ANTONIO JUNIOR DE MATTOS

TEST DATA GENERATION FOR TESTING MAPREDUCE SYSTEMS

Dissertation presented as partial requisite to obtain the Master's degree. M.Sc. program in Informatics, Federal University of Paraná.
Advisor: Prof. Dr. Eduardo C. de Almeida

CURITIBA

2011

ANTONIO JUNIOR DE MATTOS

TEST DATA GENERATION FOR TESTING MAPREDUCE SYSTEMS

Dissertation presented as partial requisite to obtain the Master's degree. M.Sc. program in Informatics, Federal University of Paraná. Advisor: Prof. Dr. Eduardo C. de Almeida

CURITIBA

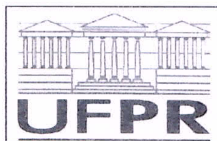
2011

Mattos, Antonio Junior de
Test data generation for testing mapreduce systems / Antonio Junior de Mattos. - Curitiba, 2011.
49 f. : il.; tab., graf.

Dissertação (mestrado) – Universidade Federal do Paraná, Setor de Ciências Exatas, Programa de Pós-Graduação em Informática.
Orientador: Eduardo Cunha de Almeida

1. Algoritmos genéticos. 2. Programação (Computação) - Testes. 3. Software - Validação. I. Almeida, Eduardo Cunha de. II. Título.

CDD 005.14

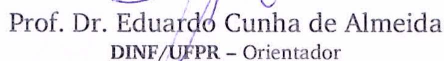


Ministério da Educação
Universidade Federal do Paraná
Programa de Pós-Graduação em Informática

PARECER

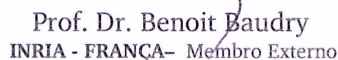
Nós, abaixo assinados, membros da Banca Examinadora da defesa de Dissertação de Mestrado em Informática, do aluno Antonio Junior de Mattos, avaliamos o trabalho intitulado, *" TEST DATA GENERATION FOR TESTING MAPREDUCE SYSTEMS"*, cuja defesa foi realizada no dia 14 de novembro de 2011, às 10:00 horas, no Departamento de informática do Setor de Ciências Exatas da Universidade Federal do Paraná. Após a avaliação, decidimos pela aprovação do candidato.

Curitiba, 14 de novembro de 2011.

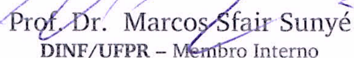


Prof. Dr. Eduardo Cunha de Almeida
DINF/UFPR – Orientador

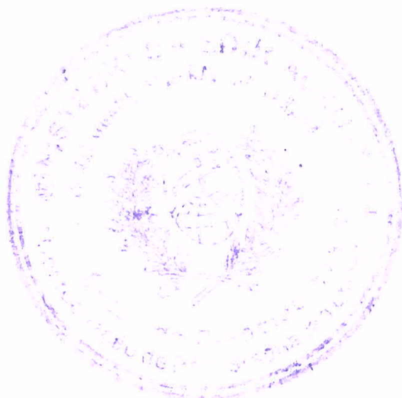
Prof. Dr. Gerson Sunyé
INRIA/UNIVERSITÉ DE NANTES - FRANÇA– Membro Externo



Prof. Dr. Benoit Baudry
INRIA - FRANÇA– Membro Externo



Prof. Dr. Marcos Sfair Sunyé
DINF/UFPR – Membro Interno



CONTENTS

1 RESUMO **1**

1.1 Motivação 1

1.2 Contribuíção 1

1.3 MapReduce 2

1.4 Geração de Dados de Teste 2

1.5 Solução 4

1.6 Experimentos 6

1.7 Conclusão 6

2 INTRODUCTION **9**

2.1 Motivation 9

2.2 Contribution 10

2.3 Outline 10

3 MAPREDUCE **12**

4 TEST DATA GENERATION **16**

4.1 Test Quality Assessment 16

4.1.1 Code Coverage 16

4.1.2 Mutation Analysis 17

4.2 Test Data Generation Techniques 17

4.3 Metaheuristic Search Techniques 18

4.3.1 Hill Climbing 19

4.3.2 Simulated Annealing 19

4.3.3 Evolutionary Algorithms 20

4.3.3.1 Genetic Algorithm 20

4.3.3.2 Generalized Extremal Optimization 21

4.3.3.3 Bacteriologic algorithm 21

4.4 Test Data Generators 23

4.5 Generating Test Data For MapReduce Jobs 25

5 SOLUTION **26**

5.1 MapReduce specific mutation operators 26

5.2 Test data generation 27

5.3 Using MapReduce to generate test data 29

6 IMPLEMENTATION **30**

6.1 Main Components 30

6.2 User Interface 33

7 EXPERIMENTS **35**

7.1 Applying the bacteriological algorithm 36

7.2 Traditional mutants 36

7.3 WordCount 37

7.4 Breadth-First Search 39

7.5 Discussion 44

8 CONCLUSION **45**

LIST OF FIGURES

1.1 WordCount 7
1.2 Breadth-First Search 8

3.1 Execution of Map and Reduce operations 12

6.1 Execution module 31
6.2 Mutant management module 32
6.3 Bacteriologic module 33
6.4 Mutation interface 34

7.1 WordCount 40
7.2 Test Data Size 41
7.3 Breadth-First Search 43

LIST OF TABLES

1.1 Comparação entre algorítmo bacteriológico e algorítmo genético 2

1.2 Comparação entre algorítmo de otimização generalizada extrema e algorítmo genético 3

1.3 Operadores propostos 4

3.1 Word count example 15

4.1 Coverage Summary for Class: Interval. 17

4.2 Comparison between bacteriological and genetic algorithms for a C# parser. 24

4.3 Comparison between GEO and genetic algorithms. 25

5.1 Proposed Operators 27

7.1 Bacteriological Mutants 36

7.2 Mutation Analysis with traditional mutation operators 36

7.3 WordCount Mutants 38

7.4 Input Data Impact Evaluation 38

7.5 Breadth-First Search Mutants 42

7.6 BFS results 43

ABSTRACT

MapReduce is a framework for parallel processing large data sets, which is largely adopted for complex web applications and data processing. The framework proposes a simple interface, based on two high-order functions, allowing the rapid development of large-scale distributed software. Among the many aspects of MapReduce software development, producing reliable, correct and efficient software is an obvious target. We present an automatic test data generation and qualification approach for MapReduce applications, also called jobs. This approach uses an evolutionary algorithm to generate the test data and proposes domain-specific mutation operators to evaluate the quality of the data through mutation analysis. We validated this framework through implementation and experimentation on different MapReduce jobs.

CHAPTER 1

RESUMO

1.1 Motivação

MapReduce [13] tornou-se o padrão de industrial para processamento paralelo de grandes conjuntos de dados. Grandes companhias e institutos de pesquisa utilizam esse *framework* para processarem seus dados.

Como para qualquer outro *software*, teste pode ser utilizado para avaliar a qualidade de aplicações MapReduce, chamadas *jobs*. Porém, *jobs* MapReduce trabalham com grandes quantidades de dados, e gerar dados de teste relevantes que possam revelar problemas na qualidade desses *jobs* é uma grande dificuldade.

Algumas ferramentas de teste para *jobs* MapReduce estão disponíveis [1, 2, 18]. Entretanto, nenhuma delas gera dados de teste.

1.2 Contribuíção

O trabalho apresentado aqui contribui para o estabelecimento de técnicas sistemáticas de teste para *jobs* MapReduce, atráves das seguintes propostas:

- modelos de falha que focam em problemas de *design* em separar uma tarefa entre funções *Map* e *Reduce*.
- uma técnica automática de busca para gerar dados de teste que objetivam essas falhas.
- uma série de experimentos que ilustram a dificuldade de detectar essas falhas e a capacidade da nossa solução em gerar dados de teste relevantes.

1.3 MapReduce

MapReduce é um modelo de programação e *framework* para o desenvolvimento de aplicações paralelas de larga escala. Esse *framework* expõem uma interface baseada em duas funções: *Map* e *Reduce*. Usuários constroem suas aplicações definindo o correto comportamento dessas funções.

Durante a execução o *framework* encarregasse de: 1) copiar a aplicação do usuário entre nodos; 2) particionar os dados de entrada; 3) escalonar a execução da aplicação entre o conjunto de nodos; 4) gerenciar falhas.

1.4 Geração de Dados de Teste

A geração de dados de teste é um componente do teste de *software* que requer grande esforço [29]. Técnicas de geração de dados de teste tentam encontrar dados de entrada que satisfaçam algum requisito de teste. Essas técnicas podem ser classificadas em [16]: aleatória, orientada por caminho, orientada por objetivo e inteligentes.

Por ser um problema de busca, a geração de dados de teste pode se beneficiar de técnicas de busca meta-heurística. Duas técnicas de busca local comumente utilizadas são, [24]: Subida da Colina e Arrefrecimento Simulado. Algorítmos evolucionários também são utilizados para evoluir soluções candidatas, os principais são: algorítmos genéticos [19], otimização generalizada extrema (OGE) [11] e algorítmo bacteriológico [6].

Em trabalhos recentes, algorítmos evolucionários foram utilizados para gerar dados de teste dinamicamente. Nesses trabalhos, os algorítmos bacteriológico e otimização generalizada extrema foram utilizados e comparados com o algorítmo genético. A tabela 1.4 mostra a comparação do algorítmo bacteriológico com o algorítmo genético.

Table 1.1: Comparação entre algorítmo bacteriológico e algorítmo genético

Algorítmo	No. gerações	Pontuação de mutação (%)	Tempo de execução (Hrs)
Genético	200	85	26
Bacteriológico	30	96	2.5

A tabela 1.4 mostra a comparação do algorítmo de otimização generalizada extrema com o algorítmo genético.

Table 1.2: Comparação entre algorítmo de otimização generalizada extrema e algorítmo genético

Programa	Cobertura AG (%)	Cobertura OGE (%)	Execuções AG	Execuções OGE
1	**92.41**	89.19	253640	268171
2	100	100	**109**	651
3	100	100	1853	2277
4	97.76	**98.49**	**45343**	51721
5	**99.83**	99.53	**48286**	51381
6	100	100	**100**	550
7	66.62	**66.67**	799764	**799300**

Para avaliar a qualidade de dados de teste as duas formas mais comuns são: cobertura de código [25] e análise de mutantes [14].

A cobertura de código é uma técnica para medir quanto do código da aplicação é coberto por um conjunto de dados de teste. Entretanto, ela não diz aos desenvolvedores se o código coberto funciona corretamente. Existem diferentes técnicas de cobertura de código, as mais comuns são: cobertura de método, cobertura de expressão, cobertura de condição, cobertura de caminho e cobertura de chamada.

A análise de mutantes é uma técnica de teste baseada em falhas que mede a adequação de dados de teste. A medida de adequação é chamada de pontuação de adequação mutante. A pontuação pode ser usada para indicar a efetividade do dado de teste em revelar falhas. Falhas que representam erros comuns de programadores são inseridos no programa original, criando mutantes. Para avaliar a qualidade do dado de teste, os mutantes são executados com o dado para verificar se as falhas são reveladas.

Enquanto existem algumas ferramentas de teste para *jobs* MapReduce, nenhuma delas gera dados de teste. A combinação de uma dessas ferramentas com um gerador de dados de teste externo somente permitiria a geração de dados de teste unitário, ou seja, sem a intergração com o *framework*.

Operador	**Descrição**
Inserir *Combiner*	Força o uso da função *Reduce* como *Combiner* do *job* MapReduce.
Remover *Combiner*	Remove o *Combiner* do *job*.
Alterar o número de *Reducers*	Força o uso de um número diferente de *Reducers* no *job*.

Table 1.3: Operadores propostos

1.5 Solução

Para verificar se um conjunto de testes pode expor erros em *jobs* MapReduce, nós propomos novos operadores de mutação que são específicos para MapReduce. A tabela 1.3 mostra esses operadores. Uma vez que eles não introduzem mudanças sintáticas, os operadores são próximos aos operadores de mutação semântica [9], que representam uma possível falta de entendimento da linguagem de descrição, ou nesse caso em particular, o paradigma MapReduce.

O algorítmo global para a geração de dados de teste é descrito pelo algorítmo 1. $\mathcal{D}$ representa o conjunto de todos os possíveis dados de entrada para um caso de teste, $\mathcal{K}$ representa o conjunto de pares $\langle d, m \rangle$, onde d é o dado de teste e m é o mutante morto por esse dado, $data^{\mathcal{K}}$ denota o conjunto de todos os dados de teste em $\mathcal{K}$.

Dados, o *job* MapReduce (JUT), um caso de teste (TC) e o conjunto inicial de dados de teste ($initial^{\mathcal{D}}$), nós geramos todos os mutantes para o JUT a atribuímos esse conjunto à variável $living^{\mathcal{M}}$, que armazena todos os mutantes restantes. A geração de mutantes é feita somente uma vez durante o processo. A inicialização continua por atribuir o conjunto de dados de teste inicial para a variavél $improved^{\mathcal{D}}$, o conjunto de dados de teste que é melhorado pelo algorítmo.

Depois da inicialização, um *loop* começa até que o valor de *fitness* ou um número pré-definido de iterações seja atingido. No *loop*, a primeira etapa é calcular os resultados para o novo dado de teste, que são armazendos na variável $expected^{\mathcal{R}}$. A segunda etapa calcula os resultados para todos os novos dados de teste e todos os mutantes restantes, os resultados são armazenados na variável $actual^{\mathcal{R}}$.

Uma vez que todos os resultados estão disponíveis eles são comparados e se os resul-

Algorithm 1: Global Algorithm

Input : $JUT \in Program$
Input : $TC \in Program$
Input : $initial^{\mathcal{D}} \subset \mathcal{D}$
Output: $data^{\mathcal{K}} \subset \mathcal{D}$
Data: $expected^{\mathcal{R}}, actual^{\mathcal{R}} \subset 2^{Result}$
Data: $living^{\mathcal{M}} \subset 2^{Program}$
$living^{\mathcal{M}} \leftarrow GenerateMutants(JUT)$;
$data \leftarrow initial^{\mathcal{D}}$;
$newdata \leftarrow data$;
repeat
    foreach $d \in newdata$ **do**
        $expected^{\mathcal{R}} \leftarrow expected^{\mathcal{R}} + execute(JUT, TC, d)$;
    foreach $m \in living^{\mathcal{M}}, d \in newdata$ **do**
        $actual^{\mathcal{R}} \leftarrow execute(JUT, m, d)$;
    $killed^{\mathcal{M}} \leftarrow MutationAnalysis(expected^{\mathcal{R}}, actual^{\mathcal{R}})$;
    $living^{\mathcal{M}} \leftarrow living^{\mathcal{M}} - killed^{\mathcal{M}}$;
    if $living^{\mathcal{M}} \neq \emptyset$ **then**
        $newdata \leftarrow generate(\mathcal{K})$
    else
        exit loop
until *fitness-reached* $\vee$ *enough-iterations*;
return $data^{\mathcal{K}}$

tados divergirem o mutante em questão é considerado morto. Os resultados divergem se os valores forem diferentes ou se o tempo de execução tem uma diferença não equivalente. Os mutantes mortos são subtraídos de $living^{\mathcal{M}}$.

1.6 Experimentos

Nós avaliamos nossa solução utilizando dois *jobs* MapReduce: WordCount e Breadth-First Search (BFS). Os experimentos foram executados em um *cluster* com 7 nodos.

Os experimentos demostraram que o algorítmo bacteriológico é capaz de evoluir os dados de teste para os dois *jobs*. Os dados de teste melhorados foram capazes de matar todos os mutantes em menos de dez iterações.

Para o WordCount três mutantes foram criados:

- $\mathcal{M}1$ - *Combiner* removido.
- $\mathcal{M}2$ - Usa três *Reducers*.
- $\mathcal{M}3$ - usa seis *Reducers*.

A figura 1.1 mostra a evolução do tempo de execução de $\mathcal{M}0$ e $\mathcal{M}1$ durante as iterações. A diferença do tempo de execução aumenta a medida que o dado de teste evolui. A abordagem também foi aplicada a $\mathcal{M}2$ e $\mathcal{M}3$. Foram necessários oito iterações para matar todos os mutantes.

Para o BFS criamos mutantes similares aos do Wordcount. A figura 1.2 mostra a evolução do tempo de execução de $\mathcal{M}0$ e $\mathcal{M}1$ durante as iterações. Foram necessárias cinco iterações para que todos os mutantes fossem mortos.

1.7 Conclusão

Os experimentos demostraram que nosso algorítmo bacteriológico é capaz de gerar dados de teste que são eficientes em expor decisões de *design* ruins em *jobs* MapReduce. Nós acreditamos que, enquanto a geração de dados de teste baseado em cobertura de caminhos

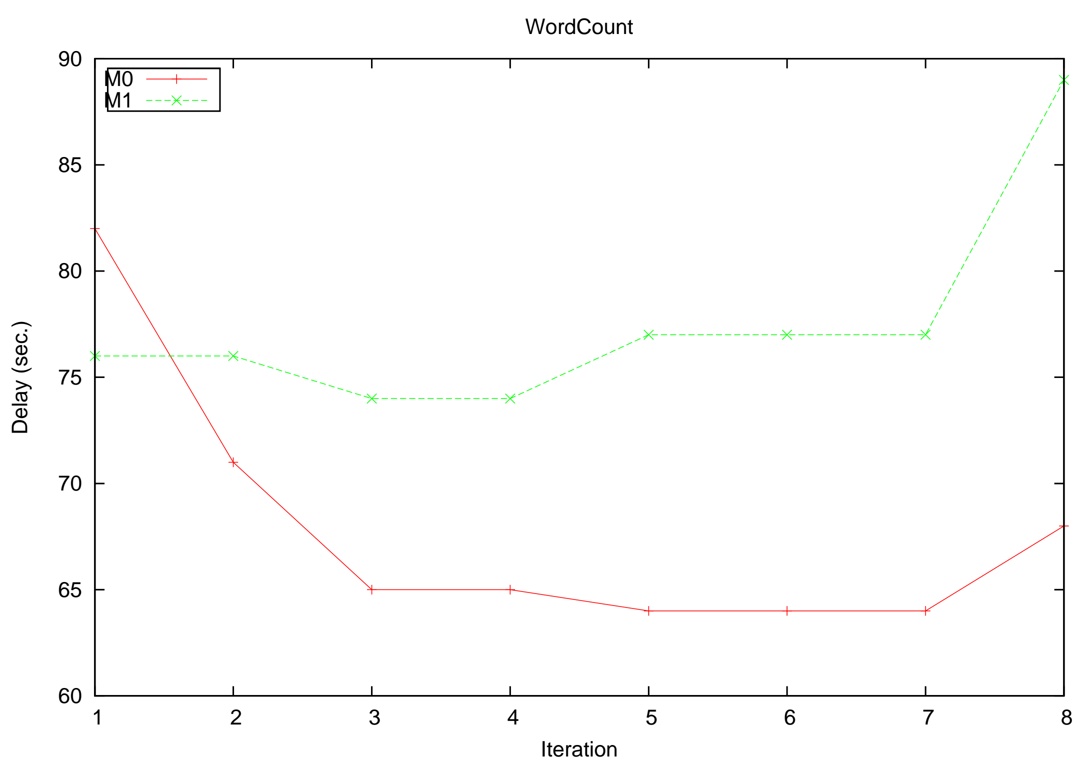


Figure 1.1: WordCount

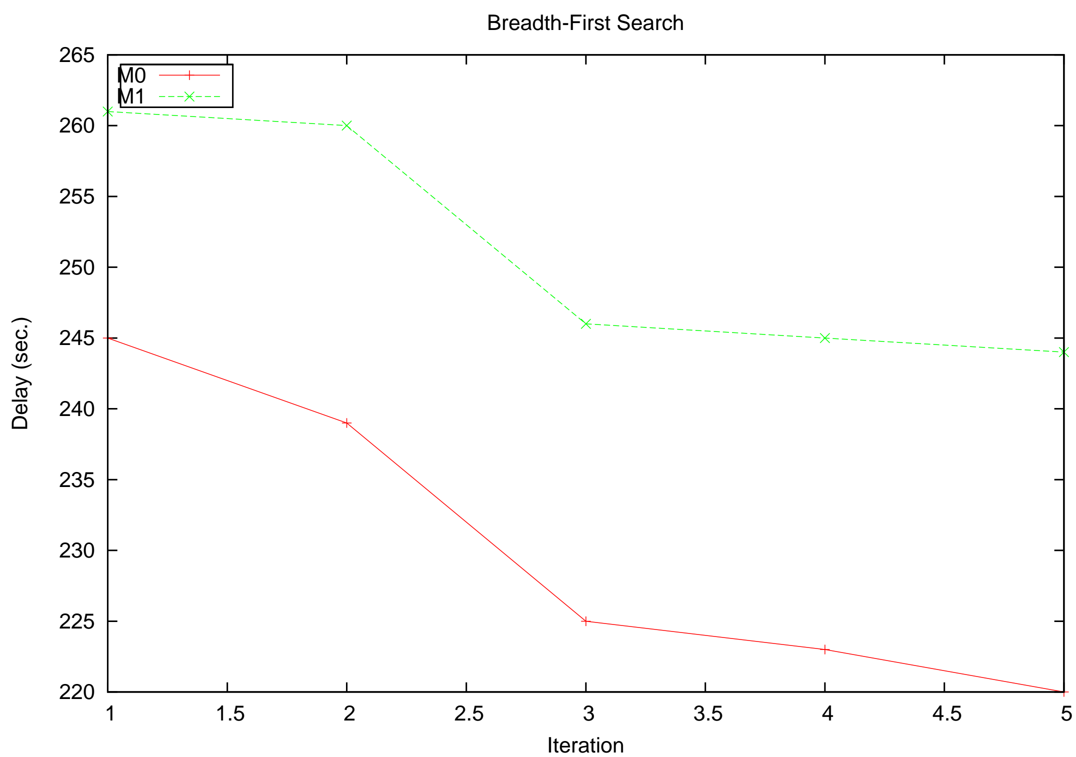


Figure 1.2: Breadth-First Search

pode ser útil para detectar erros funcionais, eles não parecem adaptados para expor erros que são relacionados ao mau entendimento do *framework* MapReduce.

CHAPTER 2

INTRODUCTION

2.1 Motivation

MapReduce [13] became the industry de facto standard for parallel processing. Attractive features such as scalability and reliability motivate many large companies such as Facebook, Google, Yahoo and research institutes to adopt this new programming paradigm. These organizations rely on Hadoop [32], an open-source implementation of MapReduce, to process their information. Besides Hadoop, several other implementations are available: Greenplum MapReduce [17], Aster Data [4], Nokia Disco [27], Microsoft Dryad [20], among others.

MapReduce has a simplified programming model, where data processing algorithms are implemented as instances of two higher-order functions: Map and Reduce. All complex issues related to distributed processing, such as scalability, data distribution and reconciliation, concurrence, fault tolerance, etc., are managed by the framework. The main complexity that is left to the developer of a MapReduce-based application (also called a job) lies in the design decisions made to split the application specific algorithm into two higher-order functions. Even if some decisions may result in a functionally correct application, bad design choices might also lead to poor resource usage.

As for any other piece of software, testing may be used to evaluate the quality of MapReduce jobs. The efficiency of testing to detect quality issues in the jobs is in turn directly related to the quality of the test suite. MapReduce jobs work with large amounts of data, which is a major difficulty to generate relevant test data that can reveal quality issues. If on one hand using large data sets as input data is not scalable, on the other hand a small amount of data may not expose errors related to the large-scale aspects: efficient resource usage, correct merge of data, etc.

A number of `MapReduce` testing tools [1, 2, 18] are available. Despite their differences,

these tools all share the same weakness – they do not generate test data. In this case, developers are responsible to generate it. While basic test data generation is easy, improving its quality may require a prohibitive effort.

2.2 Contribution

We present an original approach for the generation of test data that target resource usage defects in MapReduce jobs. We use an evolutionary algorithm to generate test data and propose semantic mutation operators to evaluate the quality of the data with respect to their ability at detecting issues in the design of the map and reduce functions. This focus on testing design decisions lies on two observations. First, these design decisions are essential to implement MapReduce jobs, and they should be systematically tested along with the functionality of the job. Second, the functionality of map and reduce is usually simple (small functions, simple control and data flow) since most of the complexity of these jobs (distribution, synchronization, etc.) is handled by the framework. Consequently, simple test cases tend to cover 100% of the job under test, can be good at detecting algorithmic errors, but are not sufficient to test the efficiency of the map and reduce design.

The work presented here contributes to the establishment of systematics testing techniques for MapReduce jobs, through the following proposals:

- fault models that focus on design issues when splitting a task between map and reduce functions
- an automatic search-based technique to generate test data which target these faults
- a series of initial experiments that illustrate the difficulty of detecting these faults, and the capacity of our algorithm at generating data that target them

2.3 Outline

- Chapter 3 introduces the fundamental concepts of the MapReduce framework.

- Chapter 4 introduces two test quality assessment methods, describes methods to data test generation and meta-heuristic search methods. Two data test generators are presented with their results.

- Chapter 5 presents our approach to test data generation for testing MapReduce systems.

- Chapter 6 shows the implementation of the main modules of our solution.

- In chapter 7 we and discuss a case study performed with our solution.

- In chapter 8 we conclude our results.

CHAPTER 3

MAPREDUCE

MapReduce [12] is a programming model and a framework to build large-scale parallel data processing applications. The programming model is inspired on the *Map* and *Reduce* primitives from functional programming languages. The framework proposes two extensions points (or hooks), whose interface is based on this same two higher-order functions Users create MapReduce programs (or jobs) by defining the precise behavior of these functions.

During execution, the framework deploys copies of the program across several worker nodes, partitions the input data and schedules the execution across a set of nodes. The framework also handles node failures and the required communication between nodes. After the deployment, the master selects idle workers to assign a map or a reduce instance and orchestrates their execution. The data flow between the map and the reduce functions is shown in Figure 3.1.

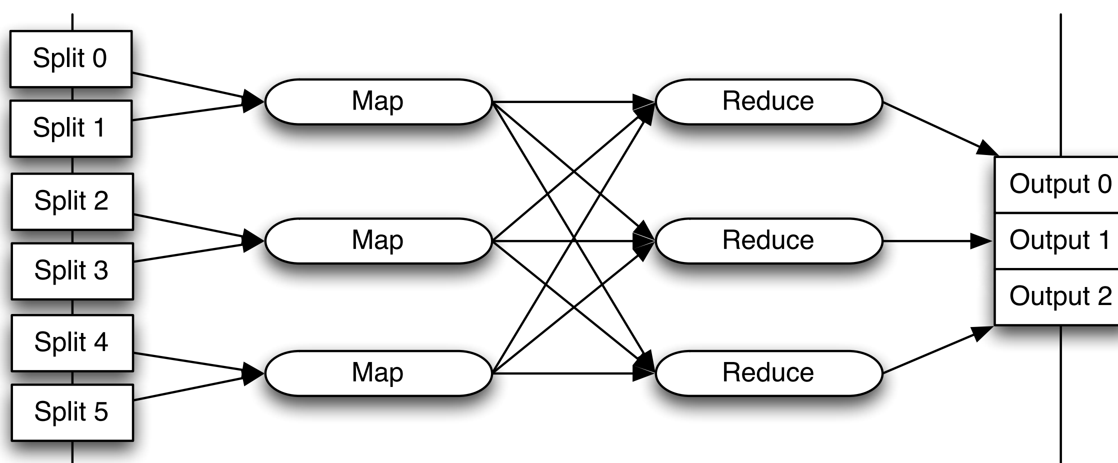


Figure 3.1: Execution of Map and Reduce operations

The input data set is divided into several *splits*. Each split is assigned to a map task and executed on a node. The result of this processing is a set of intermediate keys and

associated values. Reduce tasks take the results from map tasks to produce the final result. When all the reduce instances terminate, they append their result to the final output file.

The whole processing is based on $\langle key, value\rangle$ pairs. The Map function groups together all input values $v1$ associated to the same key $k1$ into an intermediate result set of keys and values $\langle k2, v2\rangle$. These values are passed to the Reduce function that combines them into a reduced set:

map	$k1, v1$	$\rightarrow$	$set(k2, v2)$
reduce	$k2, set(v2)$	$\rightarrow$	$set(v2)$

Eventually, the data flow may be completed with a *Combiner*, a local reduce function used for bandwidth optimization. This function runs after the Mapper and before the Reducer and is run on every node that has run map functions. The Combiner may be seen as a *mini-reduce* function, which operates only on data generated by one machine.

A canonical example of a MapReduce job is the Word Count application, which has as an input several textual documents and as an output a set of pairs $\langle Key, Value\rangle$, where each key is a different word and the value is the number of occurrences of the word on all input documents. The responsibility of of the Mapper is to separate the text into a set of words and that of the Reducer is to aggregate matchings words and count the number of occurrences. In this example, the function of the Combiner is almost identical to that of the Reducer. The main difference is that combiners are executed locally to each node, immediately after each mapper, and use local data, while reducers execute on different nodes and use data that comes from different mappers.

The Java implementation of the map function is presented in Listing 1. The **map()** method has three parameters: **key**, which is never used; **value**, which contains the text to be processed; and **context**, which will receive the output pairs. The body of the method uses the class **StringTokenizer** to break the input text into tokens and then, for each token, writes a pair containing the token and the number 1. The map does not count words, if there are several occurrences of a word in the input, there will also be several occurrences of it in the output.

```java
public static class TokenizerMapper extends Mapper<Object, Text, Text, IntWritable> {

    private final static IntWritable one = new IntWritable(1);
    private Text word = new Text();

    @Override
    public void map(Object key, Text value, Context context)
                   throws IOException, InterruptedException {

        StringTokenizer itr = new StringTokenizer(value.toString());

        while (itr.hasMoreTokens()) {
            word.set(itr.nextToken());
            context.write(word, one);
        }
    }
}
```

Listing 1: Class TokenizerMapper

The implementation of the reduce function is presented in Listing 2. The **reduce()** method has also three parameters: **key**, which contains a single word; **values**, a set containing all values associated to the key (i.e. the word); and **context**, the output. The behavior of the method is quite simple, it sums all values associated to the key and then writes a pair containing the same key and the calculated amount.

```java
public static class IntSumReducer extends Reducer<Text, IntWritable, Text, IntWritable> {

    private IntWritable result = new IntWritable();

    @Override
    public void reduce(Text key, Iterable<IntWritable> values, Context context)
                      throws IOException, InterruptedException {

        int sum = 0;
        for (IntWritable val : values) {
            sum += val.get();
        }
        result.set(sum);
        context.write(key, result);
    }
}
```

Listing 2: Class IntSumReducer

Eventually, the reduce function could also be used as a combiner, to locally aggregate different occurrences of the same word. The choice of using or not a combiner, as well as the number of reducer instances, is not automatic, it is left to developer. Still, this choice may have an important impact in both, the correctness and the efficiency of the job. An example of the inputs and the outputs of both functions when applied to a simple sentence is presented in Table 3.1.

map	"Never Say Never Again"	$\rightarrow$	$\langle Never, 1\rangle$,$\langle Say, 1\rangle$, $\langle Never, 1\rangle$, $\langle Again, 1\rangle$
reduce	$\langle Never, \{1, 1\}\rangle$, $\langle Say, \{1\}\rangle$, $\langle Again, \{1\}\rangle$	$\rightarrow$	$\langle Never, 2\rangle$,$\langle Say, 1\rangle$, $\langle Again, 1\rangle$

Table 3.1: Word count example

CHAPTER 4

TEST DATA GENERATION

4.1 Test Quality Assessment

Two common ways by which a test data set may be assessed are: code coverage [25] and mutation analysis [14].

4.1.1 Code Coverage

Code coverage is a technique to measure how much of an application source code is covered by the test data set. However, it cannot tell the developer whether the covered code works appropriately nor can it objectively assess the quality of the code or the test itself [23]. There are several variations of code coverage technique [10]. The most common are:

- *Method Coverage*: the simplest form of coverage only checks whether a method has been called during the runtime of a test.
- *Statement Coverage*: focuses only on whether a line of source code is executed or not.
- *Condition Coverage*: analyses conditional branches within source code, checking that each branch is executed.
- *Path Coverage*: checks a possible route through the code or a given part of it.
- *Entry Coverage*: deals with any possible call and return of a method during the current execution.

Listing 3 shows an example of a java class and a java test class, respectively. Table 4.1.1 summarizes code coverage analysis results.

```
public class Interval implements Range {

    public boolean includes(int value) {
        return value >= from && value <= to;
    }
}

public class IntervalTest {

    public void testIncludes() throws Exception {
        assertTrue(interval.includes(5));
    }
}
```

Listing 3: Class TokenizerMapper

Table 4.1: Coverage Summary for Class: Interval.

Class	Class,%	Method, %	Line, %
Interval	100%(1/1)	100%(1/1)	100%(3/3)

4.1.2 Mutation Analysis

Mutation testing or mutation analysis is a fault-based testing technique that measures the adequacy of a externally created test data. The adequacy measure is called "mutation adequacy score". This score can be used to measure the effectiveness of a test data in terms of its ability to reveal faults. The general principle underlying mutation testing work is that the faults represent the mistakes that programmers often make and other related testing heuristics. Such faults are deliberately seeded into the original program, by a simple syntactic change, to create a set of faulty programs called mutants, each containing a different syntactic change. To assess the quality of a given test data set, mutants are executed against it to see if the seeded faults are revealed. This causes mutation analysis to run extremely slow and, as a consequence it is not very often used.

4.2 Test Data Generation Techniques

Test data generation which is a component of software testing is a labor intensive task [29]. Test data generation techniques attempt to find input data to satisfy a given testing

requirement. A number of test data generation techniques have been developed and automated. The techniques can be classified into the follow categories [16]:

- *Random* techniques determine test data based on assumptions concerning fault distribution.
- *Path-oriented* techniques generally use control flow information to identify a set of paths to be covered and generate the appropriate test data.
- *Goal-oriented* techniques identify test data covering a selected goal such as a statement or branch, irrespective of the path taken.
- *Intelligent* techniques of automated test data generation rely on complex computations to identify test data.

All these techniques have their weaknesses. Random data test generators may create many test data and yet fail to find those that satisfy test requirement because the requirement is not incorporated into the generation process. Path-oriented test data generator may fail to find data that will traverse a path, since it first identify the path and some paths may be infeasible. An intelligent approach may generate test data quickly, however the analysis may be quite complex and requiring a great insight about many programming situations.

Being a search problem by nature, test data generation may benefit from metaheuristic search techniques since exhaustive enumeration of a data input is infeasible for any reasonably-sized software and random methods are unreliable and unlikely to meet test requirement.

4.3 Metaheuristic Search Techniques

This section gives an overview of some metaheuristic techniques that have been used to test data generation, namely Hill Climbing, Simulated Annealing and Evolutionary Algorithms [24].

4.3.1 Hill Climbing

"Hill Climbing" is a well known local search algorithm. Hill Climbing works to improve one solution, with an initial solution randomly chosen from the search space as a starting point. The neighbourhood[1] of this solution is investigated. If a better solution is found, then this replaces the current solution. The neighbourhood of the new solution is then investigated. If a better solution is found, the current solution is replaced again, and so on, until no improved neighbours can be found for the current solution. This progressional improvement is linked to the climbing of hills in the "landscape" of a maximising objective function. In this landscape, peaks characterise solutions with locally optimal objective values, and troughs signify solutions with the locally poorest objective values. In a "steepest ascent" climbing strategy, all neighbours are evaluated, with the neighbour offering the greatest improvement chosen to replace the current solution. In a "random ascent" strategy (sometimes referred to as "first ascent"), neighbours are examined at random and the first neighbour to offer an improvement is chosen. Hill climbing is simple and gives fast results. However, it is easy for the search to yield sub-optimal results when the hill climbed leads to a solution that is locally optimal, but not globally optimal. In such cases, the search becomes trapped at the peak of a hill, unable to explore other areas of the search space. The search will also become stuck along plateaux in the landscape. In such circumstances, no neighbouring solution is deemed to offer an improvement over the current solution, since they all have the same objective value. Therefore, in non-trivial landscapes, results obtained with hill climbing are highly dependent on the starting solution. A common extension to this algorithm is to incorporate a series of "restarts" involving different initial solutions, to sample more of the search space and minimise this problem as much as possible.

4.3.2 Simulated Annealing

It is desirable to have a search framework that is less dependent on the starting solution. Simulated Annealing is similar in principle to Hill Climbing. However, by allowing for a

[1] Surrounding solutions.

probabilistic acceptance of poorer solutions, Simulated Annealing allows for less restricted movement around the search space. The probability of acceptance p of an inferior solution changes as the search progresses, and is calculated as:

$$p = e^{-\frac{\delta}{t}}$$

Where δ is the difference in objective value between the current solution and the neighbouring inferior solution being considered, and t is a control parameter known as the temperature. The temperature is cooled according to a cooling schedule. Initially the temperature is high, in order to allow free movement around the search space, and so that dependency on the starting solution is lost. As the search progresses, the temperature decreases. However, if cooling is too rapid, not enough of the search space will be explored, and the chances of the search becoming stuck in local optima are increased.

4.3.3 Evolutionary Algorithms

Evolutionary Algorithms use simulated evolution as a search strategy to evolve candidate solutions, using operators inspired by genetics and natural selection. Genetic Algorithms are probably the most well known form of Evolutionary Algorithm. Genetic Algorithms are closely related to Evolution strategies. For Genetic Algorithms, the search is primarily driven by the use of recombination - a mechanism of exchange of information between solutions to “breed” new ones - whereas Evolution Strategies mainly use mutation - a process of randomly modifying solutions.

4.3.3.1 Genetic Algorithm

Genetic algorithm [19] is an optimization that mimics natural processes, such as selection and mutation in natural evolution, to evolve solutions to problems whose solution spaces are impractical to traditional search or optimization techniques.

A genetic algorithm begin with a random population of solutions (called chromosomes) and, through recombination process and mutation operations, gradually evolves the population toward an optimal solution. However, obtaining the optimal solution might not be possible, in this case an approximated solution will be obtained. A selection process

is used to select solutions with best fitness from the current population that will serve as parents to the next generation. An objective function is used to evaluate the fitness of each solution. The three operations of a genetic algorithm are:

- *Selection*: chooses the solutions which are going to participate in crossover, they are chosen according to their fitness value. The choice can be seen as spinning a roulette wheel where each individual has a slot proportional to its fitness value. The wheel is spun as many times as the size of the population, and so a new population is available, which is going to participate in crossover. This new population is made of solutions from the old population, and the number of each type of solution is proportional to its fitness (there are many of the fittest and few of the ones with a low fitness).
- *Crossover*: the members of the population after selection are mated randomly, then every pair is crossed, to create new pairs.
- *Mutation*: modifies one or several genes of a solution (solutions or chromosomes are composed of smaller partes named genes).

4.3.3.2 Generalized Extremal Optimization

Generalized Extremal Optimization algorithm (GEO) [11] is a proposed metaheuristic devised to tackle complex optimization problems. It is a variation of the Extremal Optimization method (EO) [7], which enabled GEO to be applied directly to a broad class of nonlinear constrained optimization problems, with the presence of any combination of continuous, discrete and integer variables [3]. Both EO and GEO are evolutionary algorithms by the simplified evolutionary model of Bak-Sneppen [5].

4.3.3.3 Bacteriologic algorithm

The bacteriologic algorithm [6] is an original adaptation of genetic algorithms inspired by the biological process of the bacteriologic adaptation [30]. The general idea is that

a population of *bacteria* is able to adapt itself to a given environment. If bacteria are spread in a new stable environment they will reproduce themselves so that they fit better and better to the environment. At each generation, the bacteria are slightly altered and, when a new bacterium fits well a particular part of the environment it is memorized. The process stops when the set of bacteria has completely colonized the environment. From this principle, the bacteriologic algorithm takes an initial set of bacteria as an input, and its evolution consists in series of mutations (using a mutation operator) on bacteria, to explore the whole scope of solutions. The final set is build incrementally by adding bacteria that can improve the quality of the set. Along the execution there are thus two sets, the solution set that is being built and the set of potential bacteria, which we call a bacteriologic medium. The global process is incremental and each step is called a generation. A generation consists in four steps:

1. compute the the fitness value for each bacterium in the bacteriologic;
2. select the bacteria that can improve the solution set and add them to the solution;
3. select and mutate bacteria in the medium to generate generates new bacteria;
4. remove bad or useless bacteria from the medium.

The different functions applied at each step are defined in the following. We call $\mathcal{B}$ the set of all possible bacteria for a given problem.

Fitness function. fitness : $2^{\mathcal{B}} \to \mathcal{R}^+$

The fitness function evaluates the quality of set of bacteria regarding the global objective ($2^{\mathcal{B}}$ designates the set of all possible bacteria sets). Along the execution of the algorithm, a bacterium is added to the solution set if it can improve the quality of the set. Thus, the quality of a bacterium at a given moment corresponds to the fitness value the solution set would have if this bacterium was added. Thus, the $relFitness$ function computes the relative fitness of bacterium with respect to the current solution set:

$$relFitness : \mathcal{B} \times 2^{\mathcal{B}} \to \mathcal{R}^+$$
$$relFitness(B, b) = fitness(B \cup \{b\}) - fitness(B)$$

Memorization function. mem : $\mathcal{B} \rightarrow boolean$

This function thus computes the relative fitness of the bacterium. If the fitness satisfies a given condition, the function returns true and the bacterium can be memorized. In the experiments, we use different conditions for memorization. For example, a bacterium can be memorized if its fitness is greater than a given threshold (that it is called *memorization threshold*). Another function memorizes the best bacterium in a bacteriologic medium if its fitness has not been improved for n generations (n being fixed for a particular execution). The goal of this function is to find a trade-off between the convergence speed of the algorithm, and the size of the generated solution set.

Mutation operator. mutate : $\mathcal{B} \rightarrow \mathcal{B}$

The mutation operator generates a new bacterium by slightly altering an ancestor bacterium. This operator is crucial for the algorithm, since it is the one that actually creates new information in the process. We can note that by iterative applications of this operator we should explore the whole set of possible bacteria $\mathcal{B}$.

Filtering function. filter : $2^{\mathcal{B}} \rightarrow 2^{\mathcal{B}}$

Each time a new bacterium is generated it is added in the bacteriologic medium. This set grows along the algorithm execution, and it is necessary to remove bacteria at some time to control the memory space during the execution. On the other hand, useful bacteria should not be removed. The goal of the filtering function is to deal with this trade-off between the growing size of the medium and the usefulness of bacteria. It thus removes bacteria that verify a given condition at some time during execution.

4.4 Test Data Generators

Test data generators may belong to different categories and they can apply different search techniques to search for data within a given search space. Also a test data generator may use either static or dynamic execution during test data generation. For example, Boyer *et al.* [8] used Hill Climbing to search for input data which would execute a given path on a constraint system. Constraint systems are derived from programs through symbolic execution. DeMillo and Offutt proposed a domain reduction technique as part of their

Constraint-based testing work [15]. Although using a constraint system, DeMillo's work differentiate from Boyer's work. The former is a goal-oriented technique while the latter is a path-oriented technique. Miller and Spooner [26] were the first to use dynamic execution and a search technique to generate test data. Their approach is path-oriented. Tracey and co-authors applied Simulated annealing [31] to path-oriented or goal-oriented test data generation.

Recently, evolutionary algorithms have been used to dynamic test data generation for white-box and black-box testing. Baudry *et al.* [6] and Bruno *et al.* [3], both proposed test data generation techniques using evolutionary algorithms other than genetic algorithms. Both, present their results against an implementation with genetic algorithm.

In his work, Baudry uses a bacteriological algorithm to search test data for mutation testing. The algorithm makes use of mutation adequacy score to drive the search. The quality of test data generated by this technique is assessed by mutation analysis as seen on section 4.1.2. Table 4.4 shows a comparison between bacteriological and genetic algorithms results from Baudry's work.

Table 4.2: Comparison between bacteriological and genetic algorithms for a C# parser.

Algorithm	No. generations	Mutation score (%)	Execution Time (Hrs)
Genetic	200	85	26
Bacteriological	30	96	2.5

Bacteriological algorithm offered a huge improvement over the genetic algorithm for mutation testing.

Bruno's work makes use of a GEO algorithm to generate test data to cover specific paths of a system under test. The quality of test data generated is then assessed by the the level of source code covered by the test data, see section 4.1.1. Table 4.4 shows results between GEO an genetic algorithm on seven Java subject programs (SPs): simplified triangle, remainder, product, linear search, binary search, middle value and triangle.

The GEO algorithm didn't offer much value in terms of performance. It's main benefit comes from the fact that it is easier to setup and tune than genetic algorithms.

Table 4.3: Comparison between GEO and genetic algorithms.

SP	GA Coverage (%)	GEO Coverage (%)	GA SP executions	GEO SP executions
1	**92.41**	89.19	253640	268171
2	100	100	**109**	651
3	100	100	1853	2277
4	97.76	**98.49**	**45343**	51721
5	**99.83**	99.53	**48286**	51381
6	100	100	**100**	550
7	66.62	**66.67**	799764	**799300**

4.5 Generating Test Data For MapReduce Jobs

There are some `MapReduce` test tools [1, 2, 18] available, but none of them address the test data generation problem. Even thought one could combine these tools with an external test data generator, only unit test data could be generated. In this case, testing would not be integrated with the framework. To date none of available test data generators offer such integration, this is an open issue.

CHAPTER 5

SOLUTION

This chapter presents our approach for the generation of test data for testing MapReduce jobs. Before presenting the general approach and the algorithms used, we introduce new specific mutation operators, used to evaluate the quality of the generated test data.

5.1 MapReduce specific mutation operators

As we stated in the introduction, MapReduce jobs differ from traditional applications in the sense that they only describe the functional part of a program. Other aspects of the program, such as security, and persistence, are left to the framework. Therefore, most MapReduce jobs are small and concise pieces of software, which manipulates high-level data structures. When executed, even with a few input data, the code of the job is totally covered (all paths coverage). The consequence for mutation analysis is straightforward: simple test data can detect any syntactic mutation (i.e. mutated code is always executed), making mutation analysis ineffective.

We strongly believe that most tricky implementation errors are due to two main causes: (i) a wrong adaptation of the algorithm to the MapReduce paradigm or (ii) a wrong configuration of the job, such as an inaccurate number of reducers or the absence of a combiner. While these errors may result in functional errors, in most cases this is not the case. Nevertheless, they have an impact on the resource usage that only arises when the job is executed on several nodes, using an important amount of data.

To check if test suites can expose these errors, we propose new mutation operators that are specific to MapReduce. We summarize these operators in Table 5.1. Since they do not introduce syntactic changes, the operators are close to semantic mutation operators introduced by Clark et al. [9], i.e. they represent a possible misunderstanding of the description language, or in the precise case, of the MapReduce paradigm.

Operator	Description
Insert combiner	Forces the use of the reduce function as a combiner to the MapReduce job.
Remove combiner	Removes the combiner from the job.
Alter the number of reducers	Forces the use of different number of reducers of the job.

Table 5.1: Proposed Operators

5.2 Test data generation

The global algorithm for generating and improving test data is described by Algorithm 2. Before presenting the algorithm, some previous definitions are necessary. Let us denote by $\mathcal{D}$ the set of all possible input data for a given test case, by $\mathcal{K}$ a set o pairs $\langle d, m \rangle$, where d is a test data and m is a mutant killed by this data and by $data^{\mathcal{K}}$ denote a set of all test data included in $\mathcal{K}$.

Algorithm 2: Global Algorithm

Input : $JUT \in Program$
Input : $TC \in Program$
Input : $initial^{\mathcal{D}} \subset \mathcal{D}$
Output: $data^{\mathcal{K}} \subset \mathcal{D}$
Data: $expected^{\mathcal{R}}, actual^{\mathcal{R}} \subset 2^{Result}$
Data: $living^{\mathcal{M}} \subset 2^{Program}$
$living^{\mathcal{M}} \leftarrow GenerateMutants(JUT)$;
$data \leftarrow initial^{\mathcal{D}}$;
$newdata \leftarrow data$;
repeat
  foreach $d \in newdata$ **do**
    $expected^{\mathcal{R}} \leftarrow expected^{\mathcal{R}} + execute(JUT, TC, d)$;
  foreach $m \in living^{\mathcal{M}}, d \in newdata$ **do**
    $actual^{\mathcal{R}} \leftarrow execute(JUT, m, d)$;
  $killed^{\mathcal{M}} \leftarrow MutationAnalysis(expected^{\mathcal{R}}, actual^{\mathcal{R}})$;
  $living^{\mathcal{M}} \leftarrow living^{\mathcal{M}} - killed^{\mathcal{M}}$;
  if $living^{\mathcal{M}} \neq \emptyset$ **then**
    $newdata \leftarrow generate(\mathcal{K})$
  else
    exit loop
until *fitness-reached* $\vee$ *enough-iterations*;
return $data^{\mathcal{K}}$

Initially, we have two programs, a MapReduce job (JUT) and a test case (TC), as

well as a initial set of test data ($initial^{\mathcal{D}}$) In the first step, we generate all mutants for the JUT and assign this set to the variable $living^{\mathcal{M}}$, which stores all the remaining mutants (i.e. that were not killed by any test sequence). Mutant generation is only done once during all process. The initialization continues by assigning the initial test data to the $improved^{\mathcal{D}}$ variable, the set of test data which is improved in the algorithm.

After the initialization, we start a loop until the desired fitness or a predefined number of iterations is reached. Inside the loop, the first step is to calculate the results for the new test data, using the **execute()** function. This function returns a tuple Result=$\langle Program, TestData, TestResult, Delay \rangle$, containing the result and the delay for each test case execution. The variable $expected^{\mathcal{R}}$ keeps a set of Results. This set is used as the reference for test case execution. The second step of the loop calculates the results for all new test data and all remaining mutants. It takes a product of all remaining mutants and all new test data and calculates results for all pairs. The result is assigned to the variable $actual^{\mathcal{R}}$, which is also a set of the Result tuple.

Once all – expected and actual – results are calculated, we can proceed with mutation analysis functions, presented by Algorithm 3. This function takes all calculated results, expected and actual, and returns all mutants killed by the test data. A mutant is considered killed if, for a given test data, it produces a different result or the same result but in a non-equivalent delay. The mutation analysis function also updates $\mathcal{K}$, keeping track of the mutants killed by each test data (i.e. bacterium). Test data that do not kill any mutant are discarded.

After the mutation analysis, all killed mutants are subtracted from $living^{\mathcal{M}}$ If live mutants still remains, test data must be improved. The bacteriological algorithm uses $\mathcal{K}$ as an input, to select the best bacteria (i.e. test data) from the population. The best bacteria are mutated, i.e. new test data is generated, and the loop continues. Once the loop exit condition is reached, $data^{\mathcal{K}}$ contains the desired test data.

Algorithm 3: Mutation Analysis

Input : $expected^{\mathcal{R}}, actual^{\mathcal{R}}$
Output: Set of Program
$killed \leftarrow \{\}$;
foreach $e \in expected^{\mathcal{R}}$ **do**
  foreach $a \in actual^{\mathcal{R}}$, **do**
    if $TestResult^e \neq TestResult^a \vee Delay^e \not\approx Delay^a$ **then**
      $\mathcal{K} \leftarrow \mathcal{K} + \langle TestResult^a, Program^a \rangle$;
      $killed \leftarrow mutants + Program^a$;
return *mutants*

5.3 Using MapReduce to generate test data

When generating test data for MapReduce jobs, a tempting approach is to use the MapReduce framework to accelerate the whole process. Indeed, mutant generation and evolutionary algorithms are good candidates for parallelization. For instance, during the mutant generation, we could identify all possible mutation for a given program and then use the framework to execute, in parallel, two steps: (i) apply the mutation operators and (ii) execute test cases, mappings each combination of a bacteria and a mutant to a map task.

While this approach could attenuate the main drawback of mutation analysis, the large amount of time it consumes, it is not adapted to the present case for two reasons. First, due to the few number of mutant operators and the size of the job source code, a local generation is more efficient. Second, test cases need a cluster to be executed, they cannot be executed on a single node.

Concerning the bacteriological algorithm, the new bacteria operation could be parallelized. And in this case, it would be perfectly adapted to the present case. Nevertheless, since this kind of optimization is not our main focus, we chose to generate bacteria locally.

CHAPTER 6

IMPLEMENTATION

In this chapter, we present implementation details of our solution presented in the previous chapter. The solution was developed in Scala [28] (version 2.9.1).

In the section 6.1 we describe the main components that form the core of our solution and their interactions. Section 6.2 describe the interface users use to implement their mutation operators.

6.1 Main Components

In this section, we describe the main components of our solution: the execution module which executes a test data against a MapReduce job and the evolution module which evolves the test data.

Figure 6.1 shows the three main classes of the execution module. The most important operations provided by this module are:

- **calculate():** tells the master to evaluate a list of test data against a list of mutant jobs.

- **instantiate():** loads a job into the ClassLoader.

- **run():** executes a test data against a job.

Figure 6.2 shows the main classes of mutant management module. The MutantSet class performs the mutation score calculation.

The execution module implements the evaluation step of bacteriologic algorithm. The three remaining bacteriologic steps are implement by the bacteriologic module, figure 6.3. The main operations are:

- **evolve():** takes a list of bacteria and applies a mutator operator to some bacteria.

Master	
Master(int, int, List<Object>, List<Mutant>, CountDownLatch)	void
nrOfResults()	int
start()	long
workers()	List<Object>
router()	Object
receive()	Object
preStart()	void
postStop()	void

Worker	
Worker()	void
run(Bacterium, Mutant)	int
receive()	Object
instantiate(Mutant)	Object

Runner	
Runner()	void
calculate(int, int, List<Mutant>, List<Object>)	void

Figure 6.1: Execution module

MutantSet	
MutantSet(double, String, String)	void
set()	Object
dead()	List<Mutant>
receive()	Object
mutationScore()	double
mutants()	Object
toString()	String

Remaining	
Remaining(List<Mutant>)	void
apply(List<Mutant>)	Remaining
unapply(Remaining)	Option<List<Mutant>>

Mutant	
Mutant(int, String, String)	void
apply(int, String, String)	Mutant
unapply(Mutant)	Option<Tuple3<int,String,String>>

Figure 6.2: Mutant management module

- **memorize():** memorizes a bacterium to the solution set.
- **purge():** removes unfit bacteria from the medium.

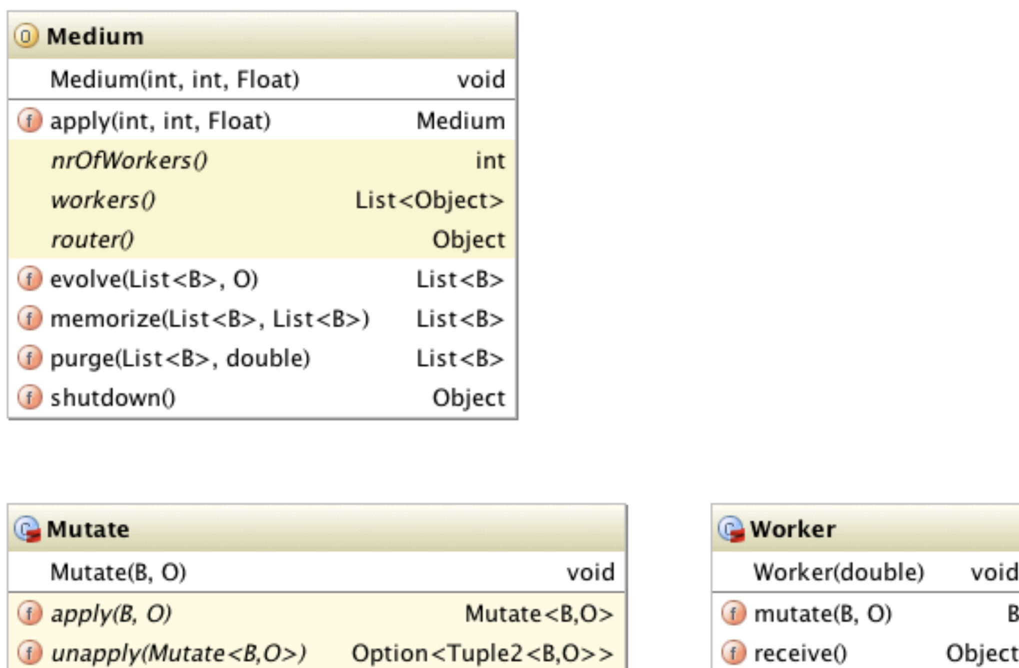


Figure 6.3: Bacteriologic module

A worker is responsible for applying a selected mutation operator to the bacterium. These three modules, together, form the the test data generator core.

6.2 User Interface

Given the functionality provided by the generator core modules, users have to implement mutation operators according to their data types. The mutation operators provided are them used to evolve the test data during the evolution step of bacteriologic algorithm.

Figure 6.4 shows the Mutation interface and the mutate operation, the only operation users have to implement. The mutation operation takes an object of type B and must return a new object of type B.

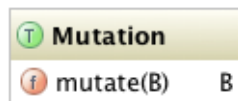


Figure 6.4: Mutation interface

CHAPTER 7

EXPERIMENTS

This chapter presents the results of applying our test data generation approach to two MapReduce jobs: WordCount and Breadth-First Search (BFS). WordCount, previously introduced in chapter 3, is a very simple job, which is bundled with Apache Hadoop. The input data of this job is a text document. Breadth-First Search is a graph search algorithm that finds the shortest paths from a source node to all other nodes in the graph. Its input data is a graph.

The experiments were executed on a cluster with 7 nodes running Debian GNU/Linux. The cluster nodes are interconnected by a 1 Gbps network and have a similar configuration: 2 Intel Core2 dual-core processors E8400@3GHz, 4 GB RAM memory and 500 GB SATA HD. The code of the experiments and their data are available on a public repository[1]. All the experiments used Java version "1.6.0_20" and Hadoop MapReduce Release 0.20.2.

The experiments show that the bacteriological algorithm is able to evolve the test data for two different jobs. The improved test data was able to kill all the generated mutants in less than ten iterations. Additionally, the experiments show that the proposed MapReduce specific mutants are more difficult to kill than traditional mutants, requiring the use of more specific data, in larger quantities.

The chapter is organized as follows. We first explain how the bacteriological algorithm presented in Section 4.3.3.3 was adapted to the input data used in the MapReduce jobs. We then present an evaluation of the use traditional (syntactic) mutants for mutation analysis of 4 MapReduce jobs, showing their poor adaptation for this kind of program. Finally, we present the application of our approach to the two MapReduce jobs.

[1] `https://github.com/antonioj-mattos`

Text	
Grow	Doubles the size of the text, duplicating the content.
Shrink	Reduces by 50% the size of the text.
Shuffle	Mixes the contents (words).
Disqualify	Exclude the less used words.
Graph	
Reduce	Removes 20% of the edges.
Raise	Adds new edges to the graph.

Table 7.1: Bacteriological Mutants

7.1 Applying the bacteriological algorithm

In order to apply the bacteriological algorithm for these experiments, we implemented two sets of mutation operators, one for each type of data. The first set of operators applies to text files and the second to graphs. Table 7.1 summarizes the mutant operators.

7.2 Traditional mutants

The first experience presented here concerns the use of traditional mutation operators [22] to perform mutation analysis on MapReduce jobs. Besides WordCount and BFS, we used two other jobs, SecondarySort, also bundled with Hadoop and PageRank, available from the Haloop project[2]. The following mutation operators were used: arithmetic operation substitution and inversion of boolean values.

Job	Mutants	Killed	Score	Test Data (kB)
WordCount	4	4	100%	0.5
BFS	21	21	100%	0.5
PageRank	33	33	100%	479
SecondarySort	25	25	100%	0.9

Table 7.2: Mutation Analysis with traditional mutation operators

Table 7.2 summarizes the results of the mutation analysis. All tests use a single input data file. The input text for WordCount has 55 words. The input graph for BFS has 22 vertices. The input file for PageRank contains 6,012 URLs. The input file for SecondarySort contains pairs with all digits from -9 to 9.

[2] http://code.google.com/p/haloop/

By reason of the particular characteristics of MapReduce jobs (i.e. concise and high-level source code), few mutants were created and single and small test data were able to kill all mutants. We believe that this would be also the case for several other jobs, making the use of traditional mutants ineffective in most cases. Obviously, the implementation of the map and the reduce functions is not limited and can be very complex, in which case traditional operators could be as effective as for any other program. However, this was not the case for the MapReduce jobs found on different projects, such as Cloud9[3], Mahout[4], Hbase[5] and Hive[6].

7.3 WordCount

The first MapReduce job implementation we choose to apply our approach is WordCount. This choice is motivated by its simplicity in terms of source-code and input data (text files). Furthermore, it is fairly simple to predict the impact of the MapReduce-specific mutation operators on the WordCount execution. The impact of the combiner on this job depends heavily on the input data. The more a split contains equivalent words, the more the combiner is useful. Additionally, the size of the input also impacts on the execution: if not enough data is provided, a single split is created and the execution is performed by a single mapper and a single reducer. In this case, the mutant job behaves the same as the original job and cannot be detected.

This information concerning the behavior of the generate mutants is very helpful. If we can forecast which kind of data is more likely to expose the error of each mutant, we can carry on a preliminary study that measures the impact of the best test data for each mutant, in terms of resource usage. If the difference of resource usage between a mutant and the original job is significant enough, we are sure that the mutant can be detected.

Table 7.3 summarizes the mutants created for this job. The original jobs uses a combiner and a single reducer. Mutant 1 does not use a combiner and mutants 2 and 3

[3]http://lintool.github.com/Cloud9/
[4]http://mahout.apache.org/
[5]http://hbase.apache.org/
[6]http://hive.apache.org/

$\mathcal{M}1$	Remove Combiner
$\mathcal{M}2$	Use $\frac{n}{2}$ Reducers
$\mathcal{M}3$ Use	n Reducers

Table 7.3: WordCount Mutants

File	$\mathcal{M}0$ **delay (ms)**	$\mathcal{M}1$ **delay (ms)**	**Difference**
1	49,518	227,654	78.2%
2	215,687	278,738	22.6%
3	284,285	283,083	100.63%

Table 7.4: Input Data Impact Evaluation

uses a different number of combiners, 3 and 6, respectively. The number of combiners is calculated based on the number of available nodes, denoted by n.

Before applying our approach, we evaluated the impact of different input data on the original program ($\mathcal{M}0$) and on $\mathcal{M}1$ During the evaluation, we used three text files, of 768 MB each, which were automatically split in 12 parts of 64MB. The first file contains the same string all over the file, the second one contains the same string at each split and the third one contains different strings.

Table 7.4 presents the results of this first evaluation. Results shows that producing the same string all over the file (i.e. file 1) is the best data, since there is a large difference of performance between $\mathcal{M}0$ and $\mathcal{M}1$. In this case, the combiner counts the number of entries at each node and outputs only one string per node. The network traffic is very low and the reduce phase computes only 12 entries. This is not the case for $\mathcal{M}1$, since word entry is transferred to the reducers where the count computation is done. As result, the computation delay is 78.2% lower.

The worst data is found in the third file. In this case, it is hard to differentiate $\mathcal{M}0$ from $\mathcal{M}1$. In fact, the usage of the combiner tends to be only an overhead, since the count of the words will be one to all the strings. For the second file, $\mathcal{M}1$ is 22.6% lower than $\mathcal{M}0$. This is due to the computation of the outputs from the Maps that are at the same node. It is noticed if one uses only one Map per node, then the result would be similar to the third file. The same words are found only at different nodes and the Reduce

computes the same amount of data.

This evaluation helps us in many ways. It shows that test data have a meaningful impact on the performance or, in other words, that mutants can be detected and that our approach is feasible. It also help us to improve the operators presented in Section 7.1, showing that test data should evolve not only its contents, but also its size.

After performing this initial evaluation, we apply our approach to the WordCount job. First, we define a fitness function to calculate the quality of bacteria, based on the difference on the execution delay between $\mathcal{M}0$ and a mutant. The more the difference is important for a given mutant, the greater is the fitness of the test data. We consider that a mutant is killed if the difference is greater than 10%. During the iterations, test data (i.e. the bacteria) is stored in the Hadoop Distributed File System (HDFS). The size of the set of test data is not limited.

Figure 7.1 shows the evolution of the execution time of $\mathcal{M}0$ and $\mathcal{M}1$ through the iterations. As the test data evolves, the difference between the two delays increases. The iteration starts with a single text file of 64 MB, containing at most a single occurrence of each word (i.e. the worst case for mutant $\mathcal{M}1$)

Figure 7.2 shows the evolution of the test data set through the iterations. As expected, the data set grows over time. Indeed, bigger data sets are needed to expose a bad choice of the number of reducers, i.e., to kill mutants $\mathcal{M}2$ and $\mathcal{M}3$.

The approach is also applied to mutants $\mathcal{M}2$ and $\mathcal{M}3$. For $\mathcal{M}2$ and $\mathcal{M}3$, we needed to improve the test data during 8 iterations to kill both mutants, i.e. reach a difference greater than 10%. The final test data is a single file of 128MB. For this test data, the execution delay of $\mathcal{M}0$ and $\mathcal{M}2$ and $\mathcal{M}3$ is 68,543 ms, 80,525 ms and 80,512 ms, respectively.

7.4 Breadth-First Search

The Breadth-First Search implementation is more complex than the WordCount in terms of code and input data (graphs). However, it is also simple to predict the impact of the mutation operators on its execution. Intuitively, the worst and best cases to identify a

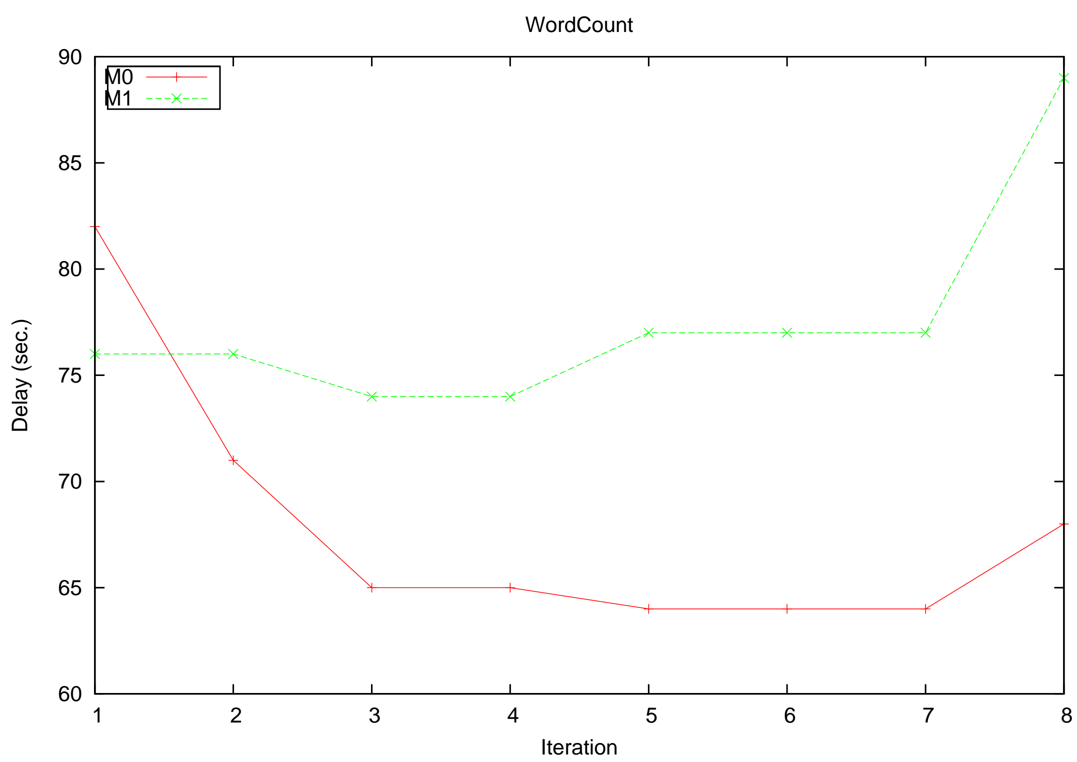


Figure 7.1: WordCount

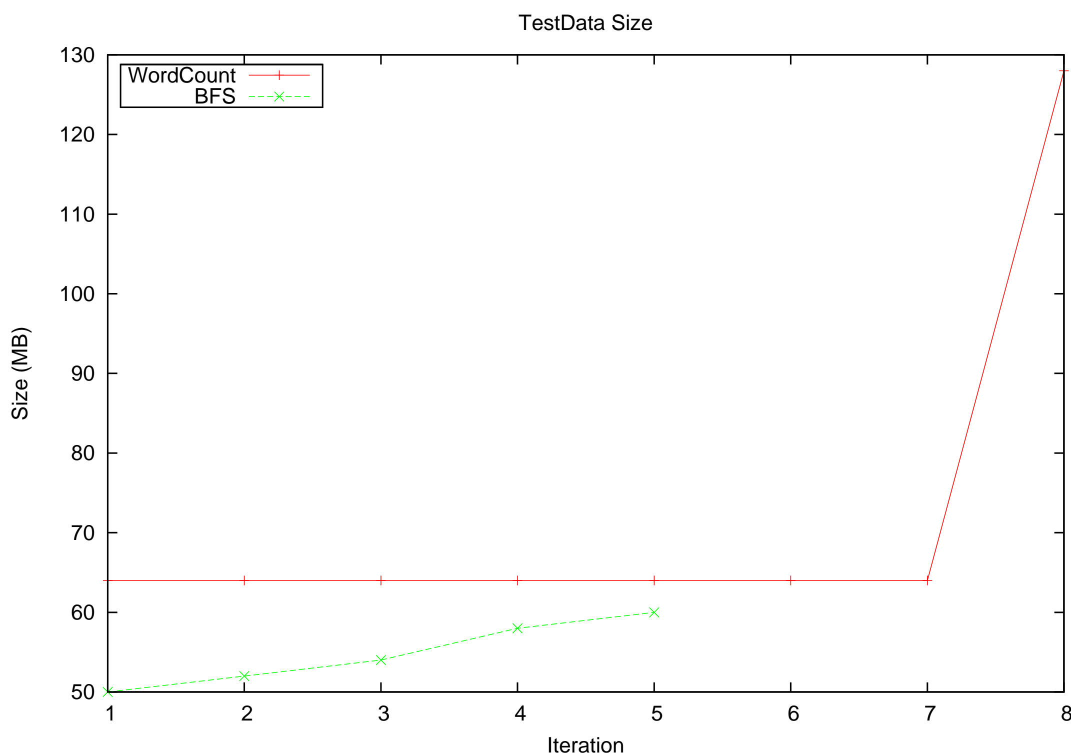


Figure 7.2: Test Data Size

mutant are related to the density of a graph, as the following results will prove. In graphs with low density, the search tends to be serialized and the MapReduce distribution would be worthless.

The Breadth-First Search behaves as follows. It starts at a given vertex x, then, in a first step it enqueues all the vertices at distance one edge away from x. In the second step, it dequeues a vertex y and examine it. At the same time, the vertices one edge away from y are enqueued as well. This process goes on until the sought element is found or the queue is empty. In the MapReduce implementation, the Breadth-First Search uses counters that are incremented each time a shorter distance is encountered in the reducer (see [21] for a complete discussion on the MapReduce BFS implementation). At the end of each MapReduce iteration, the driver program reads the counter value and determines if another iteration is necessary.

$\mathcal{M}1$	Remove Combiner
$\mathcal{M}2$	$\frac{n}{2}$ reducers
$\mathcal{M}3$	n reducers

Table 7.5: Breadth-First Search Mutants

Table 7.5 summarizes the mutants created for this job. The mutant $\mathcal{M}1$ is similar to the WordCount's and aims at removing the combiner from the MapReduce workflow. The mutants $\mathcal{M}2$ and $\mathcal{M}3$ change the number of reducers of the execution to $\frac{n}{2}$ and n respectively, where n is the number of the Hadoop nodes.

Before applying our approach, we also performed a preliminary evaluation of the test data. Two types of graphs were tested: complete and sparse. The complete graph has two versions related to its size: 100kB and 768MB. The sparse only the small version with 100kB. This experiment aims at identify the effects of a mutation on the density of each graph.

Concerning the bacteriological algorithm, the fitness function is the time difference between the execution of a mutant and the original job $\mathcal{M}0$. No filter was used, since the number of bacteria was small and all the bacteria were memorized in the HDFS to be used along the iterations.

As expected, the results show that complete graphs are more efficient test data than sparse graphs (see Table 7.6). In sparse graphs each vertex is linked to only one other vertex. It turns out to be the worst case for the BFS doing as many interactions as the number of vertices minus one. In this case, the system becomes a ring and the computation is serialized with only one instance of Map and Reduce to execute the BFS. The combiner only forwards the Map output to the Reduce, since it receives the result that have the same vertex destination (only one in the worst case). In this case the combiner has no computation and becomes only an overhead of 3% of the time. This preliminary evaluation was important to establish the time baseline for the worst and best cases used by the fitness function. It was equally important to evaluate the quality of both test data (graphs) for each job mutant.

After the evaluation, we applied our approach to mutants $\mathcal{M}1$, $\mathcal{M}2$ and $\mathcal{M}3$. We

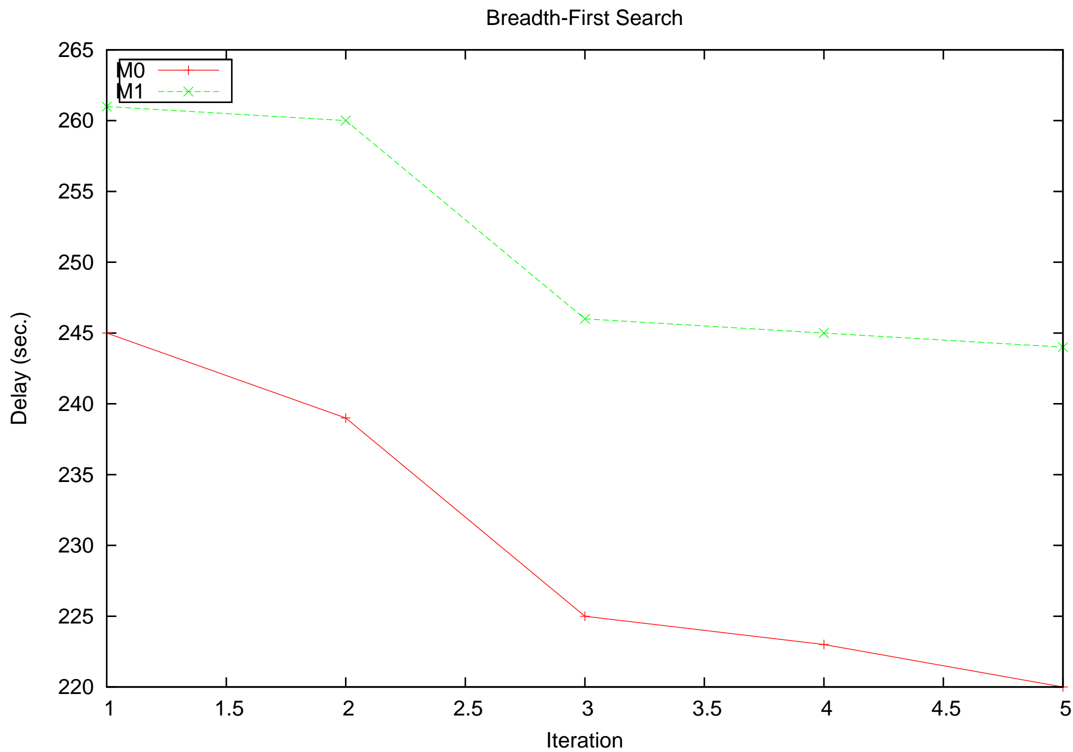


Figure 7.3: Breadth-First Search

needed 5 iterations to generate a test data that was able to kill the mutant $\mathcal{M}1$. Indeed, a greater performance difference was detected (and the mutant was killed), when a more complete graph was generated, reducing the execution delay. The other mutants, $\mathcal{M}2$ and $\mathcal{M}3$, were killed at the first iteration. Indeed, when using a different number of reducers, the job produced a wrong result.

Graph	$\mathcal{M}0$ **delay (ms)**	$\mathcal{M}1$ **delay (ms)**	**Difference**
Complete	653,041	1344,257	51.4%
Sparse	3419,107	3319,438	3%

Table 7.6: BFS results

7.5 Discussion

In the experiments, we used the performance of a job to evaluate the usage of resources. While this is the most facile solution, it may be influenced by the bad performance of a single mapper or reducer node, which delays the execution of the job. We believe that an analysis of the partial delays can be helpful to improve the test data. However, since this information is not available globally, we would need to monitor each node of the cluster to gather it.

During the execution of our approach, only a few mutants are generated for each job. However, this is rather a positive point. Indeed, if the explosive number of mutants is already an important issue when testing standalone software, the impact on distributed software in even greater, since having several executions has prohibitive cost.

The experiments also showed as that the size of the test data is as important as its contents. Indeed, an important amount of data may be necessary to expose some design errors, such as an incorrect number of reducers.

CHAPTER 8

CONCLUSION

The MapReduce framework is very effective at separating all concerns related to distribution, data consistency and fault tolerance from the application specific processing algorithms. The major benefit of this clean and efficient separation is that it is fairly easy to write MapReduce jobs that can process huge quantity of data on cluster of computers.

We highlight a surprising consequence of the advanced separation of concerns of MapReduce for testing: the challenge for testing is more in assessing the efficiency of resource usage and the validity of design decisions on the global performance, rather than looking for errors in the application jobs. We have empirically demonstrated this effect through two experiments. First, we have demonstrated that very simple test data can detect all arithmetic and logical errors that can be injected in Map and Reduce algorithms. Second, we have injected errors in the design of the Map and reduce functions. In that case, we were able to reveal important performance differences through testing, but it was necessary to generate larger and more complex data.

The experiments also showed that our bacteriological algorithm is able to generate test data that is efficient enough to expose bad design decision. We believe that, while test data generation based on paths coverage can be useful for detecting functional errors, they do not seem adapted to expose errors that are related to a bad understanding of the framework. Indeed, a bad decision when choosing how the input data is split, or which function between Map and Reduce should implement a given behavior, may double the overall performance of the job.

We are currently working on other MapReduce-specific mutation operators, beyond the operators presented here. Our goal is to propose operators that are able to modify the way data is split and change the order data is sent to mappers and reducers. While the order of the input data is not ensured by the framework, the order tends to be the

same during the execution of test cases.

BIBLIOGRAPHY

[1] Herriot, large-scale automated test framework. http://wiki.apache.org/hadoop/, 2010.

[2] Mrunit project. http://archive.cloudera.com/docs/mrunit/index.html, 2010.

[3] Bruno T. De Abreu, Eliane Martins, and Fabiano L. De Sousa. Generalized extremal optimization: a competitive algorithm for test data generation. 2008.

[4] Inc. Aster Data Systems. In-database mapreduce for rich analytics.

[5] P. Bak and K. Sneppen. Punctuated equilibrium and criticality in a simple model of evolution. *Physical review letters*, 71(24):4083–4086, December 1993.

[6] Benoit Baudry, Franck Fleurey, Jean-Marc Jézéquel, and Yves Le Traon. From genetic to bacteriological algorithms for mutation-based testing: Research articles. *Softw. Test. Verif. Reliab.*, 15:73–96, June 2005.

[7] Stefan Boettcher and Allon G. Percus. Optimization with extremal dynamics. *Physical Review Letters*, 86:5211–5214, 2001.

[8] Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. Select–a formal system for testing and debugging programs by symbolic execution. In *Proceedings of the international conference on Reliable software*, pages 234–245, New York, NY, USA, 1975. ACM.

[9] John A. Clark, Haitao Dan, and Robert M. Hierons. Semantic mutation testing. In *ICST Workshops*, pages 100–109. IEEE Computer Society, 2010.

[10] Steve Cornett. *Code Coverage Analysis*. http://www.bullseye.com/coverage.html.

[11] Fabiano Luis DE SOUSA, Fernando Manuel RAMOS, Pedro PAGIONE, and Roberto M. GIRARDI. New stochastic algorithm for design optimization. *AIAA journal*, 41(9):1808–1818.

[12] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, 2008.

[13] Jeffrey Dean, Sanjay Ghemawat, and Google Inc. Mapreduce: simplified data processing on large clusters. In *In OSDI'04: Proceedings of the 6th conference on Symposium on Operating Systems Design & Implementation*. USENIX Association, 2004.

[14] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11:34–41, April 1978.

[15] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17:900–910, September 1991.

[16] David E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[17] Greenplum. A unified engine for rdbms and mapreduce, 2008.

[18] Haryadi S. Gunawi, Thanh Do, Pallavi Joshi, Joseph M. Hellerstein, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, and Koushik Sen. Towards automatically checking thousands of failures with micro-specifications, Jun 2010.

[19] J. Holland. *Adaptation in natural and artificial systems*. University of Michigan Press, 1975.

[20] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In Paulo Ferreira, Thomas R. Gross, and Luís Veiga, editors, *EuroSys*, pages 59–72. ACM, 2007.

[21] Jimmy Lin and Chris Dyer. *Data-Intensive Text Processing with MapReduce*. Synthesis Lectures on Human Language Technologies. Morgan & Claypool Publishers, 2010.

[22] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: an automated class mutation system. *Softw. Test., Verif. Reliab.*, 15(2):97–133, 2005.

[23] Brian Marick. How to Misuse Code Coverage, 1999.

[24] Phil McMinn. Search-based software test data generation: a survey: Research articles. *Softw. Test. Verif. Reliab.*, 14:105–156, June 2004.

[25] Joan C. Miller and Clifford J. Maloney. Systematic mistake analysis of digital computer programs. *Commun. ACM*, 6:58–63, February 1963.

[26] W. Miller and D. L. Spooner. Automatic generation of floating-point test data. *IEEE Trans. Softw. Eng.*, 2:223–226, May 1976.

[27] Prashanth Mundkur, Ville Tuulos, and Jared Flatow. Disco: a computing platform for large-scale data analytics. In *Proceedings of the 10th ACM SIGPLAN workshop on Erlang*, Erlang '11, pages 84–89, New York, NY, USA, 2011. ACM.

[28] Martin Odersky and al. An overview of the scala programming language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[29] Roy P. Pargas, Mary Jean Harrold, and Robert R. Peck. Test-data generation using genetic algorithms. *Software Testing, Verification And Reliability*, 9:263–282, 1999.

[30] Michael L. Rosenzweig. *Species Diversity in Space and Time*. Cambridge University Press, May 1995.

[31] N. Tracey, J. Clark, K. Mander, and J. McDermid. An automated framework for structural test-data generation. In *Proceedings of the 13th IEEE international conference on Automated software engineering*, ASE '98, pages 285–, Washington, DC, USA, 1998. IEEE Computer Society.

[32] Tom White. *Hadoop: The Definitive Guide*. O'Reilly Media, 1 edition, June 2009.