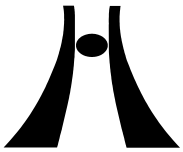
Code:_____



**HÖGSKOLAN**
**I GÄVLE**

**Faculty of Engineering and Sustainable Development**

# Representation of utility difference proportions: matrices and consistency checking

*Yiming Gong*
*Xichen Jiang*
*June 2012*

Bachelor Thesis, 15 credits, C
Computer Science

**International Bachelor Program of Computer Science**
**Examiner: Ulla Ahonen-Jonnarth**
**Supervisor: Magnus Hjelmblom**

# Representation of utility difference proportions: matrices and consistency checking

by

Yiming Gong
Xichen Jiang

Faculty of Engineering and Sustainable Development
University of Gävle

S-801 76 Gävle, Sweden

Email:
Ofk09ygg@student.hig.se
Ofk09xJn@student.hig.se

**Abstract**

When making a good decision, people must evaluate the advantages and disadvantages of each option, and take all the alternatives into consideration. But it's always difficult for most of us. How to make a good decision and make a smart choice becomes more and more important in our daily life. In our thesis work, we have performed a pilot study within a research program in decision analysis regarding representation of utility difference proportion matrices, with the problem of utility aggregation in view. We have proposed a matrix representation based on a map of maps and implemented it as a Java class. We have also defined a simple measure of inconsistency between utility difference proportions, and implemented this measure as a method in the matrix class. The function of the matrix class has been verified with some simple test programs.

# Contents

# 1 Introduction

## 1.1 Background

We all have times when we have to make complex decisions in which several criteria are involved. The technology of multi-criteria decision-analysis (MCDA) has nowadays been applied to many fields. There are a lot of methods for MCDA such as Analytic Hierarchy Process (AHP), Analytic network process (ANP), Multi-attribute utility theory (MAUT) and so on. Among them, AHP is the most widely used, since it is regarded as having greater flexibility and effectiveness than other method. The proponents of the AHP method promise the user to get the final solution for a particular problem with accurate and acceptable result.

In the complex society of today, decision problems involving a single criterion have become increasingly unusual. Therefore, multiple-criteria techniques have emerged as an important study field within decision theory. See [1] and [2]. "Multi-criteria decision-making or multiple-criteria decision analysis is a sub-discipline of operation research that explicitly considers multiple criteria in decision-making environments."[3]

Let's look at an example. When we intend to buy a laptop, price, performance, weight, quality and appearance may come to be the main criteria to be considered. It is unusual to have the lowest price with best quality. In that situation, when price is in conflict with quality, we need to think of them deliberately and select the consequence that we value the most. [3]

To develop the example further: let us assume our laptop has been used over five years, and one day it is not functioning properly. We decide to buy a new one to replace it. When we step into the PC mall, three brands impress us the most. They are A, B and C. All three of them look fabulous. It is a hard choice for us to choose one from another. In this tricky situation, we would like to introduce Multi-Criteria Decision-Analysis as an efficient way to help us pick the best laptop. For many customers, price is often his or her first concern; performance can certainly affect users' purchase motivation. Compared to desktop computers, the weight (size) of laptop is also a particular issue that is important to consider. As a result, price, performance and weight (size) come to be our criteria. By evaluating each brand using these criteria, we will probably get a satisfactory choice.

The purpose of multi-criteria decision-analysis is to support us when facing problems that involve multiple criteria. It helps us choose the most preferred alternative from a set of available alternatives when many things matter.

## 1.2 Aim of Research

The research group in decision, risk and policy analysis at the University of Gävle has recently started a research program in decision analysis. The aim of this research program is to explore the theoretical foundations of an analytical decision support tool based on additivity and proportionality with the problem of utility aggregation in view. See [4] and [5]. Desired features of this tool are, for example, the ability to interactively add consequences and aspects, and establish proportions between utility differences while the program checks the consistency of these proportions. This thesis work is a small pilot study which aims to contribute to the foundation for this analytical decision support tool. The pilot study concentrates on the basic representation of utility difference proportion matrices and the problem of determining whether specific utility difference proportions are consistent or not.

# 2 Scientific Background

## 2.1 Analytic Hierarchy Process

Analytic Hierarchy Process (AHP) was introduced by Tomas L. Saaty in the 1970s. It is a systematic method for comparing a list of objectives or alternatives. [6] Due to its flexibility of the hierarchy representation and pair-wise comparisons, AHP has become one of the best known and most widely used MCDA approaches. [7]

Let's recall the laptop example which was introduced previously, to show how AHP is applicable to a typical problem: a customer is urgently in need of a new laptop and has 3 aspects in mind which will affect his final purchasing choice. They are the utility with respect to price, the utility with respect to weight (size) and the utility with respect to performance (e.g. the memory capacity and CPU frequency). The store offers the customer three alternatives, C1, C2 and C3. Let's assume the customer compares the alternatives and finds that following holds.

$c_3$ is cheaper than $c_2$ which is cheaper than $c_1$

$c_3$ weighs less than $c_1$ which weighs less than $c_2$

$c_1$ performs better than $c_3$ which performs better than $c_2$

Each of the alternatives will meet the customers' requirements, but the customer has no idea which one to choose. To simplify this kind of problem, we will illustrate the example hierarchically below:

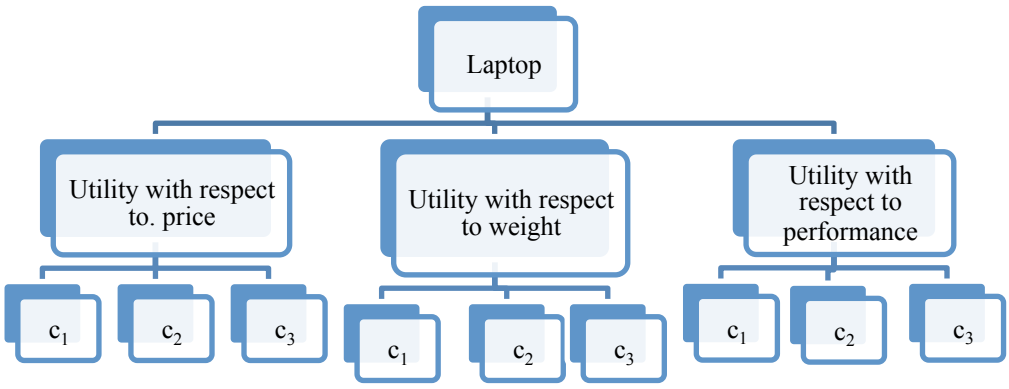



*Figure 1: A special case of Analytic Hierarchy Process (AHP)*

We denote criteria *utility with respect to price* ("price-utility") as $\alpha_1$, *utility with respect to weight* ("weight-utility") as $\alpha_2$ and *utility with respect to performance* ("performance-utility") as $\alpha_3$. To evaluate our laptop with respect to price-utility, i.e. $\alpha_1$, we perform pair-wise comparisons between $c_1$ $c_2$ and $c_3$. Assume the price-utility of $c_2$ is one third of $c_3$'s, and the price-utility of $c_2$ is also two times of $c_1$'s.

**Table 1: Pair-wise comparison between $c_1$, $c_2$ and $c_3$ with respect to price-utility**

| $\alpha_1$(price-utility) | $c_1$ | $c_2$ | $c_3$ |
|---|---|---|---|
| $c_1$ | 1 | 1/2 | 1/6 |
| $c_2$ | 2 | 1 | 1/3 |
| $c_3$ | 6 | 3 | 1 |

Theoretically, we can summarize the process of AHP method into three steps mentioned below:

**1. Hierarchy**
In order to help decision makers to find an alternative that best suits their goal and understanding the problem they are facing, AHP helps the decision makers to decompose their decision problem into a hierarchy of easily understandable sub-problems, each of which is relatively independent. The hierarchy includes decision goals, alternatives, and the criteria for assessing them.
**2. The pair-wise comparisons**
After the hierarchy has been built, the decision maker needs to compare each pair of alternatives.
**3. Assign weights**
Under the rule of AHP, the decision maker is often required to answer questions such as "How important is criterion A relative to criterion B". Rating the relative "priority" of the criteria is done by assigning a weight between 1 (equal importance) and 9 (extreme importance) to the more important criterion. [8]

The proponents of AHP argue that it is a flexible and straightforward process, which help users to structure the decision problem with hierarchies. On the other hand, a common criticism against AHP is the artificial limitation of the use of the 9-point scale. Furthermore, from a theoretical point of view, it is important to notice that application of AHP requires that utility is measured on a ratio scale. This is cognitively complicated and in decision theory it is often argued that utility can at best be measured on an interval scale. Notice however that if utility is measured on an interval scale then utility *differences* can be measured on a ratio scale. This idea is used in the research program mentioned in section 1.2.[9]

## 2.2 Some notions and ideas

This section will outline some notions and ideas in the research program mentioned in section 1.2.

### 2.2.1 Proportion

A proportion on a set A is a function from the Cartesian product of A with itself to the set of real numbers,

$$\rho: A \times A \rightarrow Re$$

which fulfills the following requirement called the *proportion condition*: for all a, b, c A

$$\rho(a, b) \bullet \rho(b, c) = \rho(a, c).$$

The use of proportions in decision analysis can be described as follows.

> "Aspects measured on ratio scales are closely related to proportions… For example, if utility is measured on a ratio scale, then there exists a utility proportion. If we imagine that person's notion of utility is measured on a ratio scale, then it can be represented by a proportion. In order to test if a person's notion of utility is measured on a ratio scale, we can ask him or her about the values for the utility proportions for different objects in some problem domain A. It is then possible to check if these values fulfill the proportion condition" [10].

### 2.2.2 Difference Proportion

Let's assume that ρ is a proportion of A. We make no special assumption about the character of the elements in A.

In general, a *diff-set* with respect to a set A and an aspect α is the set of positive differences between the objects in A with respect to α. E.g., if A consists of consequences a, b and c, and α is (with respect to α) better than b and b is better than c, then the diff-set consists of <a, b>, <a, c> and <b, c>, all of them positive. If c is better than b, which is better than a in some aspect, then the diff-set for that aspect is <c, b>, <c, a> and <b, a>. This case is quite similar to the situation for aspect $\alpha_1$ in our laptop example. If $A_2$ is a diff-set, then ρ: $A_2 \times A_2 \rightarrow Re^+$( the positive real numbers) is a *difference proportion* on $A_2$ if ρ is a proportion on $A_2$ and the following holds for all <a, b>, <b, c>,<a, c> and <x, y> $\in A_2$: ρ (<a, b>, <x, y>) +ρ (<b, c>, <x, y>) = ρ (<a, c>, <x, y>) [7] Based on this definition, we can create a kind of measure of the consistency (or rather the *inconsistency*) of the proportions between a number of utility differences. We note that the difference proportion condition above is fulfilled if and only if the difference between the left hand side (LHS) and the right hand side (RHS) of the equation above is 0. Therefore we define the following measure of the inconsistency of the proportions between <a, b>, <b, c> and <a, c>: I(<a,b>, <b,c>, <a,c>) = ρ(<a,b>, <x,y>) +ρ(<b,c>, <x,y>) - ρ(<a,c>, <x,y>) for any difference <x,y> $\in A_2$.

Note that if the difference proportion condition is fulfilled for some <a,b>,<b,c> and <a,c>, then I(<a,b>, <b,c>, <a,c>) equals 0. It means that the inconsistency is 0 i.e. the proportions between <a,b>, <b,c>, <a,c> are fully consistent.

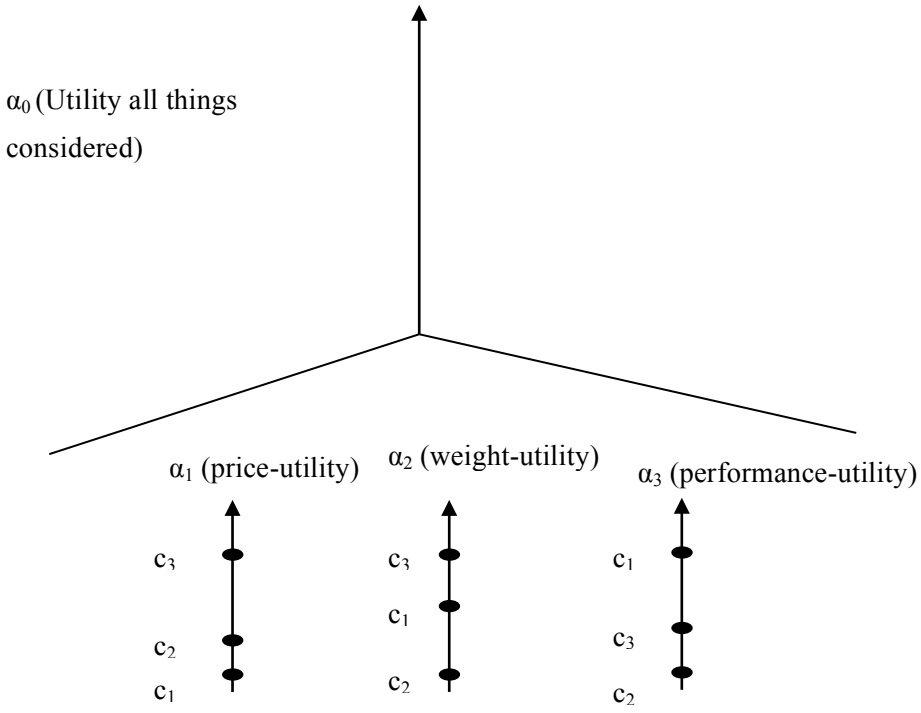
### 2.2.3 Utility Difference Proportion Matrices

A *utility difference* is the difference between utilities of two different consequences with respect to the same aspect. A utility difference-proportion matrix is a collection of proportions between utility differences. A matrix consists of intra-factorial or inter-factorial comparisons. An *intra-factorial difference proportion (sub-) matrix* includes pair-wise comparisons between several utility differences under one aspect while an *inter-factorial* one deals with proportions in the same kind of situation. As an example, we use the laptop purchase problem again; we took price-utility, performance-utility and weight (size)-utility as three aspects to think of, and we have three consequences: brand of laptop, they are B, A and C. Let's introduce a series of notation to make the laptop example be easily understood.

$c_i$ denotes the $i^{th}$ consequence, e.g. $c_1$, $c_2$, $c_3$ and so on. To simplify the notation, we often choose integers like 1, 2 and 3 instead of $c_i$. $n_c$, denotes the number of consequences. $\alpha_k$ is the name of $k^{th}$ aspect, e.g. $\alpha_1$, $\alpha_2$, $\alpha_3$. Aspect 0, $\alpha_0$, represents aspect utility all things considered. The number of aspects is denoted $n_\alpha$, $\Delta_k(c_i, c_j)$ means utility difference from $c_i$ to $c_j$ with respect to $\alpha_k$; it is shortened to <i, j> when there is no need for an explicit reference to the aspect number. $n_\Delta$ is the total number of utility differences, $\alpha_0$ not included.

Let's return to the same example as in section 2.1, with 3 consequences and 3 aspects. The consequences are named $c_1$, $c_2$ and $c_3$, but for simplicity, we refer to them as 1, 2 and 3. The aspects we consider are $\alpha_1$ (price-utility), $\alpha_2$ (weight-utility) and $\alpha_3$ (performance-utility). We note that $n_c$ =3 and $n_\alpha$ = 3.

The Figure below shows an aggregation tree which describes the situation of our laptop example:




The figure shows that, in $\alpha_1$, $c_3$ is better than $c_2$, which is better than $c_1$. The diff-set for $\alpha_1$ is {<3, 2>, <3, 1>, <2, 1>}. In $\alpha_2$, $c_3$ is better than $c_1$, which is better than $c_2$. The diff-set for $\alpha_2$ is {<3, 1>, <3, 2>, <1, 2>}. In α3, $c_1$ is better than $c_3$, which is better than $c_2$. The diff-set for α3 is {<1, 3>, <1, 2>, <3, 2>}. We note that $n_\Delta = 9$.

The figure below shows the intra-factorial difference proportion matrix for aspect $\alpha_1$ (price-utility). To facilitate reading, only the values above the diagonal are shown.

| $\rho_1$ | <3,2> | <3, 1> | <2,1> |
|---|---|---|---|
| <3,2> | 1 | $\rho_{1.1}$ (<3, 2>, <3, 1>) | $\rho_{1.1}$ (<3, 2>, <2, 1>) |
| <3,1> | | 1 | $\rho_{1.1}$ (<3, 1>, <2, 1>) |
| <2,1> | | | 1 |

$\rho_1$ (<3, 2>, <3, 1>), is the proportion between utility difference <3, 2> and <3, 1>, where <3, 2> is the utility difference between $c_1$ and $c_2$ under the aspect $\alpha_1$ and <3, 1> is the utility difference between $c_1$ and $c_3$ under the same aspect. $\rho_1$ (<3, 2>, <2, 1>) and $\rho_1$ (<3, 1>, <2, 1>) can be interpreted in the same way.

The figure below shows the intra-factorial difference proportion matrix for aspect $\alpha_2$ (weight-utility).

| $\rho_2$ | <3,1> | <3,2> | <1,2> |
|---|---|---|---|
| <3,1> | 1 | $\rho_{2.2}$ (<3, 1>, <3, 2>) | $\rho_{2.2}$ (<3, 1>, <1, 2>) |
| <3,2> | | 1 | $\rho_{2.2}$ (<3, 2>, <1, 2>) |
| <1,2> | | | 1 |

The figure below shows the intra-factorial difference proportion matrix for aspect $\alpha_3$ (performance-utility).

| $\rho_3$ | <1,3> | <1,2> | <3,2> |
|---|---|---|---|
| <1,3> | 1 | $\rho_{3.3}$ (<1, 3>, <1, 2>) | $\rho_{3.3}$ (<1, 3>, <3, 2>) |
| <1,2> | | 1 | $\rho_{3.3}$ (<1, 2>, <3, 2>) |
| <3,2> | | | 1 |

The figure below shows inter-factorial difference proportion matrix for aspect $\alpha_1$ (price-utility) and $\alpha_2$ (weight-utility).

| $\rho_{1.2}$ | <3,1> | <3,2> | <1,2> |
|---|---|---|---|
| <3,2> | $\rho_{1.2}$ (<3, 2>, <3, 1>) | $\rho_{1.2}$ (<3, 2>, <3, 2>) | $\rho_{1.2}$ (<3, 2>, <1, 2>) |
| <3,1> | | $\rho_{1.2}$ (<3, 1>, <3, 2>) | $\rho_{1.2}$ (<3, 1>, <1, 2>) |
| <2,1> | | | $\rho_{1.2}$ (<2, 1>, <1, 2>) |

Consistency can be checked for both intra-factorial and inter-factorial difference proportions. In this work, we focus on consistency checking of intra-factorial difference proportions. Let us recall the inconsistency measure that we defined in section 2.2.2:

$$I(<a,b>, <b,c>, <a,c>) = \rho(<a,b>, <x,y>) + \rho(<b,c>, <x,y>) - \rho(<a,c>, <x,y>)$$

As explained before, <a,b> is the utility difference between consequences a and b under some aspect. <x,y> can be any utility difference within the intra-factorial comparison matrix. Returning to the laptop case, we can substitute <a,b>, <b,c>, <a,c> and <x,y> with specific differences, e.g. <3,2>, <2,1>, <3,1> and <3,1> in aspect 1:

$$I(<3,2>, <2,1>, <3,1>) = \rho_1(<3,2>, <3,1>) + \rho_1(<2,1>, <3,1>) - \rho_1(<3,1>, <3,1>).$$

Let's illustrate again with our laptop case. Let us assume that the difference in price-utility ($\alpha_1$) between $c_2$ (B) and $c_1$ (C) is 1, and the price-utility difference between $c_3$ (A) and $c_1$ (C) is 5 and price-utility difference between $c_3$ (A) and $c_2$ (B) is 4.

The following figure shows the intra-factorial difference proportion matrix for this example:

| $\rho_1$ | <3,2> | <3,1> | <2,1> |
|---|---|---|---|
| <3,2> | 1 | 4/5 | 4 |
| <3,1> | 5/4 | 1 | 5 |
| <2,1> | 1/4 | 1/5 | 1 |

To calculate the inconsistency of the proportions between <3,2>, <2,1>, <3,1>, we insert the proper values in the measure:

$$I(<3,2>, <2,1>, <3,1>) = \rho_1(<3,2>, <3,1>) + \rho_1(<2,1>, <3,1>) - \rho_1(<3,1>, <3,1>) =$$
$$= 4/5 + 1/5 - 1 = 0$$

We conclude that the proportions involved in this example are consistent.
Let us now change the proportion between <3,2> and <3,1> from 4/5 to e.g. 3/5:

| $\rho_1$ | <3,2> | <3,1> | <2,1> |
|---|---|---|---|
| <3,2> | 1 | 3/5 | 4 |
| <3,1> | 5/4 | 1 | 5 |
| <2,1> | 1/4 | 1/5 | 1 |

Once again, we calculate the inconsistency:

$$I(<3,2>, <2,1>, <3,1>) = \rho_1(<3,2>, <3,1>) + \rho_1(<2,1>, <3,1>) - \rho_1(<3,1>, <3,1>) =$$
$$= 3/5 + 1/5 - 1 = -1/5$$

We see that the proportions between <3,2>, <2,1>, and <3,1> are no longer consistent since the inconsistency is not equal to 0.

# 3 Method

## 3.1 Design Issues

1. Which programming language and compile platform should we use?
2. How should the utility difference proportion matrices be represented?
    2.1. Which data structure should we use?
    2.2. By using this data structure, how can we get and set proportion values?
3. How to implement the consistency checking?

## 3.2 Design Choices

### 3.2.1 Platform

Java is the main programming language, which will be compiled and executed completely under the environment of Java 1.6. The compile platform is Eclipse. There are a lot of objected Oriented Programming languages. The reason why we choose Java is that it has a significant ability to be moved from one computer system to another, which makes it possible for our pilot program to be further developed by other users easily, compared to for example C++ and C. Another benefit is that Java is robust, which is pretty reliable and makes it possible to detect problems that may first show up during execution phase by Java compiler. Besides, Java is Multithreaded; it is the capability of Java itself to perform several tasks at once within a program.

### 3.2.2 Representation

Initially, we considered using some pre-defined matrix library such as JAMA, [11] EJML [12] or UJMP [13] for the representation of utility difference proportion matrices. We decided against them as these libraries are advanced, with a lot of functionality that will not be used in our project. We also considered simply using 2-dimensional arrays, like the following example:

```
Double matrix [] [] = new Double [r] [c];
```

Arrays are random access data structures, which means that retrieving or updating elements at known locations are fairly fast operations. On the other hand, the length of the array must be established before it is created. This means that the choice to represent utility difference proportion matrices as 2-dimensional arrays is less flexible. Our final choice was to use a representation based on maps of maps, as illustrated in the figure below

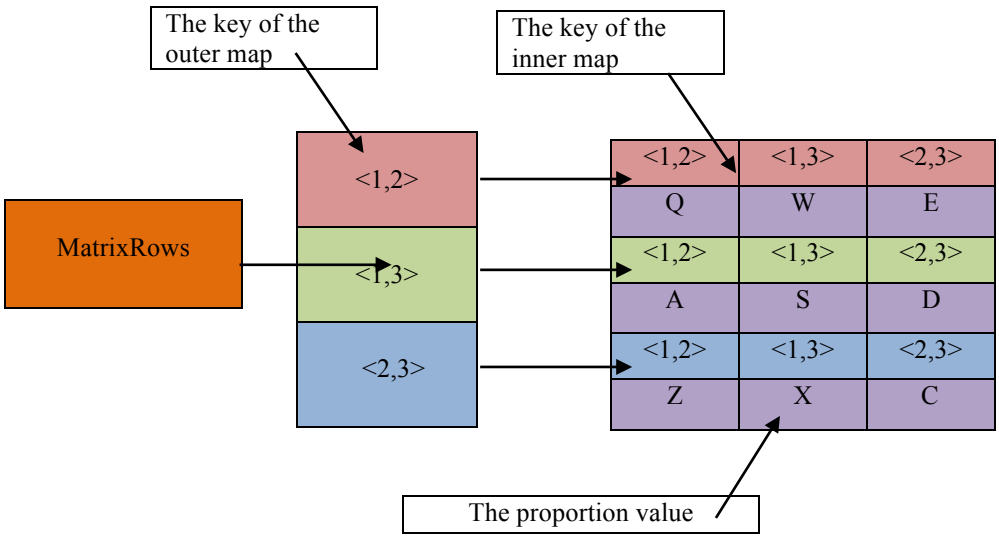



*Figure 2: The basic representation of map of maps*

To represent the maps of utility difference proportions, we used the utility class HashMap in the Java collection framework:

```
matrixRows = new HashMap<Diff, Map<Diff, Double>>();
```

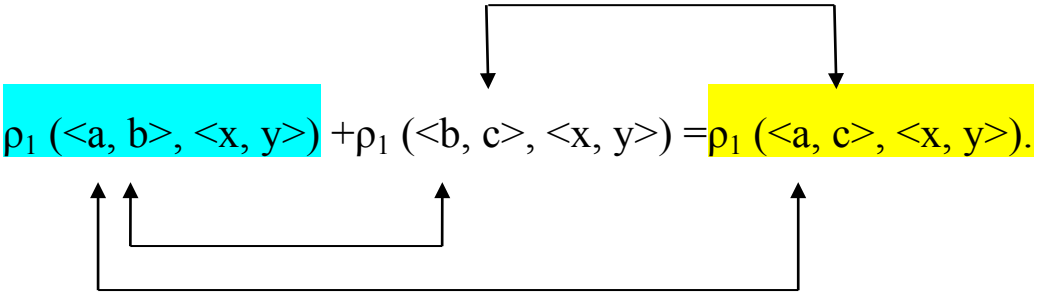
We need to use two keys to identify a specific utility difference proportion in the map of maps. A key identifies one of the utility differences in the proportion, and it is represented by a Diff object. The Diff class is presented in section 4.1.1.

Let's look at figure 2 we present above. In the "outer" HashMap matrixRows, the key is a Diff object and value is a Map <Diff, Double> object. In the "inner" maps, the keys are Diff objects and values are Double objects, representing the proportion between the differences represented by the first and second keys. The purpose of our design is to make us capable of retrieving information of the outer map by its key and then access its "inner" maps by their keys. As figure 2 shows, the program uses the key of the outer map to point to the key of the inner map, and then the proportion value can be retrieved in the same manner under the two keys. Let's look at a short example. Assume that we would like to get the proportion value D. We start with e.g. <1, 3> as the key of the outer map. Using this key, we retrieve the corresponding inner map, and then use the key <2, 3> to retrieve the proportion value D from the inner map.

As we can see, one feature of this representation is that the matrix is extendable with more utility differences. Furthermore, the diff-set (see section 2.2.2) of the matrix can easily be represented as a Java Set of Diff object which can be retrieved from the matrix.

### 3.2.3 Consistency checking

Let's now discuss the implementation of the inconsistency measure which we defined in section 2.2.2. Our goal is to achieve consistency checking that follows the theory that was discussed in section 2.2. Let us take a close look at the difference proportion criterion which is the basis for our inconsistency measure:




The equation presupposes a specific relation between the consequences (a, b, c, x, y) in the differences, as indicated by the arrows in the figure. I.e., the first consequence in the first difference of the first proportion must be the same as the first consequence in the first difference of the third proportion, and similarly for the other consequences. Furthermore, the second difference (<x, y> in the figure) of each proportion should be the same.

If all requirements we discussed above are met, we can conclude that the input proportions fulfill the ordering criteria and we can go on checking consistency for the equation. The implementation of the inconsistency measure is straightforward. We simply take the difference proportion between <a, b> and <x, y> plus the difference proportion between <b, c> and <x, y> minus the difference proportion between <a, c> and <x, y>, and return this result. As noted before, if the result is (nearly) 0, then the proportions are (nearly) consistent.

# 4 Result

## 4.1 Program Code

### 4.1.1 The Diff class

As already mentioned in section 3.2.2, utility differences are represented as Diff objects. Objects in this class are used as keys in utility difference proportion matrices.

Diff is a predefined class that was written by Magnus Hjelmblom. The code for the class is shown in Appendix A.

### 4.1.2 The IUtilityDifferenceProportionMatrix interface

An interface for utility difference proportion matrices is shown in Appendix B. It is a combination work of Magnus Hjelmblom and us. He defined the basic structure of it, and we revised it during the development of the matrix class.

### 4.1.3 The UtilityDifferenceProportionMatrix class

The interface is implemented in the UtilityDifferenceProportionMatrix class. The code for the class is shown in Appendix C. The matrix contains methods for getting proportions, setting proportions and calculating the consistency (or rather, inconsistency) of a set of proportions

The code for the *getProportion* method is shown below:

```
  @Override
public double getProportion(Diff first, Diff second) {

   Map<Diff, Double> row = matrixRows.get(first);

   if (row == null)
       throw new IndexOutOfBoundsException(
       "Can't find first index in Map(" + first + "," + second+ ")");
```

```
Double dv = row.get(second);

if (dv == null)
    throw new IndexOutOfBoundsException(
    "Can't find second index in Map(" + first + "," + second+ ")");
 return dv.doubleValue();
}
```

The method takes two parameters, “Diff first” and “Diff second”, which represent the two difference involved in this proportion. As explained in section 3.2.2, *first* will be used as a key of the outer map of Map matrixRows, and *second* will be used as a key of the inner map. The method simply retrieves the row corresponding to the first key. If it is retrieved successfully then the method continues to retrieve the proportion value corresponding to the second key. Our essential idea can be summarized with the following steps: first, get matrix row by using the first key, then get the matrix column by using the second key. At last, return the proportion. During the process, if the row or column does not exist , throw a runtime exception.

The code for the *setProportion* method is shown below:

```
@Override
public void setProportion(Diff first, Diff second, double
proportion) {
    Map<Diff, Double> row = matrixRows.get(first);

    if (row == null) {
        row = new HashMap<Diff, Double>();
        matrixRows.put(first, row);
    }

         row.put(second, proportion);

 }
```

The method takes three parameters, “Diff first” and “Diff second”, which represent the two differences involved in this proportion, and “double proportion”, which is the proportion value to set. Within this method, we first try to retrieve the row corresponding to the first Diff. If it exists, meaning that row is not *null*, then the proportion value is added to the row using the second key. If the row does not exist, we create a new HashMap<Diff, Double> object to represent this row, add the row to the outer HashMap, and then add the proportion value to this row using the second key. Values are added by using the *put* method in the Map interface.

The code of the consistency check method is presented in Appendix C, it is based on the inconsistency measure which was discussed in section 3.2.3. The *inconsistency* method takes four parameters, “Diff ab”, “Diff bc”, “Diff ac” and “Diff xy”, which represent the four differences that are involved in the consistency calculations. The method first checks the ordering criterion for consequences in the differences through a call to the *judgeDiff* method. If the ordering criterion is met, it then calculates the inconsistency value by a call to the *getInconsistencyValue* method, and then returns the answer.

The *judgeDiff* method also takes four Diff objects as parameters. It uses another assisting method, *isCriteriaMeet*, which simply checks that the first consequence of the first Diff is the same as the first consequence of the third Diff, and that the second consequence of the second Diff is the same as the second consequence of the third Diff, and that the second consequence of the first Diff is the same as the first consequence of the second Diff.

The *getInconsistencyValue* method takes four Diff object as parameters, it simply gets the corresponding proportions from the matrix, and calculates the inconsistency according to the inconsistency measure formula. The explanation of the basic idea behind these calculations is shown in section 3.2.2.

## 4.2 Test

To verify the functionality for the matrix class, we have developed four test programs. The first test program, *TestForConsistentCase*, illustrates a situation with consistent proportions. The second program, *TestForInconsistentCase*, illustrates a situation where the ordering check is fulfilled, but where the proportions are inconsistent. The third program, *TestForNotFulfilledCriteriaCase*, illustrates the situation where the ordering check is not fulfilled. Finally, we have included a test that illustrates *our laptop case* which has been used as a running example in our thesis.

### 4.2.1 Test 1: Consistent matrix

The first case (consistent proportions) is implemented in the test of *TestForConsistentCase* class, which is shown in Appendix D. The test program simply creates a new matrix with three Diff objects (see the figure below), and assigns some proportions between these Diffs and calculates the consistency between them.

```
Diff d32 = new Diff(3, 2);  //represents the difference from consequence c3 to c2
Diff d21 = new Diff(2, 1);  //represents the difference from consequence c2 to c1
Diff d31 = new Diff(3, 1);  //represents the difference from consequence c3 to c1
```

A transcript of a test run is shown below:

```
<3,2> : <3,1>==0.8
<2,1> : <3,1>==0.2
<3,1> : <3,1>==1.0
(<3,2>,<3,1>)+(<2,1>,<3,1>)=(<3,1>,<3,1>)
Consistent, The difference between LHS and RHS is : 0.0
```

*Figure 3: Program result from case consistency*

The transcript shows all the proportions between positive differences. As we see, the inconsistency value is 0. According to the discussion in section 3.2.3, the program considers the matrix to be consistent.

### 4.2.2 Test 2: Inconsistent matrix

The second case (inconsistent proportions) is implemented in test of *TestForInconsistentCase* class, the code is very similar to the *TestForConsistentCase* class, but with one proportion value slightly changed.

A transcript of a run of this test is shown below:

```
<3,2> : <3,1>==0.8
<2,1> : <3,1>==0.4
<3,1> : <3,1>==1.0
(<3,2>,<3,1>)+(<2,1>,<3,1>)=(<3,1>,<3,1>)
Inconsistent, The difference between LHS and RHS is : 0.20000000000000018
```

*Figure 4: Program result from case inconsistency*

As the transcript shows, the proportions in this case are considered inconsistent. As we see, the inconsistency value is around 0.2. Note that the theoretical value in this example is exactly 0.2, but this is not achieved because of some round-off errors in the floating point arithmetics.

### 4.2.3 Test 3: Input data does not fulfill the criteria

The third case (ordering check for proportion is not fulfilled) is implemented in the test of *TestForNotFulfilledCriteriaCase* class, the code is very similar to *TestForConsistentCase*, but with another set of diff parameters to the consistency method.

```
Diff testFirst = new Diff(1, 2);
Diff testSecond = new Diff(1, 3);
Diff testThird = new Diff(1, 3);
Diff testXY = new Diff(1,3);
```

The output of test result is shown below:

```
<3,2> : <3,1>==0.8
<2,1> : <3,1>==0.2
<3,1> : <3,1>==1.0
Exception in thread "main" se.hig.decision.NotMeetCriteriaException: Not fullfill criteria: (<3,2>,<3,1>) and (<3,1>,<3,1>)
        at se.hig.decision.UtilityDifferenceProportionMatrix.inconsistency(UtilityDifferenceProportionMatrix.java:119)
        at TestForNotFulfilledCriteriaCase.main(TestForNotFulfilledCriteriaCase.java:28)
```

*Figure 5: Program result from case not fulfilled criteria*

As the transcript shows, the diffs do not fulfill the ordering criteria. The program throws an exception to inform the user about this.

### 4.2.4 Test 4: Special case for laptop

The fourth case (our laptop case) is implemented in *TestForLaptopExample* class which shown in Appendix E. First, the program establishes the utility values for the three consequences in the first aspect. Then it calculates the utility differences between the consequences. Thirdly, the program calculates the proportions between the differences, and inserts the proportions into the matrix. Finally, it calculates the consistency for one proportion.

:

```
con3.put(con.get("1"), 4); // delta(3,2)=4
con3.put(con.get("2"), 1); // delta(2,1)=1
con3.put(con.get("3"), 5); // delta(3,1)=5

Diff testFirst = new Diff(3, 2);
Diff testSecond = new Diff(2, 1);
Diff testThird = new Diff(3, 1);
Diff testXY = new Diff(3,1);
```

The output of test result is shown below:

```
<3,2>,<3,2>==1.0
<3,2>,<2,1>==4.0
<3,2>,<3,1>==0.8
<2,1>,<3,2>==0.25
<2,1>,<2,1>==1.0
<2,1>,<3,1>==0.2
<3,1>,<3,2>==1.25
<3,1>,<2,1>==5.0
<3,1>,<3,1>==1.0
(<3,2>,<3,1>)+(<2,1>,<3,1>)=(<3,1>,<3,1>)
Consistent, The difference between LHS and RHS is : 0.0
```

*Figure 6: Program result from Laptop case*

The transcript shows the laptop case we discussed in section 2.1. As we see, the left hand side is equal to right hand side of the difference proportion equation, so the inconsistency value is 0.0. In other words, the program considers the proportions to be consistent.

# 5 Discussion and conclusion

This report presents a small pilot study within a research program in decision analysis, initiated by the decision, risk and policy analysis group at the University of Gävle. The aim of this research program is to explore the theoretical foundations of an analytical decision support tool based on additivity and proportionality. We have developed a class for representing utility difference proportion matrices which internally uses a map of maps to represent the matrix. The class contains methods for getting and setting proportion values, and a method for checking the consistency of a set of utility difference proportions. The consistency checking involves an ordering check and inconsistency calculations. We have also developed four simple test classes, which have been used to verify and illustrate the functionality of the matrix class.

It could perhaps be argued that it is very difficult to determine utility difference proportions. How well the human brain can handle proportions between differences will be left as an open question. In fact, it could be a subject of a follow-up study; see Future work below.

# 6 Future work

When users are faced with a problem with several tricky alternatives, an analytical decision support tool can help them analyze it, in order to provide the best choice by judging from different criteria. We hope that our pilot study can contribute to the development of such a tool based on the notion of utility difference proportions. The application field of this tool would be to analyze multi-criteria decision problems where the utility of each aspect can be measured on an interval scale.

A natural next step in the development of the utility difference proportion matrix class is to define some measure of the inconsistency related to an individual proportion. And then construct and implement an algorithm to calculate this inconsistency. For example, when the user adds a new proportion, the matrix could search for all proportions that are connected to this proportion by transitivity, and then inform the user of the inconsistency with respect to the connected proportions. The inconsistency of an individual proportion could e.g. be expressed as an aggregated inconsistency value or as a list of inconsistency values with respect to each connected proportion. Consistency checking could be even further developed, e.g. by defining a measure of the consistency for the whole matrix and constructing a method that implements this measure. Moreover, we could implement an algorithm for calculation

of utility values, and a user interface that interactively lets the users manipulate the utility difference proportion matrix.

An interesting follow-up study, suggested by Jan Odelstad at the University of Gävle, would be to modify our test applications and use them in an experiment to empirically test how consistent humans can be in their judgments. The participants in the experiment could for example be asked to establish some difference proportions, and the application could calculate how consistent they are.

# 7 References


[1] Habiba, U and Asghar, S., "A survey on multi-criteria decision making approaches," *Proceedings of the 5th IEEE International Conference on Emerging Technologies 2009 (ICET 2009).* pp.321-325, 19-20 Oct. 2009

[2] Ke H.F and Liu S.F; "Similarity to ideal-based grey relational projection for multiple criteria decision-making", *Grey Systems and Intelligent Services, 2009. GSIS 2009. IEEE International Conference on* , vol., no., pp.1008-1012, 10-12 Nov.2009

[3] *Multi-criteria decision analysis.* In Wikipedia, the Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Multicriteria_decision_analysis&oldid=488334353 (visited2012-04-20)

[4] Odelstad, J and Hjelmblom, M, *Utility Difference Proportions*. (Unpublished manuscript).

[5] Odelstad, J. *Mätning och mellanbegrepp* （Measurement and intermediaries, 2011, unpublished manuscript)

[6] *Analytic Hierarchy Process*. In Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/w/index.php?title=Analytic_Hierarchy_Process&oldid=492066672 (visited 2012-05-12)

[7] Macharis, C., Springael, J., De Brucker, K., Verbeke, A. 2004: "Promethee and AHP: The design of operational synergies in multicriteria analysis. Strengthening Promethee with ideas of AHP". *European Journal of Operational Research* 153: 307.317.

[8] *Expert Choice* http://www.expertchoice.com/about-us/ (visited 2012-05-20)

[9] Saaty T.L., "Highlights and critical points in the theory and application of the Analytic hierarchy process", *European Journal of Operational Research*, 426-447(1994)

[10] Odelstad, J: *Eudoxos — A Research Programme in Decision Analysis* (Unpublished Manuscript）.

[11] *A Java Matrix Package (JAMA)* http://math.nist.gov/javanumerics/jama/ (visited 2012-08-5)

[12] *Efficient Java Matrix Library (EJML)* http://code.google.com/p/efficient-java-matrix-library/ (visited 2012-08-5)

[13] *Universal Java Matrix Package (UJMP)* http://sourceforge.net/projects/ujmp/ (visited 2012-08-5)

# 8 Appendices

## 8.1 Appendix A

```
package se.hig.decision;

/**
 * A class that represents pairs of consequence indices.
 * Objects of this class are used as keys in utility difference
proportion matrices.
 *
 * @author mbm
 *
 */
public class Diff {
    private int firstConsequence;
    private int secondConsequence;

    private int M = 19;
    private int N = 37;

    /**
     *
     * @param firstConsequence First consequence
     * @param secondConsequence Second consequence
     */
    public Diff(int firstConsequence, int secondConsequence) {
                super();
                this.firstConsequence = firstConsequence;
                this.secondConsequence = secondConsequence;
    }

    /**
     *
     * @return The index of the first consequence
     */
    public int getC1() {
                return firstConsequence;
    }

    /**
     *
     * @return The index of the second consequence
     */
    public int getC2() {
                return secondConsequence;
    }

    @Override
    public int hashCode() {
            int hashValue = (firstConsequence%M +
    M*secondConsequence)%N;
                return hashValue;
    }

    @Override
    public boolean equals(Object obj) {
                if (obj instanceof Diff) {
```

```
            Diff other = (Diff) obj;
            if(this.firstConsequence == other.firstConsequence &&
            this.secondConsequence == other.secondConsequence)
            return true;
            }
            return false;
}

@Override
public String toString() {
return "<" + firstConsequence + "," + secondConsequence + ">";
            }
}
```

## 8.2 Appendix B

```
package se.hig.decision;

/**
 * Interface for a utility difference proportion matrix, in which
 * proportions are represented with double values.
 *
 *
 *
 */
public interface IUtilityDifferenceProportionMatrix {
    /**
     *
     * @return The number of consequences connected to this matrix.
     */
    int getNrOfConsequences();

    /**
     * @return The number of the first aspect for this matrix.
     */
    int getFirstAspectNr();

    /**
    * @return The number of the second aspect aspect for this matrix.
     */
    int getSecondAspectNr();

    /**
     * @param first First difference.
     * @param second Second difference.
     * @return The proportion between first and second difference.
     */
    double getProportion(Diff first, Diff second);

    /**
     * @param first First difference.
     * @param second Second difference.
     * @param proportion The proportion between first and second
difference.
     */
    void setProportion(Diff first, Diff second, double proportion);

    /**
     *
     * @param first First difference.
     * @param second Second difference.
     * @return A measure of the inconsistency of the proportion from
first to second.
     */
   double inconsistency(Diff first, Diff second, Diff third,Diff xy);
}
```

## 8.3 Appendix C

```
package se.hig.decision;

import java.util.HashMap;
import java.util.Map;

public class UtilityDifferenceProportionMatrix implements
            IUtilityDifferenceProportionMatrix {
 private int nOfConsequences;
 private int firstAspectNr;
 private int secondAspectNr;
 private Map<Diff, Map<Diff,Double>> matrixRows;

 /**
  * @param nOfConsequences The number of consequences connected to
 this matrix.
  * @param firstAspectNr The number of the first aspect of this
 matrix.
  * @param secondAspectNr The number of the second aspect of this
 matrix.
  */
 public UtilityDifferenceProportionMatrix(int nOfConsequences,
                        int firstAspectNr, int secondAspectNr) {
            this.nOfConsequences = nOfConsequences;
            this.firstAspectNr = firstAspectNr;
            this.secondAspectNr = secondAspectNr;
            matrixRows  =  new  HashMap<Diff,Map<Diff,Double>>();  //
 New empty map of rows
 }

 @Override
 public int getNrOfConsequences() {
            return nOfConsequences;
 }

 @Override
 public int getFirstAspectNr() {
            return firstAspectNr;
 }

 @Override
 public int getSecondAspectNr() {
            return secondAspectNr;
 }



 @Override
 public double getProportion(Diff first, Diff second) {

            Map<Diff, Double> row = matrixRows.get(first);

            if (row == null)
                        throw  new  IndexOutOfBoundsException("Can't
 find first index in Map("+first+","+second+")");

            Double dv = row.get(second);
```

```
            if (dv == null)
                        throw  new  IndexOutOfBoundsException("Can't
find second index in Map("+first+","+second+")");

            return dv.doubleValue();

}

@Override
public void setProportion(Diff first, Diff second, double proportion)
{
            Map<Diff, Double> row = matrixRows.get(first);

            if (row == null) {
                        row = new HashMap<Diff, Double>();
                        matrixRows.put(first, row);
            }

            row.put(second, proportion);

}


private boolean judgeDiff(Diff first, Diff second, Diff third, Diff
key) {
            boolean flag = false;

            if (isCriteriaMeet(first, second, third)) {
                        flag = true;

System.out.println(RightProportionIndexToString(first, second, third,
key));
            }
            return flag;
}

private String RightProportionIndexToString(Diff first, Diff second,
Diff third,
                        Diff key) {
            return "(<" + first.getC1() + "," + first.getC2()
+ ">,<"+key.getC1()+","+key.getC2()+">)+(<" + second.getC1() + "," +
second.getC2()
+ ">,<"+key.getC1()+","+key.getC2()+">)=(<" + third.getC1() + "," +
third.getC2()
+ ">,<"+key.getC1()+","+key.getC2()+">)";
}

private String WrongProportionIndexToString(Diff first, Diff second,
Diff third, Diff key) {
            return "(<" + first.getC1() + "," + first.getC2()
+ ">,<"+key.getC1()+","+key.getC2()+">) and (<" + second.getC1() +
"," + second.getC2()
+ ">,<"+key.getC1()+","+key.getC2()+">)";
}

private boolean isCriteriaMeet(Diff first, Diff second, Diff third)
{
            return first.getC1() == third.getC1() &&
                                    second.getC2()  ==  third.getC2()
```

```
                                                    &&
                                        first.getC2() == second.getC1();
    }


    public double inconsistency(Diff ab, Diff bc,Diff ac,Diff xy) throws
    NotMeetCriteriaException {
                if(judgeDiff(ab,bc,ac,xy)){
                            return getInconsistencyValue(ab, bc, ac, xy);
                }
                else{
                             throw    new    NotMeetCriteriaException("Not
    fullfill criteria: "+

    WrongProportionIndexToString(ab, bc, ac, xy));
                }
    }


    private double getInconsistencyValue(Diff ab, Diff bc, Diff ac, Diff
    xy){
                return getProportion(ab, xy) + getProportion(bc, xy) -
    getProportion(ac, xy);
    }

}
```

## 8.4 Appendix D

```
import se.hig.decision.Diff;
import se.hig.decision.UtilityDifferenceProportionMatrix;


public class TestForConsistentCase {

    public static void main(String[] args) {
                UtilityDifferenceProportionMatrix proportion = new
UtilityDifferenceProportionMatrix(1, 2, 3);

                Diff d32 = new Diff(3, 2); //represents the difference
from consequence c3 to c2
                Diff d21 = new Diff(2, 1);//represents the difference from
consequence c2 to c1
                Diff d31 = new Diff(3, 1); //represents the difference
from consequence c3 to c1


                proportion.setProportion(d32, d31, 0.8);
                System.out.println(d32.toString() + " : " + d31.toString()
                + "==" + proportion.getProportion(d32, d31));
                proportion.setProportion(d21, d31, 0.2);
                System.out.println(d21.toString() + " : " + d31.toString()
                + "==" + proportion.getProportion(d21, d31));
                proportion.setProportion(d31, d31,1.0);
                System.out.println(d31.toString() + " : " + d31.toString()
                + "==" + proportion.getProportion(d31, d31));

                Diff testXY = new Diff(3,1);
                double result = proportion.inconsistency(d32, d21,
d31,testXY);
                if(result != 0){
                            System.out.println("Inconsistent, The
difference between LHS and RHS is : " + result);
                }
                else{
                            System.out.println("Consistent, The difference
between LHS and RHS is : "+ result);
                }

    }

}
```

## 8.5 Appendix E

```
import java.util.HashMap;
import java.util.Map;

import se.hig.decision.Diff;
import se.hig.decision.UtilityDifferenceProportionMatrix;


public class TestForLaptopExample {

    public static void main(String[] args) {
                UtilityDifferenceProportionMatrix proportion = new
UtilityDifferenceProportionMatrix(1, 2, 3);

                Map<String, String> con = new HashMap<String, String>();
                con.put("1", "32");
                con.put("2", "21");
                con.put("3", "31");
                Map<String, String> con2 = new HashMap<String, String>();
                con2.putAll(con);

               Map<String, Integer> con3 = new HashMap<String, Integer>();

                con3.put(con.get("1"), 4); // delta(3,2)=4
                con3.put(con.get("2"), 1); // delta(2,1)=1
                con3.put(con.get("3"), 5); // delta(3,1)=5

                for (int i = 1; i <= con.size(); i++) {
                           for (int j = 1; j <= con2.size(); j++) {
                                      String firstvalue1 =
con.get(String.valueOf(i)).substring(0, 1);
                                      String firstvalue2 =
con.get(String.valueOf(i)).substring(1, 2);
                                      Diff first = new
Diff(Integer.parseInt(firstvalue1),

    Integer.parseInt(firstvalue2));

String secondvalue1 = con2.get(String.valueOf(j)).substring(0,1);
String secondvalue2 = con2.get(String.valueOf(j)).substring(1,2);

Diff second = new Diff(Integer.parseInt(secondvalue1),
Integer.parseInt(secondvalue2));
double proportionValue = con3.get(con.get(String.valueOf(i))).doubleValue()
                                                                      /
con3.get(con2.get(String.valueOf(j))).doubleValue();
System.out.println(first.toString() + ":" + second.toString()+ "==" +
proportionValue);
proportion.setProportion(first, second, proportionValue);
                           }
                }

                Diff testFirst = new Diff(3, 2);
                Diff testSecond = new Diff(2, 1);
                Diff testThird = new Diff(3, 1);
                Diff testXY = new Diff(3,1);
```

```
double result = proportion.inconsistency(testFirst, testSecond,
testThird,testXY);
                if(result != 0){
                            System.out.println("Inconsistent, The
difference between LHS and RHS is : " + result);
                }
                else{
                            System.out.println("Consistent, The difference
between LHS and RHS is : "+ result);
                }

    }

}
```

# 9 Tables and Figures

## 9.1 Figures



## 9.2 Tables