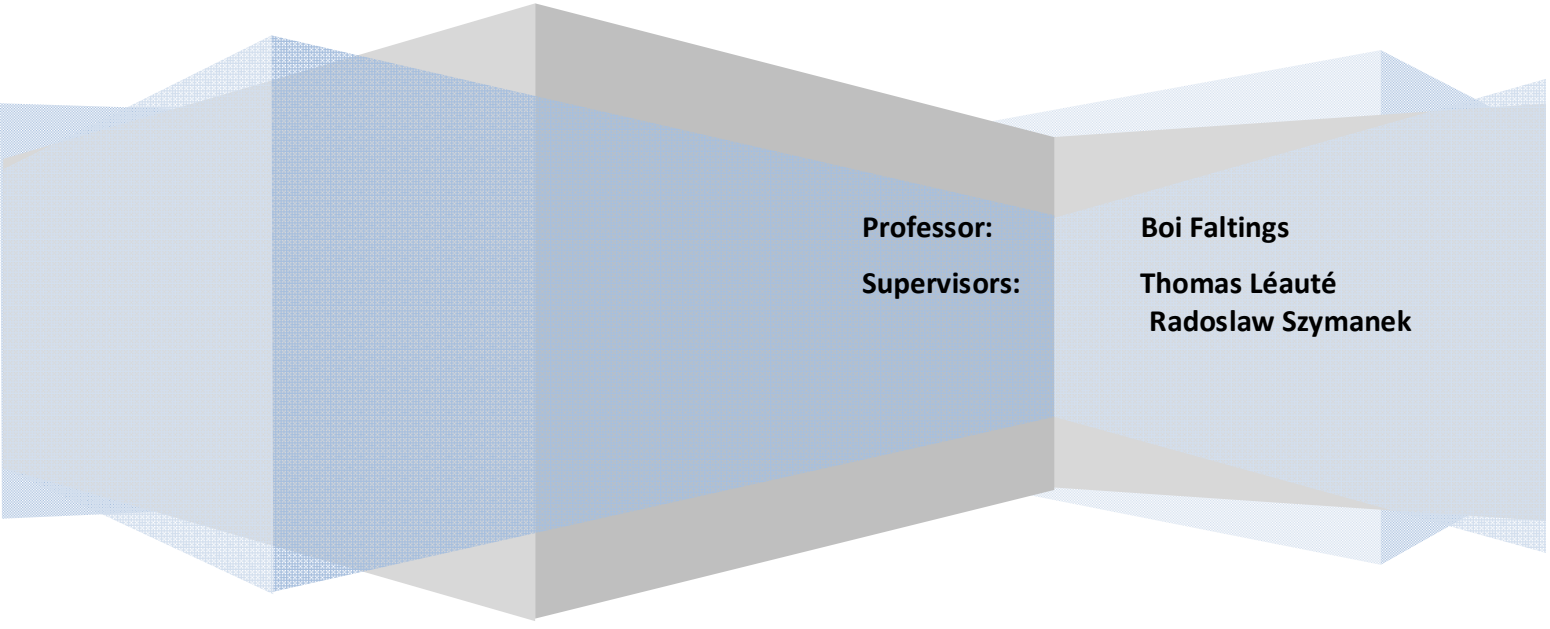
Artificial Intelligence Laboratory

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

# Efficient Data Structures for Decision Diagrams

Master Thesis

Nacereddine Ouaret



**Professor:** Boi Faltings

**Supervisors:** Thomas Léauté
Radoslaw Szymanek

# Contents

# List of figures

# 1 Introduction

Dynamic Programming OPtimization (DPOP) [1] is an algorithm proposed for solving distributed constraint optimization problems. In this algorithm, Hypercubes [1] are used as the format of the messages exchanged between the agents. Then, Hybrid-DPOP (H-DPOP) [2], a hybrid algorithm based on DPOP was proposed as an improved solution. Its main advantage over DPOP lies under the use of Multi-valued Decision Diagrams (MDDs) [3] instead of Hypercubes. Furthermore, MDDs give a more compact representation of the messages. The MDDs were implemented and presented in [2] by Kumar even though it was presented as Constrained Decision Diagrams (CDD) [4] which is a generalization of MDDs.

The first part of this project aims at implementing Hypercubes. An Implementation already exists. Therefore, the goal is to propose an implementation which is more efficient in term of speed and memory usage for the already implemented functions (join, project and slice). The solution implemented during this project should also provide new functions (split and re-order) and the possibility of saving Hypercubes in the XML [5] format. The second part of the project consists in proposing and implementing a data structure that is more efficient than the MDDs implemented in H-DPOP. This data structure should also provide more functions, and also provide the possibility of saving data in the XML format.

The report is organized as follows. Initially, we present an overview of Hypercubes in section 2. This includes the different operations applied to it. Further, the implementation of Hypercubes as well as its different operations is discussed in section 3. Tests are performed to verify the correctness of the different operations on Hypercubes in section 4. Similarly to Hypercubes, utility diagrams are presented, implemented and evaluated in sections 5, 6 and 7 respectively. Finally, some concluding remarks are drawn in section 8.

# 2 Hypercube

## 2.1 Definition

A Hypercube is represented by three parameters: ***variables (V)***, ***domains (D)***, and ***values (U)*** as shown in **Figure 1**. ***V*** $\{V_1, \dots, V_n\}$ is an array of the variables contained in the hypercube. ***D*** $\{D_1, \dots, D_n\}$ is an array of domains ordered such that $D_i$ corresponds to $V_i$ in the hypercube. Each $D_i$ is an array where each entry is a possible value for $V_i$. ***U*** $\{U_1, \dots, U_m\}$ is an array containing the utility values of the hypercube. There is a utility value for each combination of possible variables values ($m = |D_1| \dots |D_n|$). The utility values are ordered such that the $j^{th}$ utility value is related to the variables values $(v_{1,k_1}, \dots, v_{n,k_n})$ by the following equation

$$j = \sum_{i=0}^{n} k_i \prod_{l=i}^{n} |D_l|$$

where $k_i$ is the index of $v_{i,k_i}$ in the ordered domain $D_k$ (the index of the first value in each domain is 0 not 1).

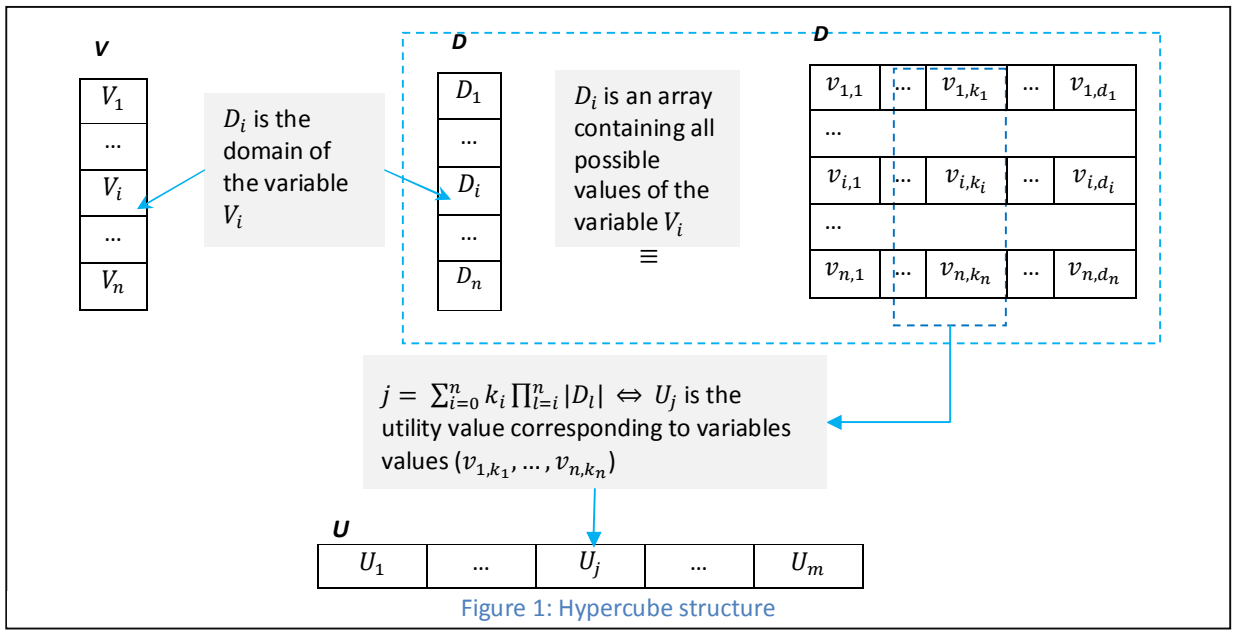


Figure 1: Hypercube structure

To illustrate this definition better, let us consider the multi-valued function represented by **table 1**. It can be represented by the Hypercube shown in **Figure 2**. This Hypercube contains three variables {*X0*, *X1*, and *X2*} which compose the array *V*. Variables *X0* and *X2* have only two possible values, 0 or 1. This means that their respective domains are only composed of these two values. Variable *X1* has three possible values, 0, 1 or 2. By grouping the domains of the variables in one array and ordering them in the correct order (the domain of *X0*, then the domain of *X1*, and finally the domain of *X2*), the array *D* is obtained as depicted in **Figure 2**. Finally, the utility values are placed in *U* according to the utility equation.

| Variables | | | Utility values |
|---|---|---|---|
| X0 | X1 | X2 | |
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 2 |
| 0 | 1 | 1 | 3 |
| 0 | 2 | 0 | 4 |
| 0 | 2 | 1 | 5 |
| 1 | 0 | 0 | 6 |
| 1 | 0 | 1 | 7 |
| 1 | 1 | 0 | 8 |
| 1 | 1 | 1 | 9 |
| 1 | 2 | 0 | 10 |
| 1 | 2 | 1 | 11 |

Table 1 : Example of a multi-valued function

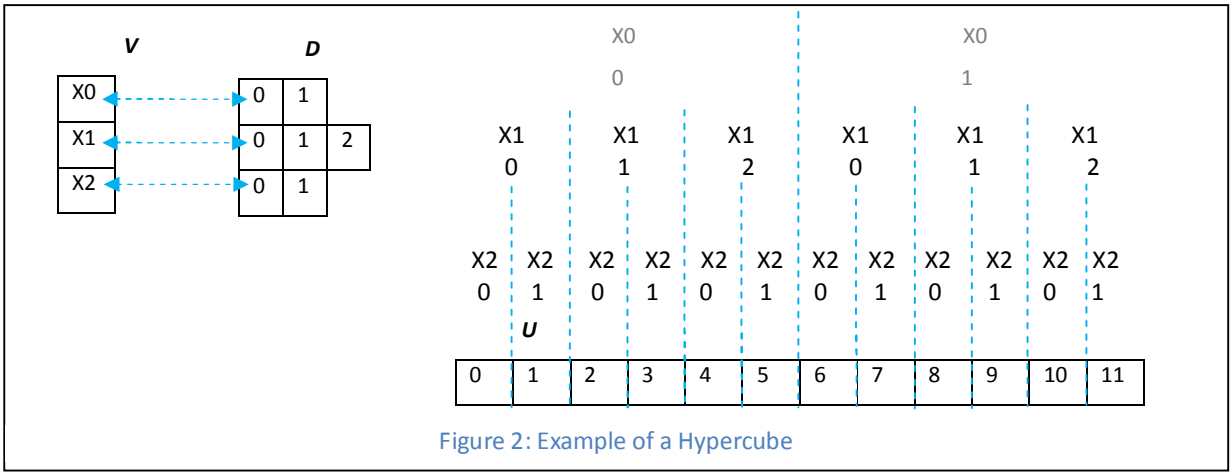



Figure 2: Example of a Hypercube

## 2.2 Operations on Hypercubes

### 2.2.1 Join

The operation of joining two Hypercubes consists in computing the three parameters (i.e. $V_{join}$,$D_{join}$, and $U_{join}$) representing the resulting Hypercube. As illustrated in the example in **Figure 3,** $V_{join}$ is obtained by the union of the variables arrays of the Hypercubes to join. For each variable $V_{join_i} \in V_{join}$ the corresponding domain $D_{join_i}$ is obtained by computing the intersection of the domains of this variable in all the Hypercubes to join. In case there is only one Hypercube containing this variable, then no intersection is calculated and this domain is kept as the domain of this variable in the new Hypercube. The entries of the utility values array $U_{join}$ are obtained by computing the sum of the utility values corresponding to the same variables values. In this example, the utility value corresponding to *X0* = 3 and *X1* = 1 in the Hypercube *h1* is 2 and the utility value corresponding to *X1* = 1 and *X4* = 5 is 3. Thus, the third entry in $U_{join}$ corresponding to *X0* = 3, *X1* = 1 and *X4* = 5 is 5.

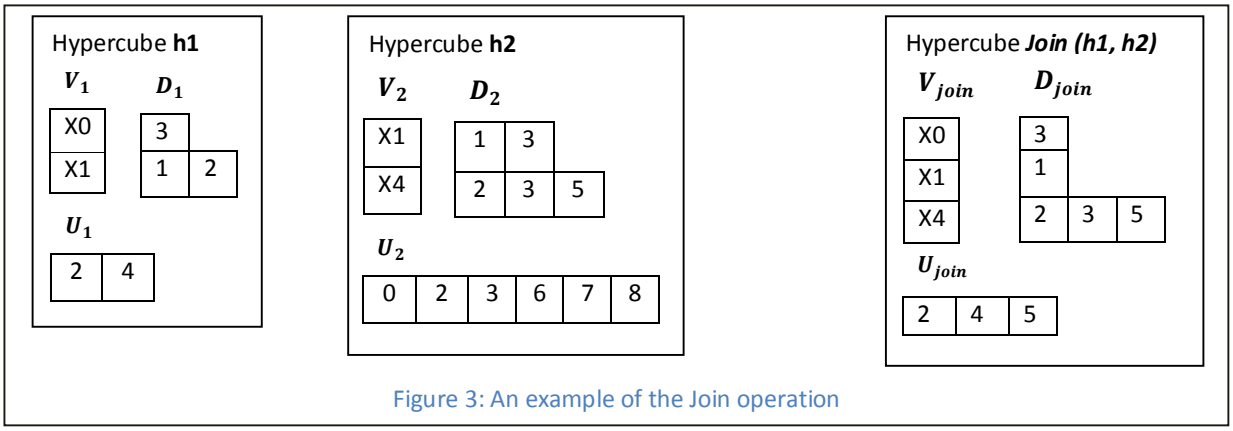



Figure 3: An example of the Join operation

### 2.2.2 Projection

The operation of projecting variables out of a Hypercube consists in reducing its dimension by removing some of its variables. This implies reducing also the array containing the utility values. The reduction of the utility values array is performed by taking either the maximum or the minimum over the set of utility values obtained by fixing the variables to keep, and varying the variables to project out over their domains. Let us consider the example shown in **Figure 4**. The Hypercube *h* has three variables, *X1*, *X3* and *X5*. When projecting variables X1, and X3 out of *h*, the resulting Hypercube contains only the variable *X5*. This variable has a domain of size two. This implies that the resulting Hypercube has only two utility values. **Figure 5** shows in details how these two utility values are obtained from the utility values of the Hypercube *h*. The first utility value corresponds to *X5* = 0. In the Hypercube *h*, the utility values corresponding to *X5* = 0 are 0, 2, 4, 6, 8, and 10. By taking the maximum, the utility value corresponding to *X5* = 0 is equal to *max(*5, 4, 0, 10, 9, 1*)* = 10. Similarly, the utility value corresponding to *X5* = 1 is 11.

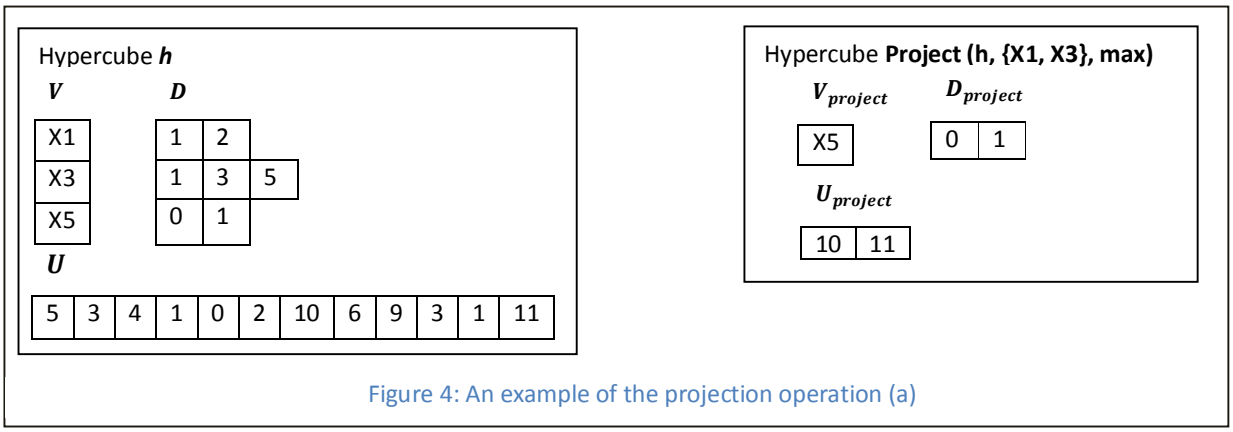



Figure 4: An example of the projection operation (a)

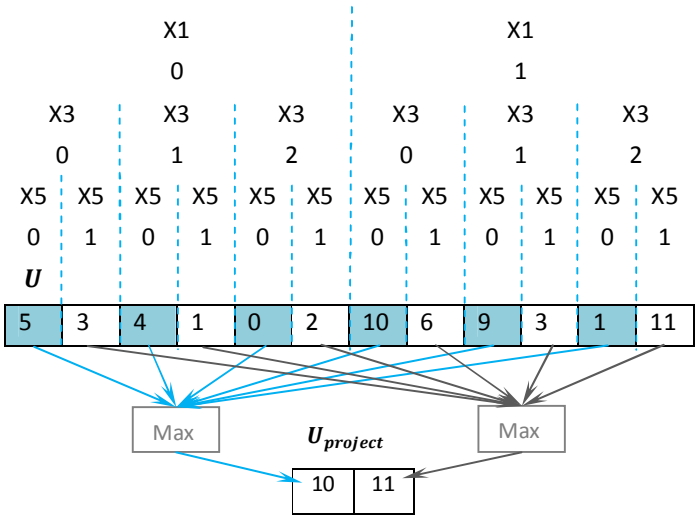



Figure 5: An example of the projection operation (b)

### 2.2.3 Slice

Slicing a Hypercube consists in reducing the domains of some of its variables. This implies reducing also the utility values array by keeping only the utility values corresponding to combinations of variables values included in the provided sub-domain. To illustrate clearly this operation, let us consider the example shown in **Figure 6**. A new domain is provided for the variable *X1*. This domain contains only the values 2 and 3. So, only utility values corresponding to *X1* = 2 or 3 are kept in the utility values array of the resulting Hypercube. A special case of this operation is when the provided sub-domain contains only one value. In this case, there is no need to keep this variable in the resulting Hypercube since it has only one possible value.

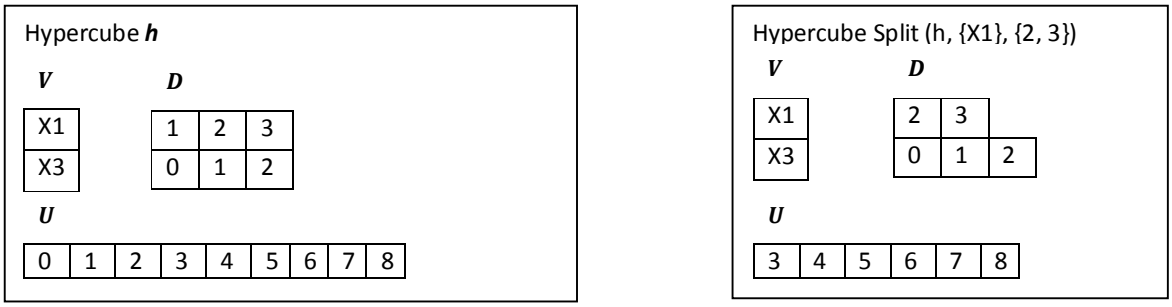



Figure 6: An example of the slice operation

### 2.2.4 Split

Splitting a Hypercube consists in removing some utility values from the hypercube based on a provided utility value threshold by keeping only utility values greater than the provided threshold. This implies reducing also the domains of the variables. Here, we need to point out that in most cases, we will have to keep some utility values even if they are smaller than the threshold. This comes from the fact that Hypercubes assume that there is a utility value for each possible combination of variables values. **Figure 7** shows one example of these cases. In the latter, the threshold is equal to 7. Since the utility value corresponding to *X1* = 1 and *X2* = 5

is 9, and the one corresponding to *X1* = 2 and *X2* = 3 is 11, the domain of *X1* in the Hypercube resulting from the slice must contain the values 1 and 2 and the domain of X2 must contain 3 and 5. Furthermore, the resulting Hypercube must also contain the utility value 2 corresponding *X1* = 2 and *X2* = 5 even if it is smaller than the threshold.


Figure 7: An example of the split operation

# 3 Hypercubes Implementation

All the classes related to Hypercubes are grouped in the *hypercube* package. As shown in **Figure 8**, this package contains three classes, *Hypercube*, NullHypercube, and *HVector,* and the interface *Addable*.

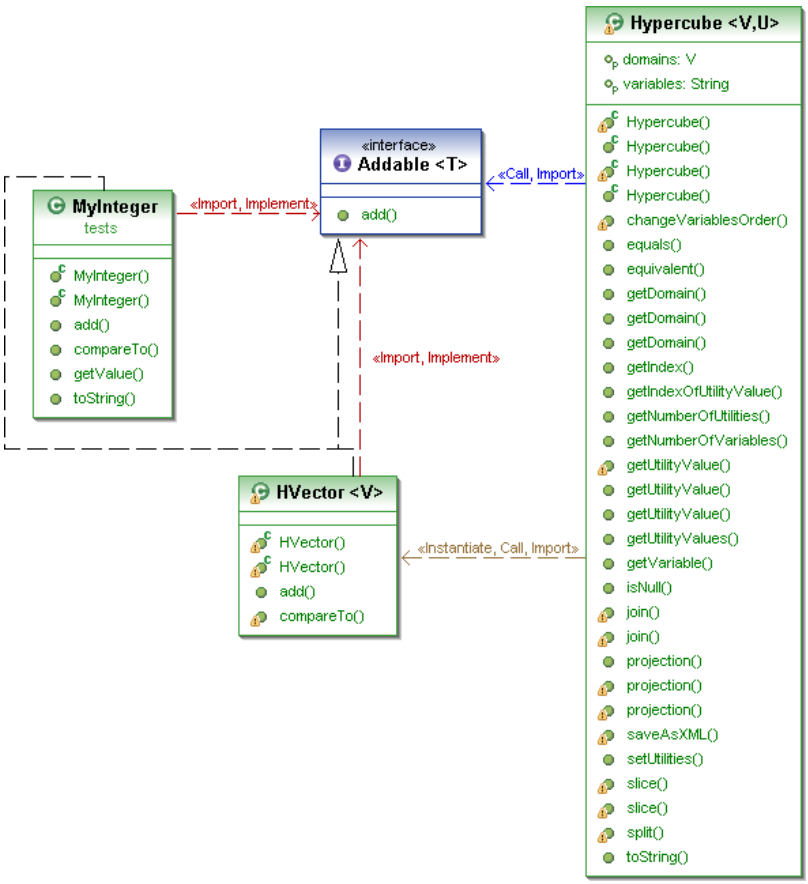



Figure 8: UML diagram of the *hypercube* package

## 3.1 hypercube.Hypercube

The class *Hypercube* is the principal class implementing the Hypercube and its different operations. This class has four parameter variables, *variables, domains, values, and step_hashmaps*. The first three variables represent the three parameters defining the Hypercube ($V, D$ *and* $U$).

*Variables* is declared as a one dimensional array of *String* objects representing the names of the Hypercube variables.

*domains* is declared as a 2-dimensional array of a generic type *V* extending the *Comparable* interface. This allows using diverse types of variables values but at the same time restricts them to the ones extending the JAVA *Comparable* interface. The ability to compare variables values along with the assumption that domains are sorted allows an easier and more implementations of the operations.

Let us take for example the *intersection* method in the class *Hypercube*. This method computes the intersection of two arrays of variable values (*array1*, and *array2*). This method uses two indexes (*index1*, and *index2*) to go through the two arrays. The indexes are set initially to 0. The value at position *index1* in array1 is compared with the one in position *index2* in *array2*. Three cases are possible here:

- If they are equal then this common value is inserted in the resulting array and both *index1* and *index2* are incremented by one.
- The value in *array1* at position *index1* is bigger than the one in *array2* at position *index2*. In this case, we can already conclude that *array1* does not contain the value in *array2* at position *index2* since this value is smaller than all the values in *array1* with higher than *index1*. Thus, *index2* is incremented.
- The value in *array1* at position *index1* is smaller than the one in *array2* at position *index2*. By the same reasoning as in the previous case, we can conclude that the value in *array1* at position *index1* is not present in *array2*. Thus, *index1* is incremented.

More about of the advantages of using sorted domains will be presented further in section *3.2*.

*values* is declared as an array of a generic type *U* extending the *Addable* interface. This interface contains only one method called *add* that is used to calculate the sum of two utility values. This method is used when joining Hypercubes.

*step_hashmaps* is a *HashMap* used to efficiently map a combination of variables values to a utility value. It maps a name of a variable (*String* object) to another *HashMap* which in its turn maps a value of this variable to a step (or shift) in the utility values array. Summing the step values corresponding to each value in a combination of variables values gives the index of the corresponding utility value for this combination. Let take for example the Hypercube represented in **Figure 2**. The *step_hashmaps* of this Hypercube is {(X1, {(0, 0), (1, 6)}), (X2, {(0, 0), (1, 2), (2, 4)}), (X3, {(0, 0), (1, 1)})}. Notice that the step corresponding to the first value is always zero for each variable. In addition, the difference between the step corresponding to the variable value and the step corresponding to the next value of the same variable is equivalent to the product of the sizes of the domains of all the variables that follow this variable in order. The *HashMap* avoids computing the utility equation each time the *getUtilityValue* method is invoked.

### *3.1.1 hypercube.Hypercube.NullHypercube*

The *NullHypercube* class implements a special Hypercube. It corresponds to a Hypercube with no variable, no domains, and no utility values. It is declared as a static class and contains a static object of type *NullHypercube* called *NULL*. The situations when *NULL* is obtained are as follows:

- Joining Hypercubes: if the intersection of the domains of the same variable in all the Hypercubes to join is an empty set.
- Projecting a Hypercube: if all the variables of the Hypercube are projected out.
- Split: if the provided threshold is bigger than all the utility values in the Hypercube.

The advantage of using a *NULL* Hypercube is to avoid testing in each operation if the Hypercube for which the operation is applied is empty or not. All the methods of the class *Hypercube* were re-implemented in *NullHypercube* to return the appropriate result for each method. So, when a Hypercube is joined with *NULL*, *NULL* is returned. The same happens with the projection, the slice and the split operations.

## 3.2 Implementation of operations

This section details how the different operations related to Hypercubes are implemented. Notice that for most of the operations multiple versions were implemented. Since, it is possible to obtain a more efficient implementation that treats special cases.

### 3.2.1 Join

Two join methods were implemented. The first one implements the operation of joining two Hypercubes and the second one implements the operation of joining a set of Hypercubes. Both methods have two parameters, the first parameter "*hypercube*" of the first method is a Hypercube to join with the one for which this method is called. The first parameter "*hypercubes*" of the second method is an array of Hypercubes to join with the current Hypercube. The second "*total_variables*" parameter of both methods is an array of variables names indicating the order that the variables should respect in every Hypercube.

The same principle is used to implement both operations. The methods start by building the array of variables of the resulting hypercube, and the array of their domains. We assume that variables respect the same order in all the Hypercubes to join and that the values inside a domain are ordered from the smallest to the largest to obtain a more efficient implementation of the join operation. Let us consider for example computing the union of two arrays such as v1 and v2 shown in **Figure 9**. The union is computed as follow:

- Three indexes, i1, i2 and i3, are used (two to traverse the arrays V1 and V2 and the third for the array *total_variables*). All of them are set to zero initially.
- If the entry in *total_variables* at i3 is equal to the entry in V1 at i1 then there are two possible cases:
  - If *total_variables[ i3 ]* is equal to V2[ i2 ], then in this case *total_variables[ i3 ]* is inserted in the resulting array and all the three indexes are incremented by one.
  - If *total_variables[ i3 ]* is not equal to V2[ i2 ], then in this case *V1[i1]* should be previous to *V2[i2]*. So, *total_variables[ i3 ]* is inserted in the resulting array and only i1 and i3 are incremented by one.
- If the entry in *total_variables* at i3 is not equal to the entry in v1 at i1 then there are two cases to consider :
  - If *total_variables[ i3 ]* is equal to V2[ i2 ], then in this case *total_variables[ i3 ]* is inserted in the resulting array and only i2 and i3 are incremented by one.
  - If *total_variables[ i3 ]* is not equal to V2[ i2 ], then in this case *V1[i1]* should be previous to *V2[i2]* and only i3 is incremented by one.

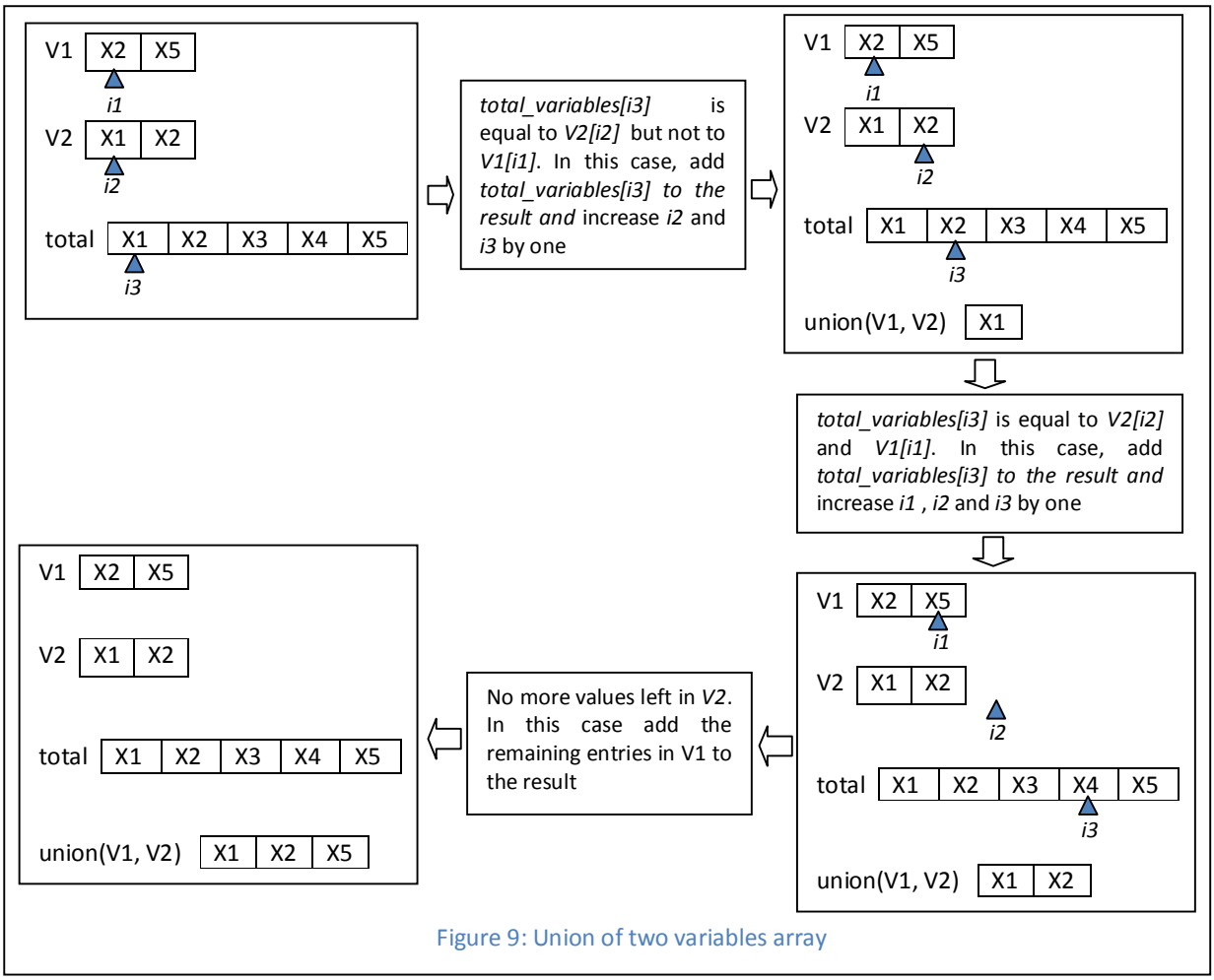



Figure 9: Union of two variables array

### 3.2.2 Projection

Two projection methods were implemented. The first method implements the operation of projecting out a provided list of variables. The second method implements a special case when the variables to project out are the last variables in the Hypercube (i.e. the variables with greater indexes). Both have three parameters.

The first parameter "*variables_names*" of the first method is an array of String containing the variables to project out of the Hypercube. The first parameter of the second method is an integer indicating the number of variables to project out starting from the last variable in the Hypercube.

The second parameter "*optimum_hypercube*" of the both methods is an empty Hypercube but not a null Hypercube. This Hypercube will be filled by the methods so that it will contain the same variables and domains as the returned Hypercube. On the other hand, the utility values array contains vectors of the projected out variables values that correspond to the maximum or minimum utility value (depending on the third variable). For the projection example in **Figure 4**, the resulting Hypercube will look like the one represented in **Figure 10**. The utility values array of this Hypercube has the same size as the one in the returned Hypercube with different entries. The first entry in the utility values array of the returned Hypercube

is 10. The first entry in the utility values array of the *optimum_hypercube* is {1, 0}. This indicates that in the projected Hypercube the utility value 10 corresponds to *X1* = 1 and *X2* = 0.

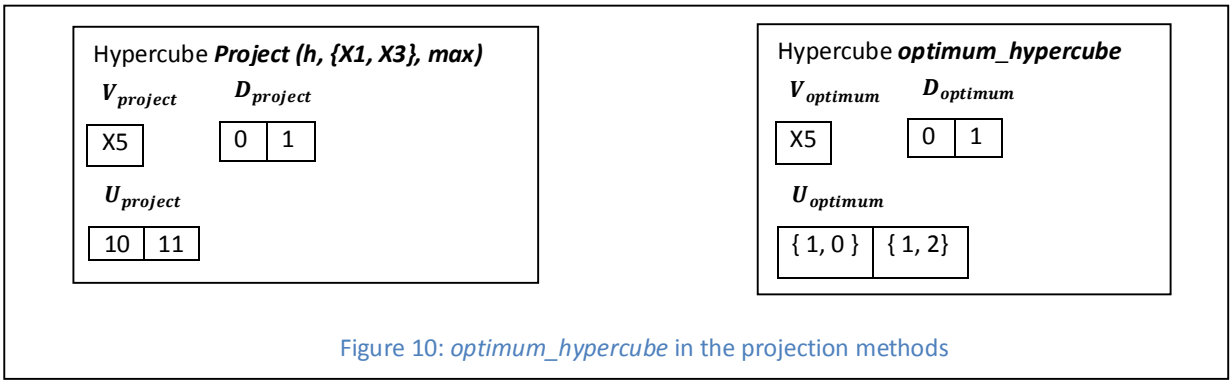



Figure 10: *optimum_hypercube* in the projection methods

The third parameter "*maximum*" is also common to both methods. It indicates whether to take the maximum or the minimum when computing the utility values of the resulting Hypercube.

Computing the array of variables and their domains is easy and straightforward. The array of variables of the resulting Hypercube contains the variables that are not projected out. Furthermore, the array of domains contains their corresponding domains unchanged. Constructing the utility values array of the resulting Hypercube and the *optimum_hypercube* is also easy to do. Using the domains of the resulting Hypercube variables, the size of the utility values array of the resulting Hypercube is computed as the product of the sizes of all the domains. To fill this array two nested loops are used. The first one loops over all the possible combination of values of the variables to be kept (i.e. variables of the resulting Hypercube). The second loop loops over all possible combination of values of variables to project out. For every completed loop of the second loop, the method keeps the maximum (or minimum) utility value corresponding to the concatenation of the variables values of the first and the second loop. Since values in the domains are ordered, then the order of the utility values in original Hypercube and the resulting one is the same. Therfore, the method does not need to look for the appropriate position when placing a utility value in the resulting Hypercube.

### 3.2.3 Slice

Two slice methods were implemented.

The first method takes as parameters an array of variables names and another array containing their corresponding sub-domains.

The second method proposes another implementation of the slice operation which is efficient for the case when the variables to slice are from the end of the variables list in the Hypercube and the provided sub-domains are only composed of one value. This method receives as a parameter an array of variables values. Since the sliced variables are from the end of the list of the Hypercube variables, there is no need to pass the names of the variables as a parameter to this method.

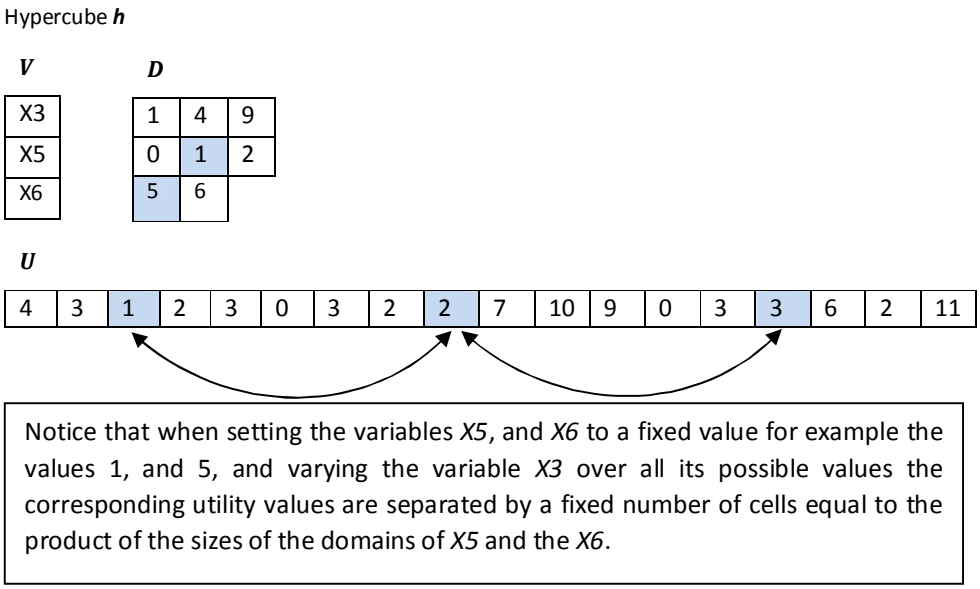



Figure 11: An example of a special case of the slice operation

**Figure 11** illustrates the principal of the slice method. It initially computes the index of the first utility value. This is done using the *step_hashMaps* by summing the step values corresponding to the provided values to the method). Having the index of the first utility value and the number of entries separating the utility values, the method extracts these utility values and inserts them in the utility values array of the resulting Hypercube.

### 3.2.4 Split

This method takes two parameters. The first parameter is a threshold and the second one is a Boolean indicating whether to keep utility values that are larger or smaller than the threshold.

The principal used by this method is simple. It goes through all the possible combination of variables values and each time, it checks if the corresponding utility value is larger or smaller (depending on the second parameter of the method) than the threshold. If the utility value is kept, the method adds the combination of variables values to the domain of the resulting Hypercube.

## 3.3 Save as XML

Before discussing in details of the structure of the XML file format used to represent a Hypercube, it is important to point out that the choice of the save format was based on two criterions.

- The format should be efficient in the sense that it shouldn't contain redundant information and the construction of a Hypercube from an XML file should not be expensive.
- The format should be human friendly in the sense that it should be easy for a human user to build XML files representing Hypercubes using a text editor.

Let us consider the Hypercube shown in **Figure 2**. The XML file representing this Hypercube is depicted in **Figure 12**. It consists of two parts.

The first part is represented by the *domains* element. It contains information about the name of the variables, their order and domains. Each variable of the Hypercube is represented by a *variable* element. This element has three attributes, *name*, *order* and type. Each *variable* element has its own *type* attribute, because variables may have different types. The content of the *variable* element represents the domain of the corresponding variable.

```
<?xml version="1.0" encoding="UTF-8"?>
<hypercube>
 <domains>
     <variable name="X0" type="java.lang.Integer" order="0">0 1</variable>
     <variable name="X1" type="java.lang.Integer" order="1">0 1 2</variable>
     <variable name="X2" type="java.lang.Integer" order="2">0 1</variable>
 </domains>
 <utility_values type="tests.MyInteger">
     <variable name="X0" value="0">
             <variable name="X1" value="0">0 1</variable>  </variable>
     <variable name="X0" value="0">
             <variable name="X1" value="1">2 3</variable>  </variable>
     <variable name="X0" value="0">
             <variable name="X1" value="2">4 5</variable>  </variable>
     <variable name="X0" value="1">
             <variable name="X1" value="0">6 7</variable>  </variable>
     <variable name="X0" value="1">
             <variable name="X1" value="1">8 9</variable>  </variable>
     <variable name="X0" value="1">
             <variable name="X1" value="2">10 11</variable>  </variable>
 </utility_values>
```

Figure 12: Hypercube XML representation example

The second part is represented by the *utility_values* element. This element contains information about how the variables values are related to the utility values. The *type* attribute of this element contains the type of the utility values contained in this Hypercube. As mentioned previously, a Hypercube contains a utility value for every combination of variables values. These combinations are represented using nested XML elements. An element (*variable* is the name used for this element) is used for every variable except the last one in order. This element has two attributes. The attribute *name* contains the name of the variable and the attribute *value* contains its value in the represented combination of values. The element corresponding to a variable is inserted within the elements corresponding to the variable which is previous to this variable in order. The last element contains a String representation of the utility values set obtained by varying the last variable over its domain. This is why the last variable is not represented by an element. Furthermore, grouping the utility values helps in obtaining a relatively compact representation of the utility values while remaining human friendly.

# 4 Tests and results

The package *test* contains the *JUnit* test cases used in this project. The class *HypercubeTest* is the *JUnit* test suite used to test the correctness of the implementations of the Hypercube operations on the Hypercubes. This class contains a method for every operation to be tested. It also contains the method that generates the random Hypercubes to be used in these tests.

## 4.1 Random Hypercubes generator

The method *randomHypercube* is used by the testing methods to generate random Hypercubes using the Class *Integer* as the type of the variables and the Class *MyInteger* as the type of the utility values.

The method works as follows:

- It starts by choosing a random number between 2 and 6 as the number of variables. Then, it constructs the arrays *variables_names* and *variables_domains* containing the names and the domains of the variables respectively.
- The method then fills the array *variables_names* with names composed of the letter "*X*" followed by a number. The variables are ordered according to this number. The method initially selects a random integer that will be used in the name of the first variable of the Hypercube. Then it just increases this number by one for each variable.
- While filling the array of variables names the method also fills the array *variables_domains*. The size of a variable's domain is chosen randomly between 1 and 3. Then, this domain is filled with integers starting from 0 and incremented by 1 each time.
- The product of the domains' sizes gives the number of utility values that the Hypercube must contain.
- Finally, the array *utility_values* is constructed and filled with integer values from 0 to the number of utility values – 1.

## 4.2 Join

The method *testJoinRandom* is responsible for testing the two implementations of the *join* operation. This method tests if the Hypercube resulting from joining a set of Hypercubes is the same as the one obtained from joining the Hypercubes of this set two by two.

First, this method builds a set of random Hypercubes using the *randomHypercube* method. Then, it joins them in three different ways and checks that the result obtained at the end is always the same.

The first way is to join the Hypercubes using one call of the join method that takes as a parameter an array of Hypercubes.

The second way is to join the Hypercubes by joining them two by two starting from the first Hypercube in the set of Hypercube using the method that takes as a parameter one Hypercube.

The third way is similar to the second one but the order is reversed (i.e. start from the last Hypercube in the set).

## 4.3 Projection

The method testing the projection implementation is *testRandomProject*.

This method tests if the Hypercube obtained by projecting out a random number of variable randomly selected from two randomly generated Hypercubes and then joining them is the same as the one obtained by first joining them and then projecting out the same set of variables.

It was observed that this is not always the case. Let us take for example the two Hypercubes, *h1 and h2*, in **Figure 13**. When first joining the Hypercubes and then projecting out the variables, the resulting Hypercube is totally different from the one obtained by first projecting out the variables and then joining the Hypercubes.

The solution adopted to avoid this special case is to make sure that the projected out variables are not common to both Hypercubes.

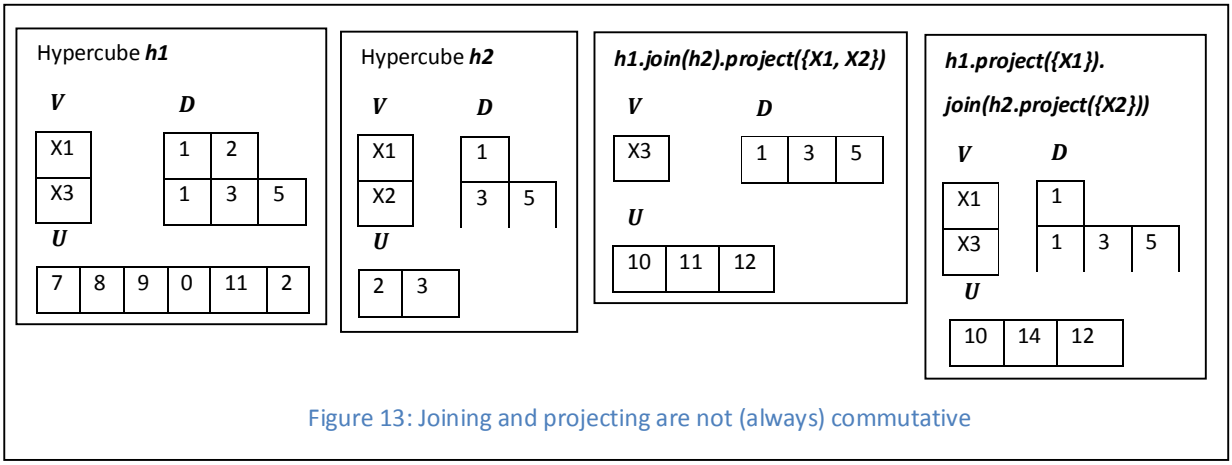



Figure 13: Joining and projecting are not (always) commutative

## 4.4 Slice

The method *testSliceRandom* tests the slice implementation by checking that the Hypercube obtained by joining two Hypercubes and then taking a slice of the resulting Hypercube is the same as the one obtained by first taking slices of the two Hypercubes and then joining the results.

During the tests, it was observed that the join and slice operations are not always commutative and that in some cases, reversing the order of the operation does not even give a result. An example of a problematic case is shown in **Figure 14**. The Hypercube obtained when slicing the Hypercubes *h1* and *h2* then joining the results is a *NullHypercube*. This is due the fact that the join method returns a *NullHypercube* when a common variable to the two Hypercubes have two completely different domains in the two Hypercubes and in the example the domain of the variable *X1* in the Hypercube *h1.slice({X1}, {2})* has no common values with the same variable in Hypercube *h2.slice({X2}, {3})*. When first joining the two Hypercubes and then slicing, {2} is no more a sub-domain of the variable *X1* leading to an error during the execution.

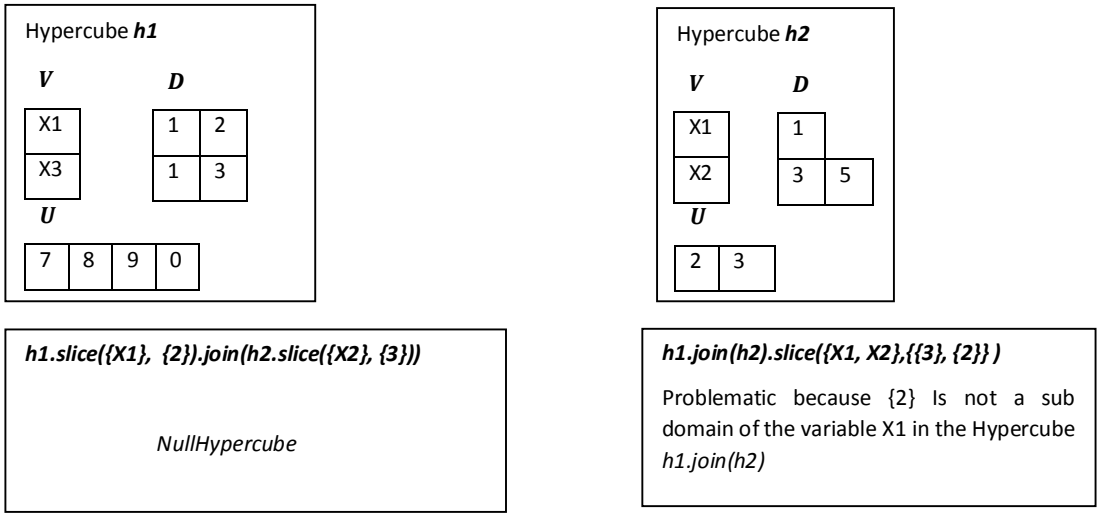



Figure 14: Joining and slicing are not always commutative

To avoid problematic cases, the method uses only common variables to the two Hypercubes and common values to these variables.

## 4.5 Split

The method *testSplitRandom* checks that the implementations of the split function works properly. The method generates a random Hypercube using the *randomHypercube* method. Then, it checks that the Hypercube obtained by projecting out a set of randomly selected variables out of the Hypercube a then splitting the result using half of the maximum utility value of the Hypercube as a threshold is the same as the one obtained by splitting the Hypercube using the same threshold and then projecting out the same variables.

# 5 Utility Diagrams

## 5.1 Definition

In the first part of the report, Hypercubes are presented as a simple approach to represent multi-valued functions. The assumption was that the Hypercube must contain a utility value for every combination of possible values of variables. This makes Hypercubes an inefficient method to represent sparse multi-valued functions. For example, the multi-valued function shown in **Figure 15** cannot be represented using Hyperubes. In order to overcome this problem, utility diagrams are considered.

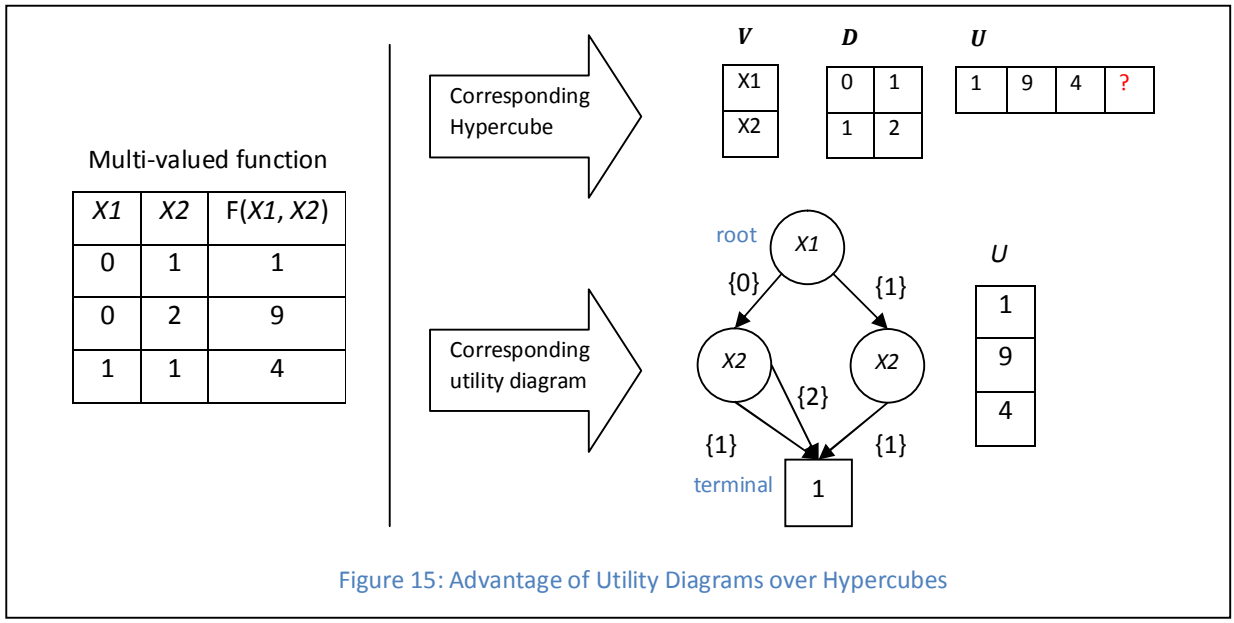



Figure 15: Advantage of Utility Diagrams over Hypercubes

Like MDDs, Utility Diagrams is an approach to represent multi-valued functions which is based on CDD. It uses a diagram and an array of utility values:

- The diagram is used to represent all possible combinations of variables values in the definition domain of the multi-valued function. A diagram is composed of nodes and edges. Nodes are represented by circles and labeled by the name of their corresponding variable except the terminal node which is represented by square and labeled by "1". Edges are represented by arrows pointing towards a destination node and labeled with the set of values they represent.
- The array of utility values contains the values of the function, also called "utility values". It contains an entry for every path in the diagram that starts at the root node and ends at the terminal node. Path numbers are generated according to a depth-first traversal of the diagram, assuming an order on the edges coming out of any node.

## 5.2 Utility diagrams reduction

The main advantage of using Utility diagrams over Hybercubes is the possibility of reducing its size. There are two types of reduction: nodes reduction and edges reduction.

### 5.2.1 Nodes reduction

This reduction consists in reducing the number of nodes by replacing a set of nodes by only one node. Two conditions are required to merge a set of nodes:

- The nodes must correspond to the same variable.
- The nodes must be roots of similar sub diagrams.

Let us consider the utility diagrams represented in the left part of **Figure 16** as an example. The sub diagrams rooted at the two nodes represented by the two dashed circles. The utility diagram obtained by merging these two nodes is on the right of **Figure 16**. Notice that the array of utility values does not influence the decision of whether nodes can be merged or not. Another thing that should be noticed is that merging two nodes does not induce any changes on the array of utility values.

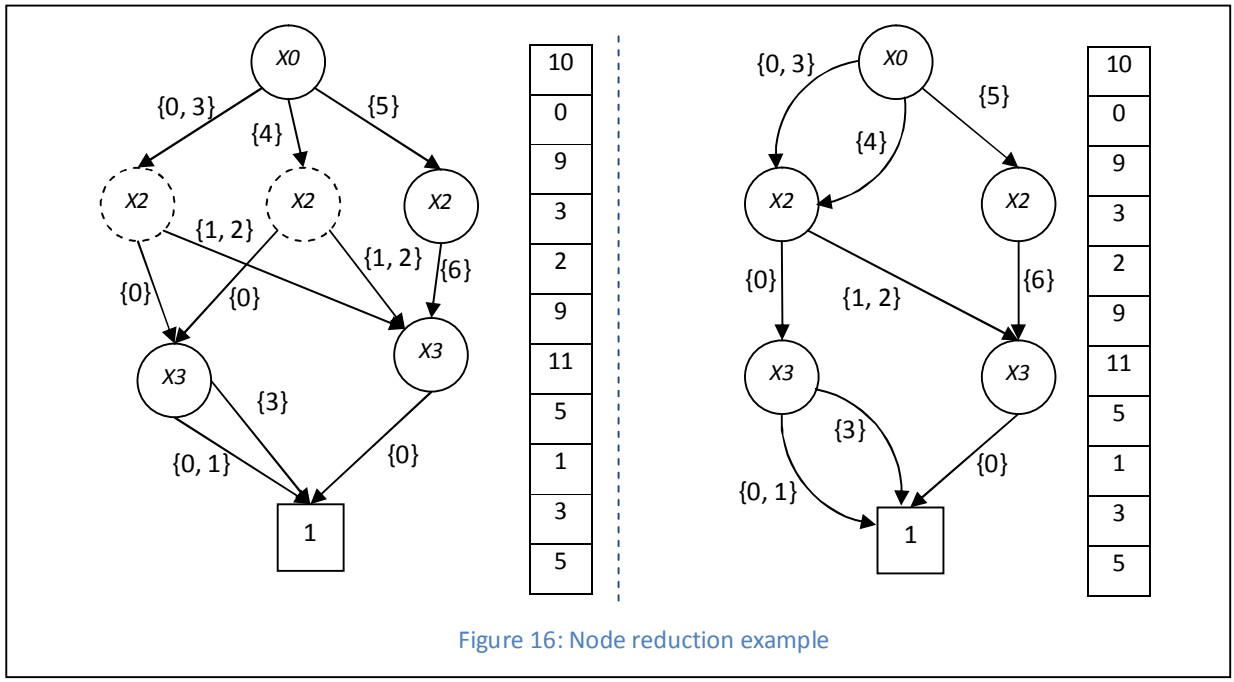



Figure 16: Node reduction example

### 5.2.2 Edge reduction

To be able to merge two edges, two conditions must be satisfied.

- The edges must have the same source and destination.
- The utility value corresponding to a path containing one the edges is equal to the utility value corresponding to the path obtained by replacing this edge by the other one.

Merging two edges consists of replacing the two edges by one edge having as domain the union of the domains of the two edges.

**Figure 17** shows an example of when edges are merged and the effect of such operation on the size of the utility diagram. Let us consider the edges $e_1$ and $e_2$. Both edges satisfy the first condition. The utility diagram has two paths involving edge $e_1$. The utility values corresponding to these two paths are 10 and 0 which are the same utility values obtained if $e_1$ is replaced by $e_2$ in these paths. This means that the second condition is satisfied as well and therefore edges $e_1$ and $e_2$ can be merged. Now Let us have a look into the edges $e_3$ and $e_4$ in the same utility diagram. Obviously these two edges satisfy the first condition. But the

utility values corresponding to paths involving edge $e_3$ (10, 10 and 0) are different from the utility values corresponding to those same paths but with edge $e_4$ replacing edge $e_3$ (0, 0 and 2). Thus, these two edges cannot be merged.

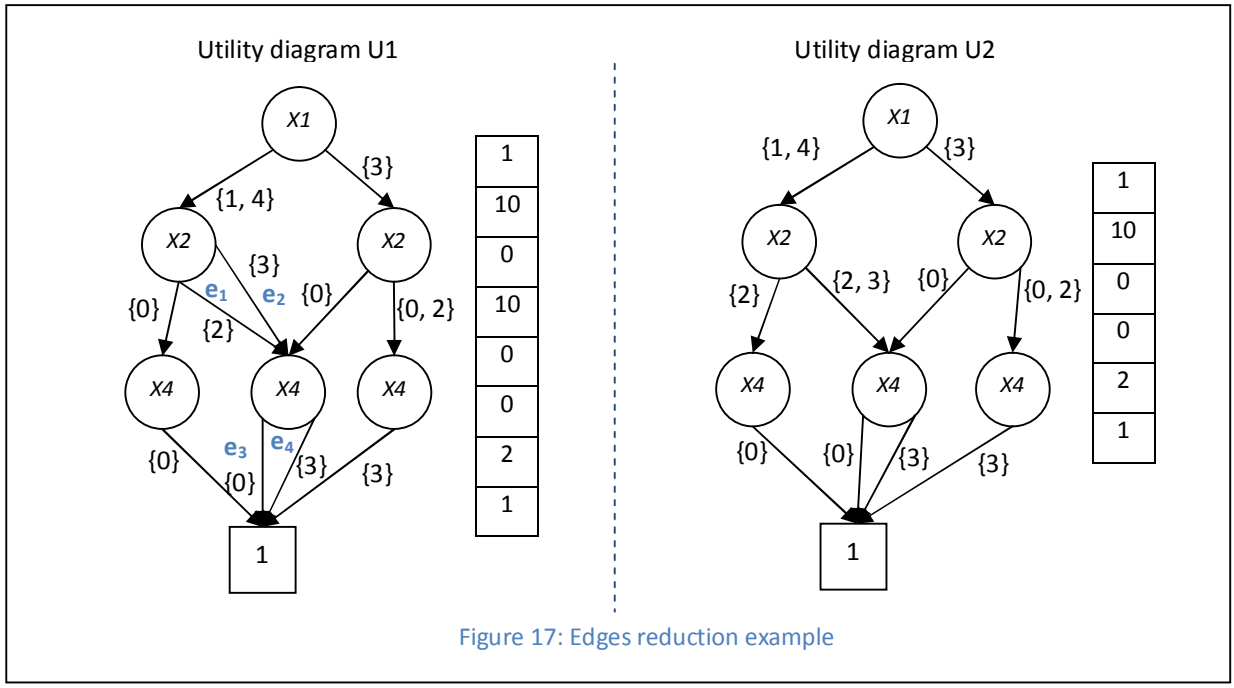



Figure 17: Edges reduction example

## 5.3 Operations on the Utility Diagrams

In this section, the different operations applied to utility diagrams will be presented on a simple example.

### 5.3.1 Join

The mechanism of the join operation is as follows:

Let us consider two utility diagrams as shown in **Figure 18**: utility diagram U1, which contains X1 and X3, and utility diagram U2 that contains the variables X2 and X3.

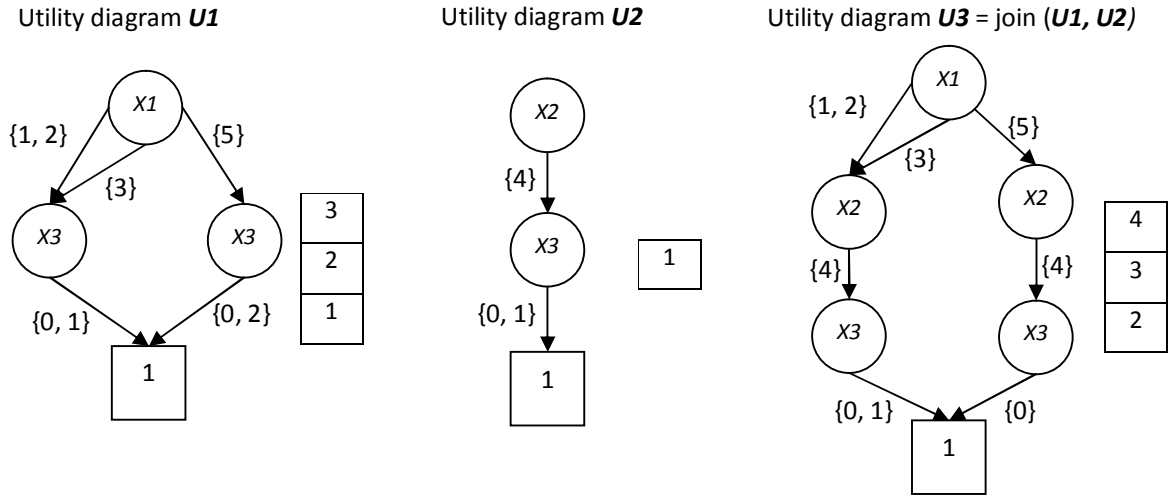



Figure 18: example of join operation

The join operation uses a pre-defined order of variables, in this example, it is X1, X2 and X3 in order. X1 as a root with its outgoing edges will form the root of the resulting utility diagram U3 since X1 is the first variable in order. Then, X2 follows, filling the second level of the resulting utility diagram. Since X3 is a common variable in both diagrams, U1 and U2, intersections between the domains, which are {0, 1} and {0, 2} in U1 and {0, 1} for U2, are computed. The outcome in this case is the domains, {0, 1} and {0}. The utility array of U3 is filled as follows: For every path p in U3, the corresponding utility values to p in U1 and in U2 are summed and the result is inserted in the utility array of U3.

### 5.3.2 Projection

Projecting a set of variables out of a utility diagram consists in reducing the number of utility diagram variables and keeping only the maximum (or minimum) utility value among the set of utilities corresponding to the same path in the resulting utility diagram. For example, let us consider the utility diagram *U* in **Figure 19**. By projecting the variable *X2* out of this utility diagram, we first obtain the utility diagrams in grey. After reduction, we obtain the utility diagram at the right of **Figure 19**. For paths such as the one formed by the two edges labeled $e_1$ and $e_2$ (i.e. *X0* = 0 or 3 and *X3* = 0 or 1), it is easy to find their corresponding utility value in the resulting utility diagram since they correspond only to one utility value in the original utility diagram. For paths that correspond to multiple utility values in the original utility diagram, only the maximum utility value is kept in the resulting utility diagram. An example of such a path is the one formed by the two edges $e_3$ and $e_4$ (i.e. *X0* = 1 and *X3* = 0). In the original utility diagram, *X0* = 1 and *X3* = 0 correspond to the utility values 3 (X2 = 0) and 9 (X2 = 1 or 2). So, the utility value corresponding to this path will be 9 in the resulting utility diagram.

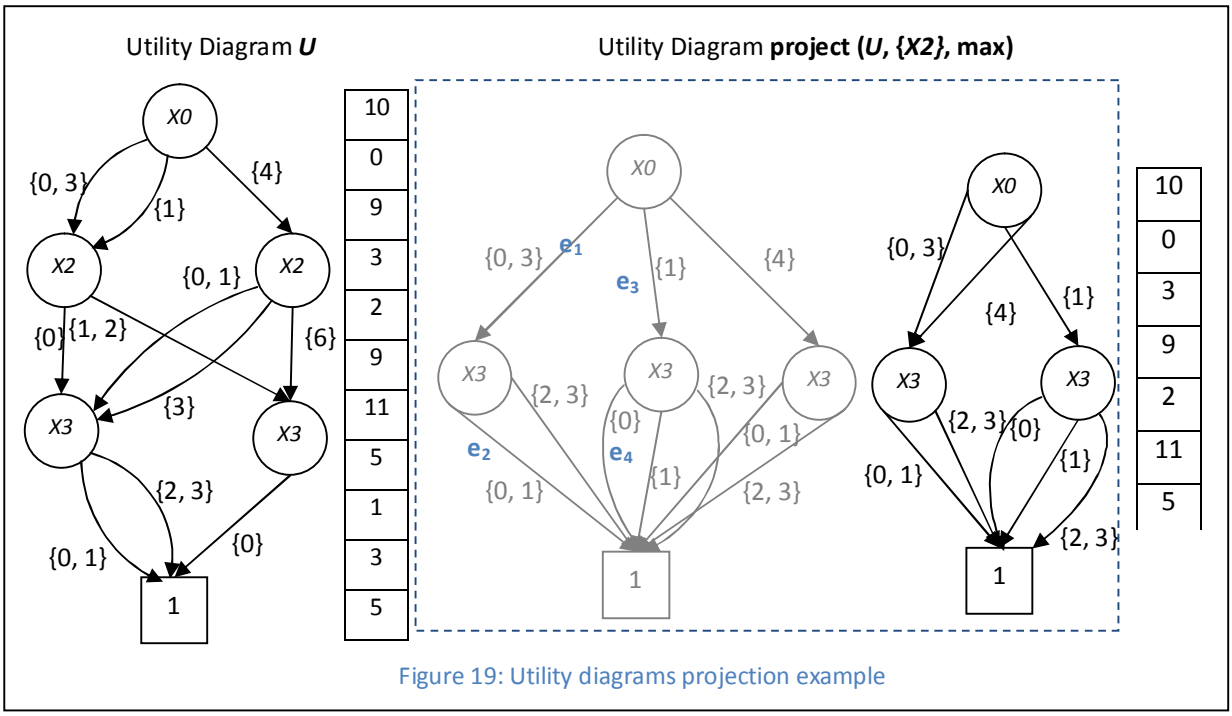



Figure 19: Utility diagrams projection example

### 5.3.3 Slice

In utility diagrams, the domain of a variable depends on the domains of a number of other variables in the utility diagram (the domains of the variables previous in order to this variable). Slicing the domain of a variable requires slicing the domains (i.e. the union of all the domains corresponding to the edges rooted at that node) of all the nodes corresponding to that variable according to a specified sub-domain. If the domain of the node does not contain any common value with the sub-domain, all the paths from the root of the diagram to that node are removed from the utility diagram (i.e. all the edges and nodes forming those paths except node that are part of other paths). Otherwise, all the outgoing edges at the node are removed except those that have common values with the sub-domain. But their domains are modified to contain only values that are contained in the sub-domain.

**Figure 20** illustrates an example of this operation. Slicing the utility diagram *UD* using {1, 2} as the new domains of *X2*, we first obtain the utility diagram represented in grey. Notice that the domains of the edges rooted at nodes corresponding to the variable *X2* are modified to keep only values in the set {1, 2}. Note that the edge e root at $n_B$ is removed from the utility diagram. This edge is removed because its domain does not contain any common value with sub domain of X2. Since this edge is removed from the utility diagram, the utility values corresponding to paths involving this edge are removed from the array of utility values. It is then possible to reduce this utility diagram by merging the two nodes corresponding to variable *X2* to obtain the diagram at the right side of the figure.

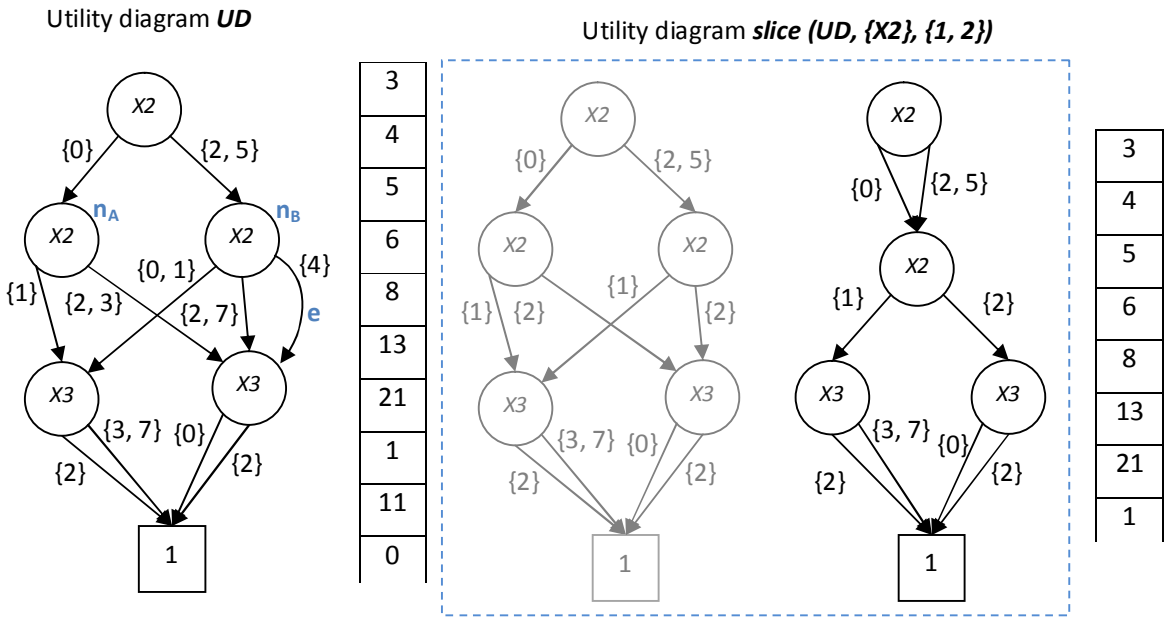



Figure 20: Slice operation example

### 5.3.4 Split

The operation of splitting a utility diagram consists in keeping only paths corresponding to utility values that are bigger (or smaller) than a certain threshold.

An example of this operation is illustrated in **Figure 21**. A slice is created from the utility diagram *U* using 6 as a threshold. Only two utility values of the utility diagram are smaller than the threshold. By removing these two paths from the utility diagram *U*, we obtain the utility diagram which is at the right side of the Figure 21. Notice that by removing edges from the utility diagram, we may need to split nodes that were merged in the original utility diagram. The node corresponding to the variable X3 is an example. The original utility diagram contains only one node for the variables X3 but the resulting utility diagram contains two.

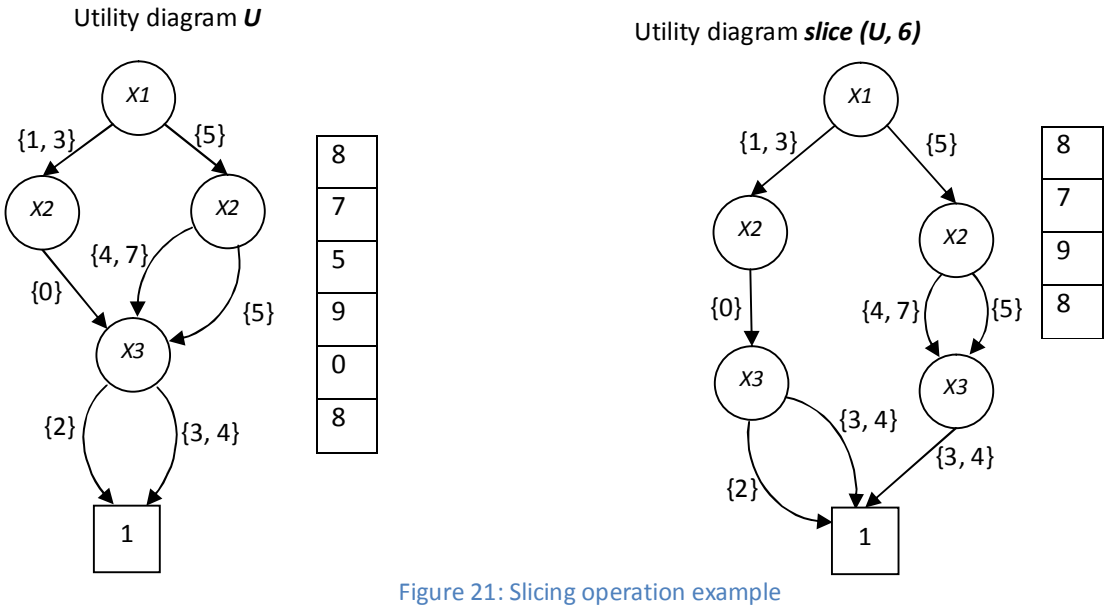



Figure 21: Slicing operation example

# 6 Utility Diagrams implementation

The *utilitydiagram* package contains all the classes concerning the utility diagrams implementation. This package contains three classes, *UtilityDiagram*, *Node* and *Edge* as illustrated in **Figure 22**.

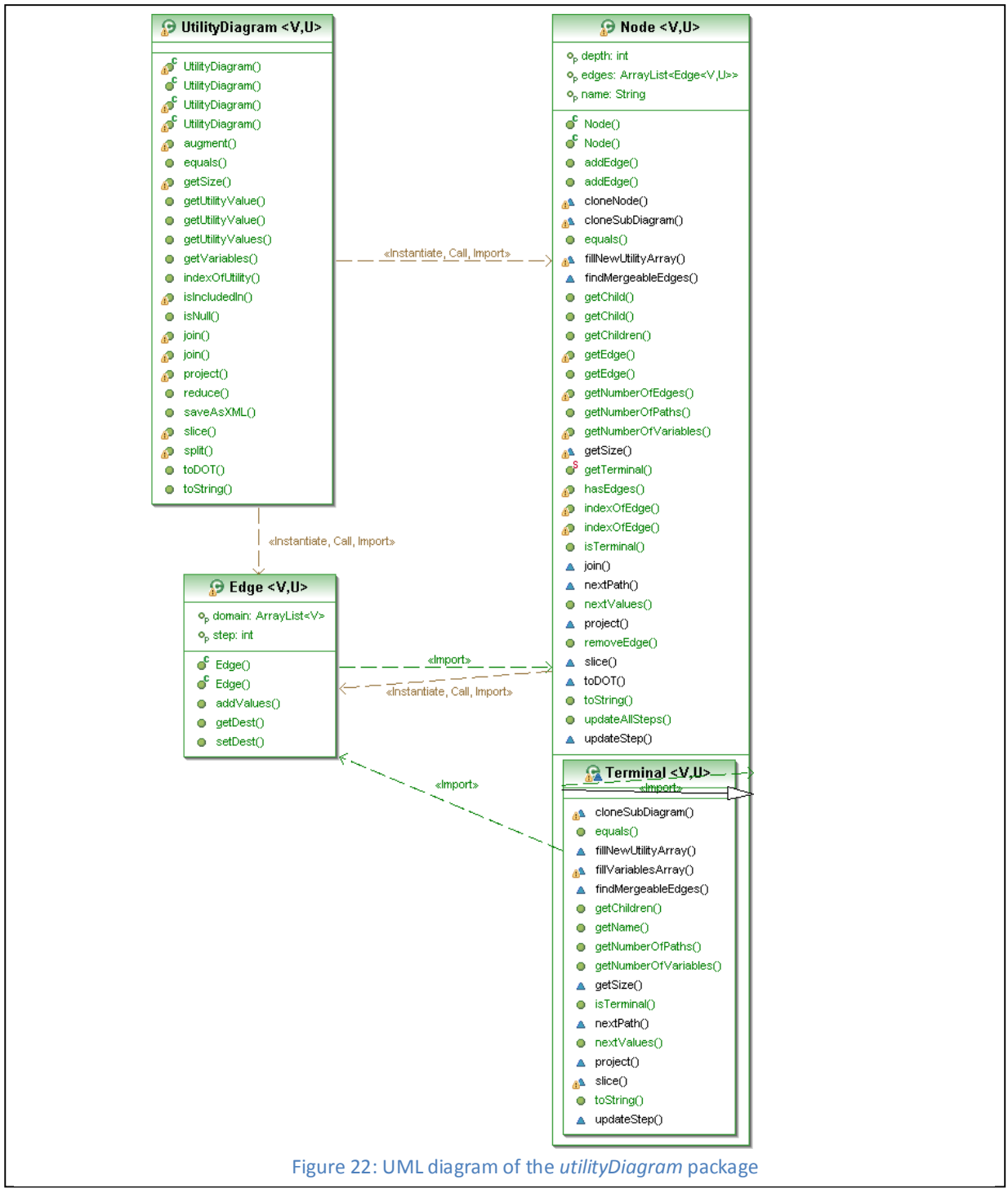



Figure 22: UML diagram of the *utilityDiagram* package

## 6.1 utilityDiagram.UtilityDiagram

This is the main class of the package. It represents a utility diagram. This class uses two parameters to represent a utility diagram. The first parameter is a *Node* object pointing to the root node of the utility diagram (having a reference to the root note is sufficient to access any node or edge of the utility diagram). The second parameter is an *Array* containing the utility values of the utility diagram.

### 6.1.1 utilityDiagram.UtilityDiagram.NullUtilityDiagram

This class plays the same role as the *NullHypercube* in the Hypercube implementation. It extends the UtilityDiagram class. It is used to represent an empty utility diagram with no diagram and no utility values. All the methods in the *UtilityDiagram* are re-implemented in this class. This allows treating this special utility diagram separately. This helps making the implementation more efficient in the sense that there is no need to check that the utility diagram for which a method is called is null. The situations in which a *NullUtilityDiagram* is return are:

- When joining two utility diagrams and one of them is NULL.
- When projection out of a utility diagram all its variables.
- When splitting a utility diagram with a threshold which is greater than the largest utility value in the utility diagram.

## 6.2 utilityDiagram.Node

The class represents a node. It contains an *ArrayList* of *Edge* objects representing the edges rooted at this node. Edges are increasingly ordered inside the *ArrayList* according to the first value of their domains.

## 6.3 utilityDiagram.Edge

This class represents an edge that uses three parameters. The parameter *domain* is an *ArrayList* containing the values corresponding to this edge. As for Hypercubes, a generic type is used to define the type of values inside the domains. The second parameter *dst* is a Node object used as a reference to the destination of this edge. The last parameter *step* is an integer representing the step value associated with this edge.

The *step* parameter is useful to compute the utility value corresponding to a path. By summing up the *step*s parameters of the edges forming the path, the index of the corresponding utility value is obtained. The value of the *step* parameter of an edge can be computed as the sum of two integer values:

- The first integer is the *step* value of the previous edge in the list of edges of the parent node.
- The second integer is the number of paths from the destination of the previous edge to the terminal node.

To illustrate this, in **Figure 23**, the step value for $e_2$ in this diagram is composed of first the integer 2, which represents the step value of the previous edge $e_1$, and the second integer which is the number of paths existing from the destination of the previous edge to the terminal node, which in this example two paths going through respectively $n_3$ and $n_4$.

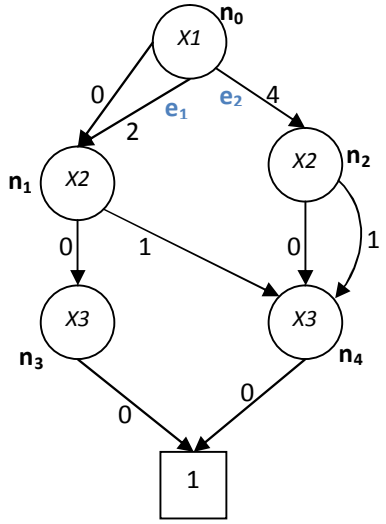



Figure 23: *step* values for edges [represented as attributes on edges]

The *step* value has also another useful utility. It is used to compute the number of paths between a node and the terminal node. Let us go back to **Figure 23**. Notice that by subtracting the step value of edge $e_1$ from $e_2$, we obtain the number of paths between node $n_1$ and the terminal node. The result of the subtraction can also be seen as the number of utility values corresponding to edge $e_1$. Therefore, this is a very efficient way of computing the number of utility values corresponding to an edge especially for large utility diagrams without having to count the number of paths between the destination node of this edge and the terminal node. We will see that this very useful in sections 6.4 and 6.5.

## 6.4 Reduction methods

Two methods were implemented for utility diagrams reduction. One for node reduction, called *reduceNodes*, and the other one is for edges reduction, called *reduceEdge*. In this section, explanations about how these methods were implemented are given.

### 6.4.1 *reduceNodes*

This method goes through the utility diagram level by level starting the first level (containing the root) to the last level (containing the terminal). At each level, the method tries to find nodes that are roots for identical sub-diagrams. If the method finds any, it removes them from the diagram and keeps only one. Then, it changes the destination node (*dst* parameter of an *Edge* object) of all the edges pointing to the removed nodes to the only one that was kept.

### 6.4.2 *reduceEdges*

The *reduceEdges* method implements the edges reduction operation. It traverses the diagram and for each node it does the following: It searches the list of edges for edges having the same destination. This means that these edges verify the first condition required to merge edges, which is that the edges must have the same source and destination. Further, in order to verify the second condition, it compares the utility values corresponding to each edge and if they are identical, it merges them.

## 6.5 Implementation of operations

### 6.5.1 Join

The class *UtilityDiagram* contains two join methods. Both methods have a common parameter which is a utility diagram to join with the current one. However, the first method has also a second parameter called *variables_order* which is an *Array* indicating the order that variables in the utility diagrams should respect. The second method does not have this parameter, but it assumes that common variables in the two utility diagrams have the same order (i.e. if the two utility diagrams contain variables *X3* and *X6*, and *X6* follows *X3* in order in one of the utility diagrams, this ordering should be also respected in the second utility diagram). Therefore, based on this assumption the second method can construct a *variables_order* array that indicates an order which is respected by the two utility diagrams and then it can use the first method to join the two utility diagrams.

The first method performs the join operation in two parts. Initially, it constructs the diagram of the resulting utility diagram using the *join* method in the *Node* class which traverses the two input diagrams in parallel in order to build the resulting diagram. **Algorithm 1** details how this method works. The *already_computed* parameter is a *HashMap* which is used to store already computed results. Re-using already computed results allows increasing the efficiency in terms of execution speed and size. The method first checks if there is no result corresponding to nodes node1 and node2. If a result is found, the method returns it. Otherwise, three cases are considered:

- One of the nodes, *node1* or *node2*, is a terminal (lines *04 - 05*).
- Both *node1* and *node2* correspond to the same variable (lines *06 – 17*).
- One of the nodes corresponds to the variable which has a lower order than the variable corresponding to the other node (lines 18 -28).

The second part consists in building the utility values array of the resulting utility diagram. For every path in the latter, the method sums the utility values corresponding to this path in the utility diagrams to join and inserts the result in the utility values array of the resulting utility diagram.

**Algorithm 1** Node ***join***( Node *node1*, Node *node2*, String *variables_order*, HashMap *already_computed*)

```
01: r ← the corresponding entry to node1 and node2 in already_computed
02: if( r is not null )
03:     return r

04: if( one of the nodes is a terminal )
05 :    the other node is returned as the result

06: if(node1 and node2 correspond to the same variable)
07:     r ← new node having the same name as node1, with no edges
08:     for every edge e1 of the outgoing edges of node1 and  e2 of the outgoing edges of node2 do
09:         d ← intersection of the domains of e1 and e2
10:         if (d is not null)
11:             dst ← join( destination of e1, destination of e2, already_computed )
12:             if (dst is not null)
13:                 add an edge to r having d as domain and dst as destination
14:     if ( r does not contain any edges )
15:         r ←  null
16:     add ( key built from the hash of node1 and node2, r) to the already_computed
17:     return  r
```

```
18: n1 ← the node (node1 or node2) corresponding to the variable with a lower order according to variables_order
19: n2 ← the node (node1 or node2) corresponding to the variable with a higher order according to variables_order
20: r ← new node having the same name as n1, with no edges
21:     for every edge e of n1 do
22:         dst ← join( e.destination, n2, already_computed )
23:         if (dst is not null)
24:             add an edge to r having as domain, the domain of e and as destination dst
25: if ( r does not contain any edges )
26:     r ← null
27: add ( key built from the hash of node1 and node2, r) to the already_computed
28: return r
```

### 6.5.2 Projection

The projection method returns a new *UtilityDiagram* object obtained by projecting out a provided set of variables which is received as the first parameter. The other parameter of this method is a Boolean that indicates whether to keep the maximum or minimum utility values when projecting out variables.

The idea used by this method is illustrated by **Algorithm 2**. For every possible combination of variables values in the original utility diagram, it does the following: It removes the values relative to the projected out variables and inserts the remaining values into the new utility diagram and stores its corresponding utility value in an *ArrayList*. If the method discovers that the new utility diagram contains a path corresponding to these values, it compares the new utility value with the old one stored in the *ArrayList*. The method replaces the old utility value in the *ArrayList* by the new one if the new value is greater (or smaller according to the second parameter of this method). When all the paths in the original utility diagram are processed, the method copies the content of the *ArrayList* in the utility array of the new utility diagram.

**Algorithm 2** *UtilityDiagram* ***project***(String[] *variables_names*, Boolean *maximum*)

```
01: r ← empty utility diagram
02: for every path p in the utility diagram do
03:     u ← utility value corresponding to p
03:     p' ← p after removing all the parts corresponding to the variables in variables_names
04:     if( r already contains p' )
05:         u' ← utility value corresponding to p' in r
05:         if ( (maximum and u > u') or( !maximum and u<u' ) )
07:             replace u' par u in r
08:     else  add p' and u' to r
09: return r
```

### 6.5.3 Slice

The slice method in the *UtiltyDiagram* class has two parameters. The first parameter is an array containing the names of the variables that will be sliced. The second parameter is another array containing the sub-domains of the provided variables.

This method uses the *slice* method in the *Node* class which looks through the current diagram and constructs the diagram of the resulting utility diagram. **Algorithm 3** details how this method works. For every node corresponding to a variable that must be sliced, the method filters all its edges and keeps only those having a domain that has common values with the provided sub-domain of the variable. If no edges are left

after the filtering process, the method also removes all the paths from the root node to the current node from the diagram.

Once the diagram is constructed, the principal method builds a utility values array from the utility values corresponding to the paths forming the diagram of the resulting utility diagram.

**Algorithm 3** *Node* ***slice***( Node *node*, String[] *variables_names*, V[][] *sub_domains*) where V is the variables type

```
01: if( node is the terminal )
02:     return terminal

03: for i ← 0 to variables_names.length do
04:     if ( node.name == variables_names[i] )
05:         r ← new node having the same name as node
06:         for every edge e of node do
07:             d ← intersection of the domain of e and sub_domain[i]
08:             if( d is not empty )
09:                 dst ← slice(destination of e, variables_names, sub_domains)
10:                 if( dst is not null )
11:                     add an edge to r having d as domain and dst as destination
12:         if( r has at least one edge )
13:             return r
14:         else return null

15: r ← new node having the same name as node
16: for every edge e of node do
17:     dst ← slice(destination of e, variables_names, sub_domains)
18:     if( dst is not null )
19:         add an edge to r having the same domain as e and dst as destination
20: if( r has at least one edge )
21:     return r
22: else return null
```

### 6.5.4 Split

The split method has two parameters. The first parameter is a threshold and the second one is a Boolean indicating to the method whether to keep values that are greater or smaller than the threshold.

The idea used by this method is detailed in **Algorithm 4**. The method builds the new utility diagram by merging the paths corresponding to the utility values that are greater (or smaller) than the threshold.

**Algorithm 4** *UtilityDiagram* ***split***(U threshold, Boolean maximum) where U is the utility values type

```
01: r ← an empty utility diagram
02: for every path p in  the utility diagram do
03:         u ← is the utility value corresponding to p
04          if ( (maximum and u > threshold) or( !maximum and u < threshold ) )
05:             insert p and u in r
06: if (r is empty)
07:     return NULL
08: else
09:     return r
```

## 6.6 Saving as XML

As for Hypercubes, our purpose was to define an XML file format which is efficient and human friendly at the same time. An example is shown in **Figure 24**. It represents the utility diagram in **Figure 14**. The XML file is divided into two parts. The first part is represented by the *diagram* element which contains information allowing the construction of the utility diagram graph. The *diagram* has two types of children elements:

- The *variable* element used to represent the variables of the utility diagram. There is a *variable* element for every variable of the utility diagram. This element has two attributes. The attribute *name* which contains the name of the variable and the attribute *type* which contains a reference to the Java Class representing the variable type.
- The *node* element used to represent the nodes forming the utility diagram. There is a *node* element for every node of the utility diagram. Every node element has three attributes:
  - The *id* attribute containing an id associated with the represented node.
  - *The variable* attribute containing the variable name corresponding to this node.
  - The *depth* attribute containing the depth of the node.

  This element has *edge* elements as children. They are used to represent the outgoing edges of a node. Their content is a string representation of the domains of the edges they represent. An edge element has two attributes.
  - The *id* attribute containing an id associated with the represented edge (in this case the ids are used to save the order of the edges).
  - The *dst* attribute containing the id of the destination node of the represented edge.

The second part of the XML file is represented by the *utility_values* element. This element contains the utility values of the utility diagram and a representation of their corresponding path. It has *path* elements as children. A *path* element has one attribute, *utility*, which contains a *String* representation of the utility value. The value contained in the *path* element is a concatenation of the indexes of the edges forming the path. In the example in **Figure 24** the first *path element* contains the value "0 0" which indicates that the path represented is the one formed by the first edge of the root node and the first edge of the second node.

```
<?xml version="1.0" encoding="UTF-8"?>
<decision_diagram type="reduced utilities">
 <diagram>
        <variable name="X1" type="java.lang.Integer" />
         <variable name="X2" type="java.lang.Integer" />

        <node id="n1" variable="X1" depth="0">
                <edge id="0" dst="n2">0</edge>
                <edge id="1" dst="n3">1</edge>
        </node>

         <node id="n2" variable="X2" depth="1">
                <edge id="0" dst="terminal">1</edge>
                <edge id="1" dst="terminal">2</edge>
        </node>

        <node id="n3" variable="X2" depth="1">
                <edge id="0" dst="terminal">0</edge>
        </node>
 </diagram>

 <utility_values type="tests.MyInteger">
        <path utility="1">0 0</path>
        <path utility="9">0 1</path>
        <path utility="4">1 0</path>
 </utility_values>
</decision_diagram>
```

Figure 24: XML format for utility diagrams

# 7 Tests and results

## 7.1 Correctness tests

The *UtiltyDiagramTest* is the *JUnit* test case used to test the implementation of utility diagrams. It tests each operation separately using the same test methodology. It is illustrated in **Figure 25**. First, it creates a random Hypercube and picks a random threshold. Based on these two objects, it creates a *UtilityDiagram* object (u in the figure) and a *Hypercube* object (h in the figure). Then, it applies the same operation to both objects. This gives result to a new *UtilityDiagram* and a new *Hypercube* objects. Finally, it checks that the resulting *UtilityDiagram* is included in the resulting *Hypercube*. This means that both objects have the variables in the same order and that every possible combination of variables values in the *UtiltyDiagram* is present in the *Hypercube* and in both objects this corresponds to the same utility value.

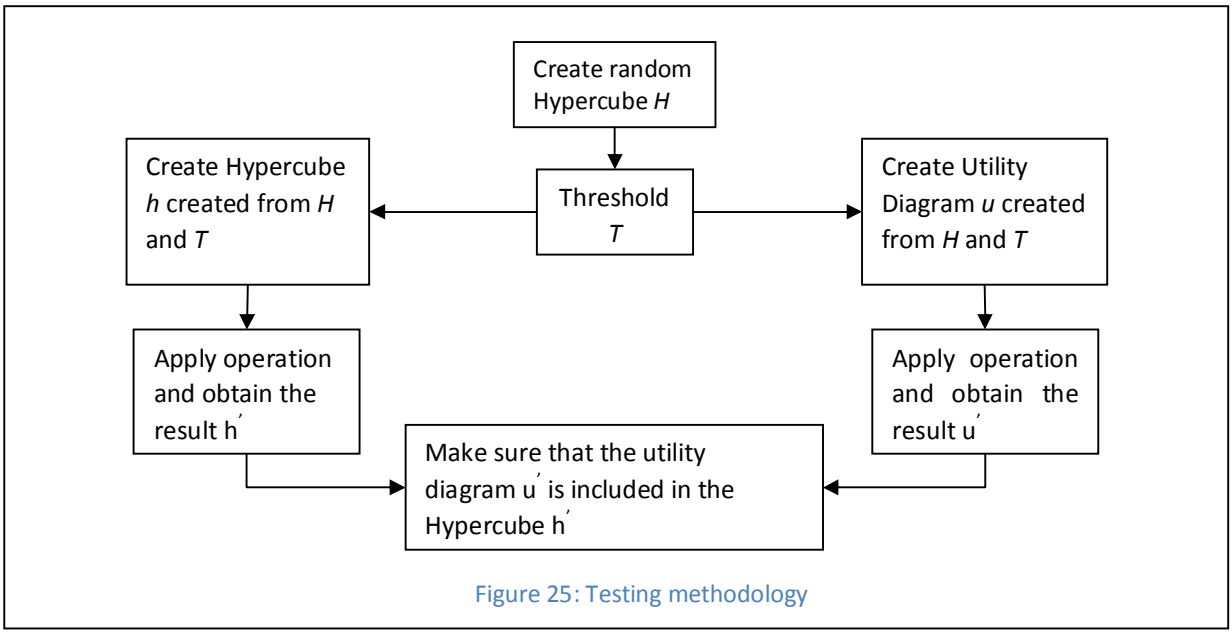



Figure 25: Testing methodology

## 7.2 Efficiency tests

After the tests of correctness, it was important to measure the efficiency of the utility diagrams implementation. So, comparison tests were performed. The proposed implementation of Utility Diagrams was compared with the MDD implementation. For testing purposes, another version of the random Hypercube generator was implemented. Further, a random utility diagram and a MDD are derived from the random Hypercube by discarding randomly some of its utility values.

### 7.2.1 Random Hypercube generator V2

The method generates a Hypercube based on three provided parameters. These parameters are:

- The number of variables of the Hypercube
- The size of the domains of the variables
- The redundancy in the utility values array (1 - 100).

The method generates a random Hypercube by the following steps:

- First, it creates the array of the variables of the Hypercube. Then, it fills it the same way as the previous version (the one used to test the Hypercube implementation) does it.
- Then, it creates the array of the domains such that every domain has the provided size. Filling the domains is also similar to the previous version of the generator.
- The array of utility values is created and then filled as in the previous generator but to create the required redundancy, only values between zero and "redundancy * utility values array size" are used.

### 7.2.2 Execution speed Comparison

These comparison tests consist in comparing the time required by each application to perform the same operation on the same multi valued function. Kumar's MDD implementation proposes only two operations. The first operation is a *projection* that allows projecting the last dimension (i.e. variables) out of a MDD. The second operation is a join operation that takes as parameters the two MDDs to join and a third parameter indicating which dimension should be the last dimension of the MDD resulting from the join operation.

To have a "fair" comparison, the chosen test scenario is as follow:

- First, generate two random Hypercubes, *hypercube1* and *hypercube2* using the same values for the generator parameters (number of variables, maximum domains size, and utility values redundancy).
- Then, two utility diagrams, *ud1* and *ud2*, are derived from *hypercube1* and *hypercube2*. Their equivalent MDDs, *mdd1* and *mdd2*, are also created.
- After that, a random variable is selected from the variables of *ud1*. The equivalent dimension in *mdd1* of this variable is found using its index.
- The time *UD_time* needed to join *ud1* and *ud2*, and then project the randomly selected variable out of the resulting utility diagram is computed and saved in a file.
- The time *MDD_time* needed to join *mdd1* and *mdd2* and specify the dimension corresponding to the variable randomly selected as the third parameter of the join method, and then project the last dimension of the resulting MDD is computed and then saved in a file.

Many parameters such as the sparsity in a multi-valued function, the number of variables, the sizes of the domains, and the redundancy in the utility values influence the speed at which operations are performed. Because of the huge time required to run a simulation, the decision was made to focus only on one parameter which is the number of variables. The other parameters were fixed to the following values:

- Sparsity = 60%
- Size of the domains = 5
- Redundancy = 0%, which corresponds to the worst case for the utility diagram implementation.

By varying the number of variables between 3 and 7, the result shown in **Figure 26** are obtained.

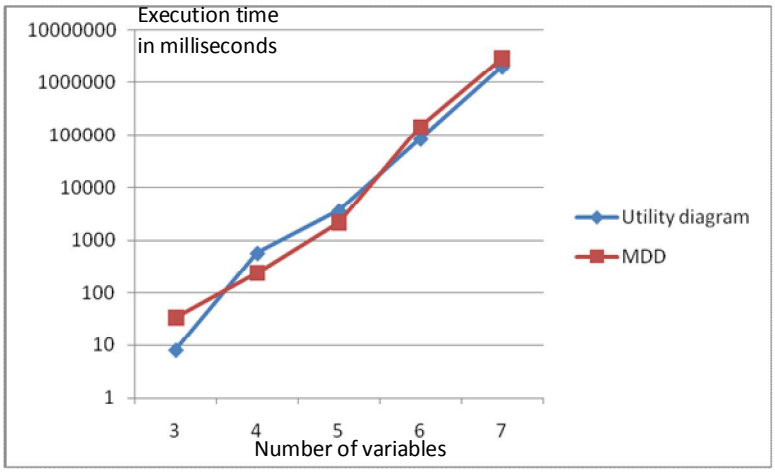



Figure 26: preliminary comparison results between MDD and utility diagrams

Comparing the two approaches, utility diagrams and MDDs, it is shown that as the number of variables increases, Kumar's MDDs requires more time than the utility diagrams to execute the same operation. As an example for number of variables equal to 7, the MDDs required 1.7 times the time required by the utility diagrams.

# 8 Conclusion

As a first part of this project, a new implementation of the Hypercubes data structure was developed offering more operations (splitting and re-ordering) than the one proposed by Kumar. In addition to these operations, a saving method was added, allowing representing Hypercubes in the XML file format. Test suites were also implemented to help further improve the implementation.

The second part of the project consisted in proposing and implementing a new approach for representing multi-valued functions which should be more efficient than the MDDs. Utility decision diagrams were proposed as a new approach. They were derived from CDDs. An implementation of this new approach was done offering more operations than the existing implementation of MDDs. Preliminary comparison tests were performed and the results showed that as the number of variables of a multi-valued function increases, the utility diagram becomes more efficient in terms of execution time.

Comparison tests performed at the end of this project showed the effect of varying the number of variables on the time required to execute operations. The number of variables is not the only parameter affecting the execution time, yet parameters such as the size of the variables domains, the redundancy, and scarcity of the multi-valued function, also have their effects. In fact, future work could aim at deeply analyzing the parameters in terms of their impact on the execution time and memory usage would add a great value to the comparison between the approaches, Utility Diagrams and Kumar's MDDs.

For Hypercubes, more operations could be added in the future such as adding rows to the Hypercube and removing rows from it. Another operation to add is a join operation that does not require the order of the variables being the same in the Hypercubes to join.

For the utility diagrams, an efficient re-ordering method could be added in the future. Also, as suggested for the Hypercubes, a join operation that does not impose any restrictions on the order of the variables could be added.

Finally, advanced versions of the operations implemented during this project which are capable of modifying the current Hypercube or utility diagram without the need of creating a new one, would indeed help in increasing the efficiency of the implementation in terms of memory usage.

# 9 References

**[1]** Adrian Petcu and Boi Faltings. A scalable method for multiagent constraint optimization. In *proceedings of the 19th International Joint Conference on Artificial Intelligence*. IJCAI-05, Edinburgh, Scontland, Aug 2005.

**[2]** Akshat Kumar, Adrian Petcu, and Boi Faltings. H-DPOP: Using hard constraints to prune the search space. *In Proceedings of the Eighth International Workshop on Distributed Constraint Reasoning*. IJCAI-DCR'07, Hyderabad, India, January 8 2007

**[3]** Tsutomu Sasao, Masahiro Fujita, Representation of discrete functions, ISBN: 9780792397205

**[4]** Kenil C. K. Cheng and Roland H.C Yap. Constraint decision diagrams. In proceedings of the Nation Conference on Artificial Intelligence, AAAI-05, pages 366-371, Pittsburg, USA, 2005

**[5]** http://www.w3.org/XML/.