing - A Fusion of Foundations, Methodologies and Applications,* 6(3), pp. 240-243.

Kovacs, T. M. D., January, 2002. *A Comparison of Strength and Accuracy-Based Fitness in Learning Classifier Systems,* Birmingham, United Kingdom: University of Birmingham.

L.B. Booker, D. G. J. H., September 1989. Classifier systems and genetic algorithms. *Artificial Intelligence*, 40(1-3), pp. 235-282.

Lanzi, P. & Loiacono, D., 2010. Speeding Up Matching in Learning Classifier Systems Using CUDA. In: B. J. Bacardit, et al. eds. *Learning Classifier Systems.* Berlin / Heidelberg: Springer, pp. 1-20.

Lea, D., 2000. A Java Fork/Join Framework. In: *Proceedings of the ACM 2000 conference on Java Grande.* Proceedings of the ACM 2000 conference on Java Grande: ACM, pp. 36-43.

Llor `a, X. & Sastry, K., 2006. Fast Rule Matching for Learning Classifier Systems. In: *Proceedings of the 8th annual conference on Genetic and evolutionary computation.* New York, NY, USA: ACM, pp. 1513--1520.

McClanahan, C., 2010. *History and Evolution of GPU Architecture.* s.l.:Georgia Tech,College of Computing.

NVIDIA, 4/16/2012. *NVIDIA CUDA C Programming Guide.* s.l.:NVIDIA.

PREE, W., 1995. *DESIGN PATTERNS FOR OBJECT-ORIENTED SOFTWARE DEVELOPMENT.* Wokingham, England: Addison-Wesley Publishing Co..

Quinn, M. J., 2004. *Parallel programming in C with MPI and OpenMP.* 1st ed. New York: McGraw-Hill Higher Education.

Shane Ryoo, Christopher I. Rodrigues & Sara S. Bagh, 2008. *Optimization Principles and Application Performance Evaluation of a Multithreaded GPU Using CUDA,* U.S.A: Center for Reliable and High-Performance Computing, University of Illinois.

Sigaud, O. & Wilson, S. W., 2007. Learning Classifiers Systems: A Survey. *Soft Computing - A Fusion of Foundations, Methodologies and Applications,* 01 09, pp. 1065-1078.

Snir, M. et al., 1995. *MPI: The Complete Reference.* 1st ed. Cambridge, MA, USA: MIT Press.

Stalph, P. O. & Butz, M. V., 2009. *Documentation of JavaXCSF,* WURZBURG, Germany: University of WURZBURG.

Tsentas, Y., 2010. *Efficient Pattern Matching,* Bristol, UK: University of Bristol.

Tushar Mahapatra & Sanjay Mishra, August 2000. *Oracle Parallel Processing.* s.l.:O'Reilly.

Wilson, S. W., 1995. Classifier Fitness Based on Accuracy. *Evolutionary Computation,* 3(2), pp. 149-175.