Chapter 5
Determining the Configurations for the nD-OPP's ($n \geq 4$)

In this chapter is described the "Test-Box" Heuristic, presented in [Pérez, 03], that gives a solution to the problem of determining the configurations that can represent the n-Dimensional Orthogonal Pseudo-Polytopes (the configurations for the 2D, 3D and 4D-OPP's were presented in section 4.2). This heuristic presents a complexity that is minor compared to the exhaustive searching method. The heuristic has as one of its fundaments the extrusion of the (n-1)-dimensional configurations to obtain the n-dimensional configurations. Among the obtained results there are mentioned the configurations for the 5D and 6D-OPP's (section 5.3). Finally, in section 5.7, we will consider the differences between the procedures for obtaining the configurations for the 4D-OPP's according to the methods described by [Hill, 98] and Aguilera & Pérez in [Pérez, 01].

5.1 The Problem of Determining the Configurations for nD-OPP's (n>4)

For the Euclidean n-Dimensional space we have 2^n possible hyper-octants (4 quadrants for 2D space, 8 octants for 3D space, and 16 hyper-octants for 4D space). As observed in sections 4.2.1, 4.2.2 and 4.2.3 (configuration for 2D, 3D and 4D-OPP's respectively), the number of hyper-octants has a repercussion over the possible number of combinations of vertices described through the presence or absence of *hyper-boxes* each one in every hyper-octant. In general, we have that the total number of combinations in nD space is [Hill,98]:

$$2^{(2^n)}$$

It was before discussed that in 4D space we have 2^{16} = 65,536 combinations. [Pérez,01] determined that there are 253 configurations for 4D-OPP's through exhaustive searching. However, if we want to determine the configurations for 5D-OPP's through exhaustive searching, we would have to consider that there are 32 hyper-octants in 5D space, and for instance to analyze 2^{32} = 4,294,967,296 combinations [Hill,98].

TABLE 5.1
Comparing the number of configurations with the number of combinations for the nD-OPP's (Taken from [Pérez, 03]).

nD Space	**Combinations**	**Configurations**	**Percentage (Configurations Vs. Combinations)**
1D	4	3	75 %
2D	16	6	37.5 %
3D	256	22	8 %
4D	65,536	253	0.3 %
5D	4,294,967,296	?	<< 0.3 %

Moreover, if the number of configurations is associated with the total number of combinations, it is evident that the first one is much lesser than the second one. For example, in 3D space we have 22 configurations for 256 possible combinations, this can be translated as that only the 8% of the combinations can perform the role of representatives (equivalence classes) of the others. See **Table 5.1** for the application of this comparison over the configurations in 1D, 2D, 3D and 4D spaces.

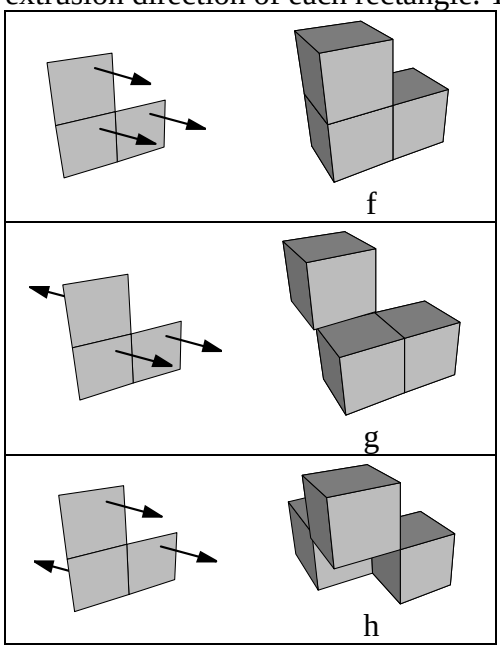
These situations lead us to conclude that the complexity imposed by the exhaustive searching makes difficult to determine the configurations for OPP's in spaces of 5 dimensions and beyond [Hill,98]. In the following section we will describe a heuristic for obtaining the configurations in a more direct way. The heuristic's first step is to obtain a subset of the nD configurations' final set through the extrusion of (n-1)D configurations.

5.2 The “Test-Box” Heuristic for Obtaining Configurations for nD-OPP’s

5.2.1 Extruding Configurations

The extrusion of a n-dimensional configuration implies that each one of its boxes will be extruded in a direction that is perpendicular to the space in which it is embedded. The extrusion of each box will describe a *hyper-box* (this process is analogous to obtaining the hypercube through the method proposed by Bragdon [Rucker,77], section 2.2.1.1). It is important to consider that an (n+1)-dimensional configuration obtained through the extrusion of a n-dimensional configuration is not unique, because there are two possible translation directions for each box. For example, in **Table 5.2** it is presented the extrusion of the 2D configuration e for obtaining 3D configurations f, g and h.

TABLE 5.2
Extrusion of 2D configuration "e" and the obtained 3D configurations
(the arrows indicate the extrusion direction of each rectangle. Taken from [Pérez, 03]).



Through extruding configurations it is possible to obtain some configurations from (n+1)-dimensional space by using the configurations from n-dimensional space which are obtained through (n-1)-dimensional configurations and so on. By this way, we obtain then a recursive process whose basic case are the configurations for 1D-OPP's (**Table 5.3**).

TABLE 5.3

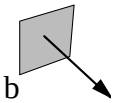
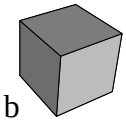
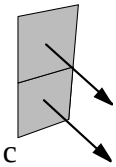
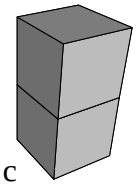
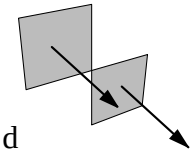
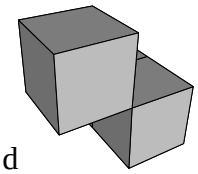
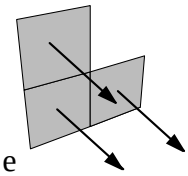
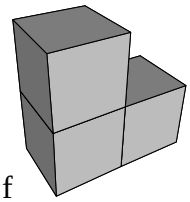
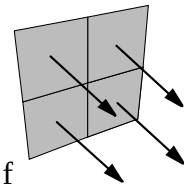
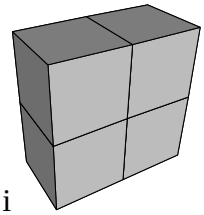
The three configurations for 1D-OPP's (taken from [Pérez, 03]).

a	 b	 c

5.2.2 Obtaining the Configurations Through a "Test-Box"

The "Test-Box" heuristic starts with the following principle: to have access to (n-1)-dimensional configurations for obtaining the n-dimensional configurations. Each (n-1)-dimensional configuration is extruded just one time and in just one direction, this means that, the boxes that compose it are extruded towards the same perpendicular direction from space in which they are embedded. Once this process is applied, the (n-1)-dimensional configuration is not required again. For example, five configurations for 2D-OPP's are extruded just one time and towards the same direction for obtaining five configurations for 3D-OPP's (**Table 5.4**).

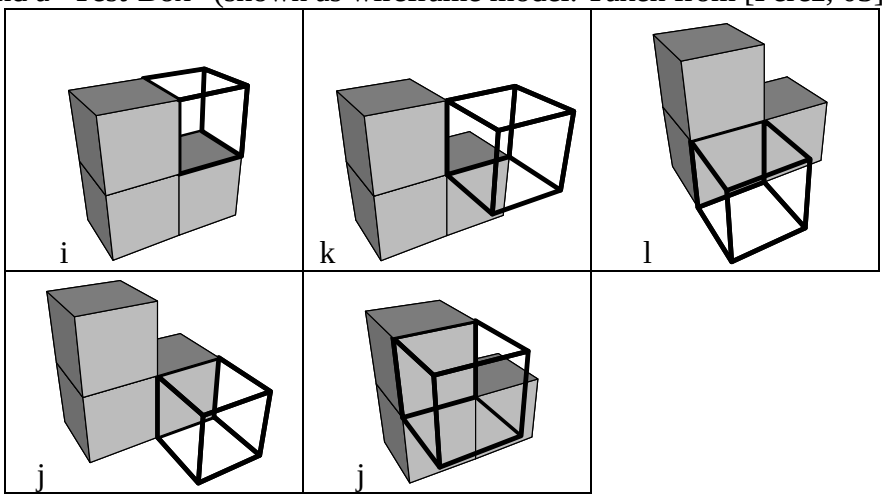
TABLE 5.4
Extruding 2D configurations in the same direction and
Obtaining their 3D analogous (Taken from [Pérez, 03]).

2D Configuration	Extrusion: 3D Configuration	2D Configuration	Extrusion: 3D Configuration
 b	 b	 c	 c
 d	 d	 e	 f
 f	 i		

Once the configurations from (n-1)-dimensional space have been extruded, we have now the same number of n-dimensional configurations. The next step is the use of each n-dimensional configuration for obtaining the remaining configurations. We will use a "Test-Box" (a rectangle, a cube, a hypercube, etc.). For each configuration, we will add it a "Test-Box" in one of its empty hyper-octants. This addition will produce a new combination which must be compared with the set of the configurations already identified, for determining whether a new configuration has been obtained or not. This process is repeated until all the configuration's empty hyper-octants have been evaluated with a "Test-Box". In **Table 5.5** are shown the 3D combinations obtained from the configuration f and by applying a "Test-Box" in all its empty octants.

TABLE 5.5

Obtaining new configurations through 3D configuration f
and a “Test-Box” (shown as wireframe model. Taken from [Pérez, 03]).



We have now the elements to propose an algorithm applying extrusions and a “Test-Box”. The algorithm is resumed in [Pérez, 03] with the following main procedures:

1. For a number n of dimensions we obtain the (n-1)-dimensional configurations. If n = 1 then we have the basic case which returns the configurations from **Table 5.3** (1D configurations).
2. The (n-1)-dimensional configurations are extruded in n-dimensional configurations.
3. To each n-dimensional configuration it is added a “Test-Box” in their empty hyper-octants, this operation will produce new combinations.
4. Each new produced combination will be evaluated with the set of already identified configurations. If it is a new configuration then it will be added to the set of identified configurations and considered to be evaluated with a “Test-Box”, because it could produce new configurations.

We present now the proposed algorithm [Pérez, 03]:

Input: The number of dimensions > 0 for the configurations to obtain.
Output: The set of configurations for the specified space.

```
getConfigurationsForSpaceUsingTestBox(dimensions)
{
        if(dimensions == 1)
                // Basic Case: just return the three configurations for 1D space.
                return getConfigurationsFor1DSpace( );
        else
        {
                /* Recursive call: the configurations from (n-1)D space are obtained and they are added
                   to the set 'previousConfigurations'. */
                previousConfigurations = getConfigurationsForSpaceUsingTestBox(dimensions - 1);
                For each configuration c in the set previousConfigurations
                {
                        /* Configuration 'c' is (n-1)D. The configuration 'newC' (n-dimensional) is the
                           result of extruding configuration 'c'. */
                        newC = extrudeConfiguration(c);
                        /* The configuration 'newC' is added to the set 'configurations' (the configurations
                           from current nD space). */
                        configurations.add(newC);
                }
                /* Starts the cycle for generating new combinations from the configurations contained in the
                   set 'configurations' using a "Test-Box" (rectangle, cube, hypercube,etc.) whose
                   position (hyper-octant to occupy) is indicated by variable 'testBoxPosition'. */
                hyperOctants = 2^dimensions;
                For each configuration c in the set configurations
                {
                        testBoxPosition = 0;
                        /* Starts the cycle for generating new combinations from configuration 'c' using a
                           "Test-Box". */
                        while(testBoxPosition < hyperOctants)
                        {
                                /* The combination 'newC' is obtained from configuration 'c' and the
                                   "Test-Box" added in the hyper-octant specified by 'testBoxPosition'. */
                                newC = getNewConfiguration(c, testBoxPosition);
                                /* It is verified if combination 'newC' was before obtained. If not, then it
                                   is added to set 'configurations' and for instance a new configuration
                                   has been found. */
                                if(configurations.isContained(newC) == false)
                                        configurations.add(newC);
                                testBoxPosition++;
                        }
                }
                /* All the possible configurations have been found. The set 'configurations' is returned as
                   output.
                return configurations;
        }
}
```

5.3 “Test-Box” Heuristic’s Results and Complexity

For determining the number of combinations analyzed to obtain the n-dimensional configurations through the “Test-Box” heuristic it is necessary to analyze the output’s size, i.e., the number of configurations. Since we will not know the number of configurations until the algorithm finishes, we have then an *output-sensitive* complexity analysis [deBerg,97].

Definition 5.1: *Let* **CTB** *(Configurations-by-Test-Box) be the number of configurations obtained by the algorithm and* 2^n *the number of hyper-octants for the nD space. Then the number of combinations to analyze is at most:*

$$CTB \cdot 2^n$$

This is an upper bound because we are considering that for each configuration (with 1, 2, 3, etc. *hyper-boxes*) there are 2^n empty hyper-octants (this is possible only for configurations with 0 *hyper-boxes*). We must consider, in fact, that configurations with 1 box have 2^n-1 empty hyper-octants, configurations with 2 boxes have 2^n-2 empty hyper-octants and so on.

Definition 5.2: *Let* $\mathbf{CTB_i}$ *be the number of those configurations with i boxes, then we have that the exact number of combinations to analyze in a nD space is:*

$$\sum_{i=0}^{2^n} CTB_i \cdot (2^n - i)$$

The algorithm in section 5.2.2 has confirmed the expected configurations for 2D, 3D [Aguilera,98] and 4D [Pérez,01] spaces. Specifically, the greatest number of combinations to analyze for obtaining the configurations in 4D space is 253 * 2^4 = 4,048.

Although this is an upper bound, it is much better than the obtained through exhaustive searching by [Pérez,01] (2^{16} = 65,536).

Through the "Test-Box" heuristic we have found 20,983 configurations for the 5D-OPP's [Pérez, 03] whose distribution is shown in **Table 5.6**.

TABLE 5.6

Configurations' distribution for 5D-OPP's (Taken from [Pérez, 03]).

Number of 5D hyper-boxes (i)	**CTB_i**	**Number of 5D hyper-boxes (i)**	**CTB_i**
0	1	32	1
1	1	31	1
2	5	30	5
3	10	29	10
4	38	28	38
5	66	27	66
6	164	26	164
7	236	25	236
8	454	24	454
9	570	23	570
10	887	22	887
11	989	21	989
12	1,388	20	1,388
13	1,406	19	1,406
14	1,754	18	1,754
15	1,607	17	1,607
16	1,831		

The precise number of analyzed 5D combinations is:

$$\sum_{i=0}^{2^5} CTB_i \cdot (2^5 - i) =$$

$$\left\{\begin{array}{l} 1\cdot 32 + 1\cdot 31 + 5\cdot 30 + 10\cdot 29 + 38\cdot 28 + 66\cdot 27 + 164\cdot 26 + \\ 236\cdot 25 + 454\cdot 24 + 570\cdot 23 + 887\cdot 22 + 989\cdot 21 + \\ 1{,}388\cdot 20 + 1{,}406\cdot 19 + 1{,}754\cdot 18 + 1{,}607\cdot 17 + 1{,}831\cdot 16 + \\ 1{,}607\cdot 15 + 1{,}754\cdot 14 + 1{,}406\cdot 13 + 1{,}388\cdot 12 + 989\cdot 11 + \\ 887\cdot 10 + 570\cdot 9 + 454\cdot 8 + 236\cdot 7 + 164\cdot 6 + 66\cdot 5 + \\ 38\cdot 4 + 10\cdot 3 + 5\cdot 2 + 1\cdot 1 + 1\cdot 0 \end{array}\right\}$$

$$= 335{,}728$$

This result represents a great improvement compared with the number of combinations to analyze through exhaustive searching:

$$\sum_{i=0}^{2^5} CTB_i \cdot (2^5 - i) = \qquad 2^{\left(2^5\right)} =$$
$$335{,}728 \qquad << \quad 4{,}294{,}967{,}296$$

For obtaining the configurations for the 6D-OPP's we would have to analyze, through exhaustive searching, a total of 2^{64} = 18,446,744,073,709,551,616 combinations. Through the "Test-Box" heuristic, we found 15,440,344 configurations [Pérez, 03] whose distribution is shown in **Table 5.7**.

TABLE 5.7
Configurations' distribution for 6D-OPP's (Own elaboration).

Boxes (i)	**CTB_i**	**Boxes (i)**	**CTB_i**	**Boxes (i)**	**CTB_i**	**Boxes (i)**	**CTB_i**
0	1	17	148,714	34	702,460	51	41,230
1	1	18	198,627	35	659,016	52	30,040
2	6	19	230,447	36	669,618	53	17,047
3	16	20	296,731	37	600,894	54	11,368
4	77	21	331,481	38	598,040	55	5,631
5	193	22	401,828	39	535,773	56	3,281
6	643	23	431,952	40	506,158	57	1,317
7	1,317	24	506,158	41	431,952	58	643
8	3,281	25	535,773	42	401,828	59	193
9	5,631	26	598,040	43	331,481	60	77
10	11,368	27	600,894	44	296,731	61	16
11	17,047	28	669,618	45	230,447	62	6
12	30,040	29	659,016	46	198,627	63	1
13	41,230	30	702,460	47	148,714	64	1
14	64,892	31	674,771	48	120,156		
15	83,257	32	709,012	49	83,257		
16	120,156	33	674,771	50	64,892		

In the following section (5.4) some formulations related to nD-OPP's configurations will be presented. Finally in section 5.5 some properties of the 'Test-Box' heuristic will be discussed.

5.4 Some Formulations for the Configurations in the nD-OPP's

Theorem 5.1: *The sum of adjacencies for any configuration with x hyper-boxes independently of the Euclidean n-dimensional space, is (see section 4.2.4):*

$$\frac{x(x-1)}{2} = \frac{1}{2}\left(x^2 - x\right)$$

Proof: A first *hyper-box* of the configuration will have *x-1* adjacencies (one for each *x-1 hyper-boxes*); a second *hyper-box* will have *x-2* adjacencies (not including the adjacency with the first *hyper-box* because it is in that first *hyper-box* counting); a third *hyper-box* will have *x-3* adjacencies (not including the adjacencies with the first and second *hyper-boxes* because they are in these *hyper-boxes*' respective counting); in general, a k-th *hyper-box* ($k < x$) will have *x-k* adjacencies. The adjacencies' total counting (i.e. the sum of all *hyper-boxes*' adjacencies) is then defined by the well known expression to compute the sum of the first *x-1* positive integers:

$$\sum_{k=1}^{x-1} k = \frac{x(x-1)}{2} = \frac{x^2 - x}{2} \qquad \square$$

Observation 5.1: *In a n-dimensional configuration consider a m-dimensional subspace ($0 \le m < n$) that passes through the origin. The maximum number of adjacencies embedded in that m-dimensional subspace is 2^{n-1}.*

For example, let $n = 3$ and consider the configuration with 8 boxes ("v", **Table 4.2**). In each one of its three main planes there are $2^{n-1} = 2^{3-1} = 4$ face adjacencies. In each one of its three main axis there are 4 edge adjacencies. Finally, in the origin there are 4 vertex adjacencies.

Lemma 5.1: *In the n-dimensional space, the maximum number of m-dimensional adjacencies for the configuration with 2^n boxes (the configuration with a hyper-box in all its hyper-octants) is:*

$$C\binom{n}{m}\cdot 2^{n-1}, \qquad 0 \le m < n$$

Proof: $C\binom{n}{m}$ is the number of *m*-dimensional subspaces, which are composed by the *m* axes from the n-dimensional space, and there are 2^{n-1} *m*-dimensional adjacencies for each one (by **Observation 5.1**). ◪

For example, let $n = 3$ and $m = 2$, then, there will be $C\binom{3}{2} = 3$ main planes, and each one with $2^2=4$ face adjacencies, giving a total of 12 face adjacencies in the 3D configuration with 8 boxes ("v", **Table 4.2**).

Corollary 5.1: *The total number of adjacencies in a configuration with 2^n boxes (the configuration with a hyper-box in all its hyper-octants) is:*

$$\sum_{m=0}^{n-1} C\binom{n}{m}\cdot 2^{n-1}$$

Proof: Each one of its terms will provide the number of *m*-dimensional adjacencies for the configuration with 2^n boxes. The upper limit for *m* is *n-1* since $0 \le m < n$ (see **Observation 5.1**). ◪

For example, let $n = 4$, then we will have:

$$\sum_{m=0}^{4-1} C\binom{4}{m} \cdot 2^{4-1} = \binom{4}{0} \cdot 2^3 + \binom{4}{1} \cdot 2^3 + \binom{4}{2} \cdot 2^3 + \binom{4}{3} \cdot 2^3 = 8 + 32 + 48 + 32$$

Which represents that, in the 4D configuration with 16 boxes, there are 8 vertex adjacencies, 32 edge adjacencies, 48 face adjacencies and 32 volume adjacencies.

Corollary 5.2: *The sum of adjacencies for the n-dimensional configuration with 2^n hyper-boxes (i.e., with all its hyper-octants filled) is:*

$$\frac{1}{2}\left(2^{2n} - 2^n\right)$$

Proof: **Theorem 5.1** provides a formula for the sum of adjacencies in a configuration with x boxes: $(x^2 - x) / 2$. By doing $x = 2^n$ it will be obtained the sum of adjacencies for the configuration with all its hyper-octants filled:

$$\frac{1}{2}\left(\left(2^n\right)^2 - \left(2^n\right)\right) = \frac{1}{2}\left(2^{2n} - 2^n\right) \qquad \square$$

Theorem 5.2: *A closed form for evaluating the sum in **Corollary 5.1** is given by **Corollary 5.2***

$$\sum_{m=0}^{n-1} C\binom{n}{m} \cdot 2^{n-1} = \frac{1}{2}\left(2^{2n} - 2^n\right)$$

Proof: It is well known that $\sum_{m=0}^{n} C\binom{n}{m} = 2^n$ and since $C\binom{n}{n} = 1$, then $\sum_{m=0}^{n-1} C\binom{n}{m} = 2^n - 1$.

Therefore

$$\begin{aligned}\sum_{m=0}^{n-1} C\binom{n}{m}\cdot 2^{n-1} &= 2^{n-1}\cdot\sum_{m=0}^{n-1} C\binom{n}{m}\\ &= 2^{n-1}\cdot(2^{n}-1)\\ &= 2^{2n-1}-2^{n-1}\\ &= \frac{1}{2}\left(2^{2n}-2^{n}\right)\end{aligned}$$

For example, consider the 10-dimensional configuration with 2^{10} = 1024 boxes. Then we can expect that the number of adjacencies embedded in each subspace is as presented in **Table 5.8**.

TABLE 5.8

The adjacencies in the 10D configuration with 1024 boxes (Own elaboration).

$C\binom{10}{0}\cdot 2^{9} = 512$	Vertex adjacencies
$C\binom{10}{1}\cdot 2^{9} = 5{,}120$	Edge adjacencies
$C\binom{10}{2}\cdot 2^{9} = 23{,}040$	Face adjacencies
$C\binom{10}{3}\cdot 2^{9} = 61{,}440$	Volume adjacencies
$C\binom{10}{4}\cdot 2^{9} = 107{,}520$	4D hypervolume adjacencies
$C\binom{10}{5}\cdot 2^{9} = 129{,}024$	5D hypervolume adjacencies
$C\binom{10}{6}\cdot 2^{9} = 107{,}520$	6D hypervolume adjacencies
$C\binom{10}{7}\cdot 2^{9} = 61{,}440$	7D hypervolume adjacencies
$C\binom{10}{8}\cdot 2^{9} = 23{,}040$	8D hypervolume adjacencies
$C\binom{10}{9}\cdot 2^{9} = 5{,}120$	9D hypervolume adjacencies
$\sum_{m=0}^{9} C\binom{10}{m}\cdot 2^{9} = 523{,}776$	$\frac{1}{2}\left(1024^{2}-1024\right) = 523{,}776$

Corollary 5.3: *The total number of adjacencies in a configuration with 2^n-1 boxes is:*

$$\sum_{m=0}^{n-1} C\binom{n}{m} \cdot (2^{n-1} - 1)$$

Proof: We know by **Observation 5.1** and **Lemma 5.1** that there are at most 2^{n-1} adjacencies in each one of the possible $C\binom{n}{m}$ m-dimensional subspaces in the configuration with 2^n hyper-boxes. By removing a hyper-box from this configuration we remove an adjacency in each one of these m-dimensional subspaces. ◪

For example, consider the 3D configuration with eight boxes ("v", **Table 4.2**). Configuration "v" has four face, edge and vertex adjacencies embedded in each one of their 2D, 1D and 0D-dimensional subspaces respectively (it has a total of 12 face adjacencies, 12 edge adjacencies and 4 vertex adjacencies). By removing a box we obtain the configuration "u" (see **Table 4.2**) with seven boxes. It has three face, edge and vertex adjacencies in each one of their 2D, 1D and 0D-dimensional subspaces respectively giving finally 9 face adjacencies, 9 edge adjacencies and 3 vertex adjacencies.

Corollary 5.4: *The sum of adjacencies for the n-dimensional configuration with 2^n-1 hyper-boxes is:*

$$\frac{1}{2}\left(\left(2^n-1\right)^2-\left(2^n-1\right)\right)=2^{2n-1}-2^n-2^{n-1}+1$$

Proof: **Theorem 5.1** provides a formula for the sum of adjacencies in a configuration with x boxes $(x^2 - x) / 2$. By doing $x = 2^n-1$ it is obtained the sum of adjacencies for the configuration with 2^n-1 *hyper-boxes*. ◪

Theorem 5.3: *A closed form for evaluating the sum in* ***Corollary 5.3*** *is given by* ***Corollary 5.4***

$$\sum_{m=0}^{n-1} C\binom{n}{m}\cdot(2^{n-1}-1)=2^{2n-1}-2^n-2^{n-1}+1$$

Proof: It is well known that $\sum_{m=0}^{n} C\binom{n}{m}=2^n$ and since $C\binom{n}{n}=1$, then $\sum_{m=0}^{n-1} C\binom{n}{m}=2^n-1$.

Therefore

$$\begin{aligned}\sum_{m=0}^{n-1} C\binom{n}{m}\cdot(2^{n-1}-1) &= (2^{n-1}-1)\cdot\sum_{m=0}^{n-1} C\binom{n}{m}\\ &= (2^{n-1}-1)\cdot(2^n-1)\\ &= 2^{2n-1}-2^n-2^{n-1}+1 \quad ◪\end{aligned}$$

For example, consider the 10-dimensional configuration with $2^{10} - 1$= 1023 boxes. Then we can expect that the number of adjacencies embedded in each subspace is as presented in **Table 5.9**.

TABLE 5.9

The adjacencies in the 10D configuration with 1023 boxes (Own elaboration).

$C\binom{10}{0}\cdot(2^9-1)=511$	Vertex adjacencies
$C\binom{10}{1}\cdot(2^9-1)=5{,}110$	Edge adjacencies
$C\binom{10}{2}\cdot(2^9-1)=22{,}995$	Face adjacencies
$C\binom{10}{3}\cdot(2^9-1)=61{,}320$	Volume adjacencies
$C\binom{10}{4}\cdot(2^9-1)=107{,}310$	4D hypervolume adjacencies
$C\binom{10}{5}\cdot(2^9-1)=128{,}772$	5D hypervolume adjacencies
$C\binom{10}{6}\cdot(2^9-1)=107{,}310$	6D hypervolume adjacencies
$C\binom{10}{7}\cdot(2^9-1)=61{,}320$	7D hypervolume adjacencies
$C\binom{10}{8}\cdot(2^9-1)=22{,}995$	8D hypervolume adjacencies
$C\binom{10}{9}\cdot(2^9-1)=5{,}110$	9D hypervolume adjacencies
$\sum_{m=0}^{9} C\binom{10}{m}\cdot(2^9-1)=522{,}753$	$\frac{1}{2}(1023^2-1023)=522{,}753$

5.5 Some Properties of the "Test-Box" Heuristic

The algorithm described in section 5.2.2 will be considered except for the steps 1 and 4, in other words, the (n-1)D configurations won't be generated nor extruded (step 1) and a boxes' combination won't be compared with others to determine if it is a configuration (step 2). That implies that the algorithm will start with the configuration with zero boxes.

Figure 5.1 presents the graph generated when the 2D configurations are obtained. In level 0 there is only the configuration with zero boxes, which contains four empty quadrants and therefore there are four possible positions for adding a "test-box" (level 1). By adding a second "test-box" at each of the three remaining positions, all possible permutations with two boxes will be obtained (level 2). By adding a box at each of the remaining two positions, all possible permutations with three boxes are obtained (level 3) and finally, by adding a fourth box in the last empty position, all possible permutations with four boxes are obtained (level 4).

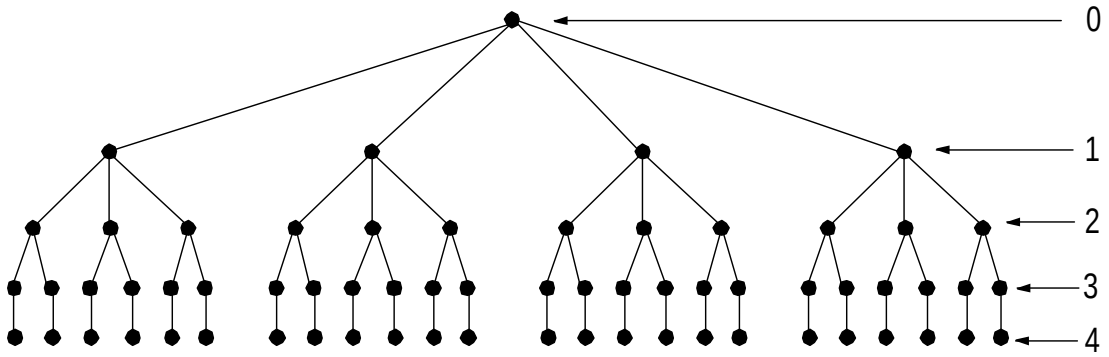


FIGURE 5.1
The graph generated by the "Test-Box" algorithm for determining the 2D configurations (See the text for details. Own elaboration).

Property 5.1: *The graph generated by the "Test-Box" algorithm has* $2^n + 1$ *levels. This property is obvious and arises from the fact that n-dimensional space has* 2^n *hyper-octants which can be occupied by 1, 2, 3, …,* 2^n *boxes. Moreover, it must considered the case when all the hyper-octants are empty.*

For example, the graph generated by the "Test-Box" algorithm for determining the 7D Orthogonal Pseudo-Polytopes' configurations will have $2^7+1=129$ levels.

Property 5.2: *The graph generated by the "Test-Box" algorithm is a permutation tree. In level 1 a box will be positioned in each one of the 2^n empty hyper-octants. In level 2 only $2^n - 1$ hyper-octants will be available to be occupied by a second box according to all the possibilities, and so forth. By associating the available positions for a box in each one of the possible levels we will have:*

$$\underbrace{(2^n)}_{level\ 1} \cdot \underbrace{(2^n - 1)}_{level\ 2} \cdot \underbrace{(2^n - 2)}_{level\ 3} \cdot \ldots \cdot \underbrace{(1)}_{level\ 2^n} = (2^n)!$$

Where $(2^n)!$ provides the count of the permutations with 2^n boxes (all the hyper-octants occupied) in the 2^n-th level.

For example, in the level 512 ($n = 9$, i.e. 9D space) the generated permutation tree will have 512! possible permutations of the configuration with all its hyper-octants occupied.

Property 5.3: *Each level i from the permutation tree, $0 \le i \le 2^n$, has ${}^{2^n}_{i}P = \dfrac{2^n!}{(2^n - i)!}$ permutations, because each box is occupying each one of the possible available hyper-octants. Obviously the total number of permutations in the tree will be equal to:*

$$\sum_{i=0}^{2^n} {}^{2^n}_{i}P$$

For example, see **Figure 5.1** (n=2). The number of permutations in each level is presented in the **Table 5.10**.

TABLE 5.10
Counts of the permutation tree's levels for generating the configurations for the 2D-OPP's (own elaboration).

Level (i)	${}^{2^n}_{i}P = \dfrac{2^n!}{(2^n - i)!}$
0	1
1	4
2	12
3	24
4	24
Total:	$\sum_{i=0}^{2^n} {}^{2^n}_{i}P = 65$

Property 5.4: *It is known that there is one configuration with 0 boxes and one configuration with 1 boxes. Then by considering the application of step 4, in the algorithm presented in section 5.2.2, from all the generated permutations with a box in tree's level 1 only one of them will be considered, which automatically discards to* ${}^{2^n}_{1}P - 1$ *sub-trees whose root is an element from level 1.*

For example, in **Figure 5.2** is presented the possible sub-tree chosen by the "Test-Box" algorithm for determining the 2D-OPP's configurations. The three remaining sub-trees are not considered again.

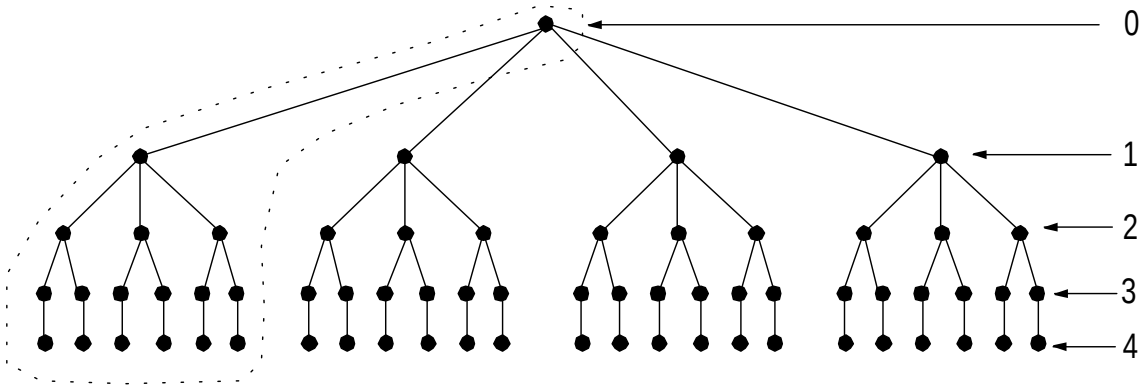


FIGURE 5.2
The first sub-tree chosen by the "Test-Box" algorithm
(See the text for details, own elaboration).

Starting from this section, when referring to the term "permutation tree" we will consider to the sub-tree mentioned in **Property 5.4**.

Property 5.5: *The number of elements in each level i from the permutation tree is given by the function E'(i,n):*

$$E'(i,n) = \begin{cases} 1 & i = 0 \\ 1 & i = 1 \\ \dfrac{{}^{2^n}_{i}P}{{}^{2^n}_{1}P} = \dfrac{{}^{2^n}_{i}P}{2^n} = \dfrac{2^n!}{2^n \cdot (2^n - i)!} & 1 < i \le 2^n \end{cases}$$

When the level i is greater than 1, the number of the possible permutations in each level must be divided by the number of permutations with one box.

For example, the number of permutations in each level from the generated tree for determining the 3D configurations is presented in **Table 5.11**.

TABLE 5.11
Counts of the permutation tree's levels for generating the configurations for the 3D-OPP's (own elaboration).

Level (i)	E'(i,3)
0	1
1	1
2	7
3	42
4	210
5	840
6	2,520
7	5,040
8	5,040

Property 5.6: *We know that there are only n configurations with two boxes. Therefore by considering step 4, from the algorithm presented in section 5.2.2, only n permutations will be selected from the possible E'(2,n). Which automatically discards to the E'(2,n) – n sub-trees whose root is one of the permutations from level 2.*

For example, in **Figure 5.3** are indicated the two selected sub-trees (one for each configuration with 2 boxes in level 2) by the "Test-Box" algorithm. The remaining sub-tree is not considered again.

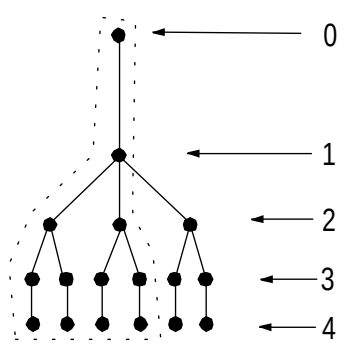


FIGURE 5.3
The two sub-trees selected in level 2 by the "Test-Box" algorithm
(See the text for details, own elaboration).

Starting from this section, when referring to the term “permutation tree” we will consider the tree whose levels 0, 1 and 2 has 1, 1 and n configurations respectively.

Property 5.7: *The number of elements in each level i from the permutation tree is given by the function $E^2(i,n)$:*

$$E^2(i,n) = \begin{cases} 1 & i = 0 \\ 1 & i = 1 \\ n & i = 2 \\ \dfrac{{}^{2^n}_{i}P}{2^n} \cdot \dfrac{n \cdot 2^n}{{}^{2^n}_{2}P} = \dfrac{n \cdot {}^{2^n}_{i}P}{{}^{2^n}_{2}P} & 2 < i \le 2^n \end{cases}$$

The factor $\dfrac{n \cdot 2^n}{{}^{2^n}_{2}P}$ *arises from* $\dfrac{n}{E'(2,n)}$ *which indicates the relation between the n configurations with 2 boxes and the E’(2,n) possible permutations.*

For example, the number of permutations in each level from the tree for determining the 3D configurations is presented in **Table 5.12**.

TABLE 5.12

Using the function $E^2(i,n)$ for determining the counts of the permutation tree’s levels for generating the configurations for the 3D-OPP’s (own elaboration).

Level (i)	$E^2(i,3)$
0	1
1	1
2	3
3	18
4	90
5	360
6	1,080
7	2,160
8	2,160

Property 5.8: *We know that there are n configurations with $2^n - 2$ boxes (because they are the complementary configurations with 2 boxes). Therefore the "Test-Box" algorithm will select only n permutations from the possible $E^2(2,n)$. Moreover, we know that there are one configuration with $2^n - 1$ boxes and one configuration with 2^n boxes (because they are the complementary configurations with 1 and 0 boxes respectively) which will be selected by the algorithm from the possible $E^2(2^n-1,n)$ and $E^2(2^n,n)$ respectively. Finally, the level 2^n from the permutation tree will have just one element, level 2^n-1 also will have one element and level 2^n-2 will have n elements.*

Figure 5.4 presents the permutation tree generated by the algorithm for obtaining the 2D configurations (n = 2). In this case level 2 and level 2^n-2 are the same.

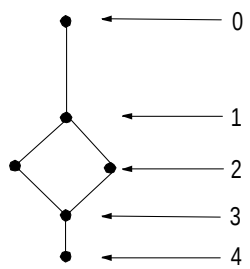


FIGURE 5.4
The permutation tree for obtaining the 2D configurations
(own elaboration).

Definition 5.3: *The number of elements in each level i from the permutation tree generated by the "Test-Box" algorithm is determined by the function E(i,n):*

$$E(i,n)=\begin{cases} 1 & i=0 \\ 1 & i=1 \\ n & i=2 \\ \dfrac{n\cdot {}^{2^n}_{i}P}{{}^{2^n}_{2}P} & 2<i<2^n-2 \\ n & i=2^n-2 \\ 1 & i=2^n-1 \\ 1 & i=2^n \end{cases}$$

Then, for example, if n=7 we can expect that its permutation tree in level 49 will have $E(49,7)=\dfrac{7\cdot {}^{128}_{49}P}{{}^{128}_{2}P}=\dfrac{128\cdot(4.31x10^{98})}{16256}=3.394x10^{96}$ elements.

The described properties until this moment create a link between the “Test-Box” Heuristic and a permutation tree. However, determining the configurations for the nD-OPP’s through exhaustive searching links that procedure with the generation of combinations of boxes (see section 4.2). It is obvious that an analysis based in combinations will have a minor complexity compared with other whose base is an analysis based in permutations. However, it must be considered that for each level of the tree related with the “Test-Box” heuristic some permutations are discarded, which rebounds in the sub-trees whose roots are precisely the discarded nodes, because they are not considered again by the algorithm. With the following property we will justify this asseveration.

Property 5.9: *A node with x boxes in the permutation tree can generate exactly 2^n - x permutations, one for each of the empty hyper-octants. Each one of the 2^n – x generated permutations will produce 2^n – x – 1 boxes and so forth. Then the number of generated nodes starting from a node with x boxes will be:*

$$1 + (2^n - x) + \sum_{j=2}^{2^n - x} (2^n - x) \prod_{i=1}^{j-1} (2^n - x - i)$$

$$0 \le x \le 2^n$$

The first term (1, level 0) counts to the node with x boxes itself, the second term (2^n-x, level 1) counts the number of permutations generated from the given node. Each one of the 2^n-x nodes generated through the root can generate up to 2^n-x-1 nodes (level 2) each one, until this level we would have counted to $1 + (2^n - x) + (2^n - x) \cdot (2^n - x - 1)$ nodes. In general, the number of nodes in a level *j* will be equal to the product of the generated nodes through just one node in level *j-1* by the number of nodes in that level (*j-1*). At its time, the number of nodes in the level *j-1* will be equal to the product of the generated nodes through just one node in level *j-2* by the number of nodes in that level (*j-2*) and so forth until level 1 whose number of nodes is 2^n-x. Then, such count will be given by the product:

$$(2^n - x) \prod_{i=1}^{j-1} (2^n - x - i)$$

Due to starting from a node with x boxes it is necessary to analyze 2^n-x levels for which their number of nodes will be counted, then finally we obtain the third term in the given formula:

$$\sum_{j=2}^{2^n - x} (2^n - x) \prod_{i=1}^{j-1} (2^n - x - i)$$

If a node x is ignored by the “Test-Box” algorithm, then also the nodes that could be generated starting from it will be ignored (i.e. the sub-tree whose root is x). Then, the above formula allows us to determine the ignored sub-tree’s number of elements.

The following example will show how by not considering a node the number of permutations to analyze is reduced drastically:

Let n=4, 2^n=16, and suppose a ignored 4D configuration with x=4 boxes, then we will have:

$$1+(16-4)+\sum_{j=2}^{16-4}(16-4)\prod_{i=1}^{j-1}(16-4-i)=1+12+\sum_{j=2}^{12}12\prod_{i=1}^{j-1}(12-i)$$

In the level 0 we have the permutation itself with 4 boxes (1 node). Such permutation generates in total 12 permutations with 5 boxes (level 1). For each term in the sum $\left(12\prod_{i=1}^{j-1}(12-i)\right)$ we have:

Level j=2 (permutations with 6 boxes),	$12\prod_{i=1}^{1}(12-i)$ = 12(12-1) = 132
Level j=3 (permutations with 7 boxes),	$12\prod_{i=1}^{2}(12-i)$ = 12(12-1)(12-2) = 1,320
Level j=4 (permutations with 8 boxes),	$12\prod_{i=1}^{3}(12-i)$ = 12(12-1)(12-2)(12-3) = 11,880
Level j=5 (permutations with 9 boxes),	$12\prod_{i=1}^{4}(12-i)$ = 12(12-1)(12-2)(12-3)(12-4) = 95,040

Level j=6 (permutations with 10 boxes),	$12\prod_{i=1}^{5}(12-i)$ = 12(12-1)(12-2)(12-3)(12-4)(12-5) = 665,280
Level j=7 (permutations with 11 boxes),	$12\prod_{i=1}^{6}(12-i)$ = 12(12-1)(12-2)(12-3)(12-4)(12-5)(12-6) = 3,991,680
Level j=8 (permutations with 12 boxes),	$12\prod_{i=1}^{7}(12-i)$ = 12(12-1)(12-2)(12-3)(12-4)(12-5)(12-6)(12-7) = 19,958,400
Level j=9 (permutations with 13 boxes),	$12\prod_{i=1}^{8}(12-i)$ = 12(12-1)(12-2)(12-3)(12-4)(12-5)(12-6)(12-7)(12-8) = 79,833,600
Level j=10 (permutations with 14 boxes),	$12\prod_{i=1}^{9}(12-i)$ = 12(12-1)(12-2)(12-3)(12-4)(12-5)(12-6)(12-7)(12-8)(12-9) = 239,500,800
Level j=11 (permutations with 15 boxes),	$12\prod_{i=1}^{10}(12-i)$ = 12(12-1)(12-2)(12-3)(12-4)(12-5)(12-6)(12-7)(12-8)(12-9)(12-10) = 479,001,600
Level j=12 (permutations with 16 boxes),	$12\prod_{i=1}^{11}(12-i)$ = 12(12-1)(12-2)(12-3)(12-4)(12-5)(12-6)(12-7)(12-8)(12-9)(12-10)(12-11) = 479,001,600

And by adding all the terms finally it is found that there are:

$$1+12+\sum_{j=2}^{12}12\prod_{i=1}^{j-1}(12-i)=1{,}302{,}061{,}345 \text{ ignored permutations.}$$

Let n = 6 (the 6D space), 2^n = 64 (hyper-octants), and the ignored permutation with x = 23 boxes, then we will find that the number of permutations in the sub-tree ignored by the “Test-Box” algorithm (whose root is the given permutation) is:

90,933,395,208,605,785,401,971,970,164,779,391,644,753,259,799,242

This computation was possible due to an implementation of the formula in the high level language Java and by using the class *BigInteger* which provides the possibility of the handling of “Immutable arbitrary-precision integers”. The referred implementation is the following:

```
BigInteger getSubTreeSize(int X, int n)
{
    // X: number of boxes in the ignored permutation.
    // n: number of dimensions in the space.
    // Formula’s first term.
    BigInteger firstTerm = new BigInteger("1");
    // Formula’s second term.
    BigInteger secondTerm = new BigInteger(Integer.toString((int) Math.pow(2,n) - X));
    int j = 2;
    int k = (int) Math.pow(2,n) - X;
    BigInteger finalCount = firstTerm.add(secondTerm);
    // The formula’s sum will be executed.
    while(j <= k)
    {
        BigInteger firstProduct = new BigInteger(Integer.toString((int) Math.pow(2,n) - X));
        BigInteger secondProduct = new BigInteger("1");
        int i = 1;
        // The formula’s product will be executed.
        while(i <= j - 1)
        {
            // The formula’s product i-esimal term.
            secondProduct=
            secondProduct.multiply(new BigInteger(Integer.toString((int) Math.pow(2,n)-X- i)));
            i++;
        }
        // The formula’s sum j-esimal term.
        BigInteger finalProduct = firstProduct.multiply(secondProduct);
        finalCount = finalCount.add(finalProduct);
        j++;
    }
    // The final count.
    return finalCount;
}
```

5.6 Binary Representation for the Configurations in the nD-OPP's

In this section we will consider a useful representation for the configurations in the nD-OPP's. We will define such representation specifically for the 3D case (section 5.6.1) and then to the nD case (section 5.6.2).

5.6.1 Binary Representation for the Configurations in the 3D-OPP's

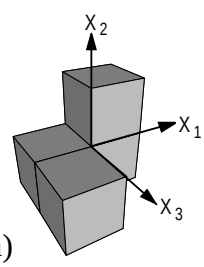
A 3D-OPP's configuration can be represented through a binary string of eight bits. These bits will indicate the 3D space's octants. If a bit has a value equal to one then its referred octant is occupied by a box; otherwise, the octant is empty. Since we will have eight bits in the binary string, the position of each bit can be interpreted as a binary number with three digits (000_2 … 111_2). These three digits will be associated with each one of the 3D space's main axes by considering the most significant bit as a reference to the X_1 axis, the subsequent bit as a reference to the X_2 axis and the least significant bit as a reference to the X_3 axis. Moreover, if a bit is 0 then we will consider the positive part of the corresponding axis; otherwise, we will consider its negative part. Then, through the binary representation of the position of a bit in the configuration's string we can infer its corresponding octant. For example, if the 6-th bit has a value equal to one, then we know that there is a box in the octant $\bar{x}_1 \bar{x}_2 x_3$ because $6_{10} = 110_2$ (a superposed bar on x_k indicates that we are considering the negative part of the referred axis). The correspondences between the bits' positions in the configuration's binary string with their octants are presented in **Table 5.13**.

TABLE 5.13

Correspondences between bits' positions in a 3D configuration's binary string and their octants (own elaboration).

Bit's Position	**Position's Binary Representation**			**Corresponding Octant** (Descriptive Axes)
	Most Significant Bit		Least Significant Bit	
0	0	0	0	$x_1 x_2 x_3$
1	0	0	1	$x_1 x_2 \overline{x}_3$
2	0	1	0	$x_1 \overline{x}_2 x_3$
3	0	1	1	$x_1 \overline{x}_2 \overline{x}_3$
4	1	0	0	$\overline{x}_1 x_2 x_3$
5	1	0	1	$\overline{x}_1 x_2 \overline{x}_3$
6	1	1	0	$\overline{x}_1 \overline{x}_2 x_3$
7	1	1	1	$\overline{x}_1 \overline{x}_2 \overline{x}_3$

For example, the combination with four boxes in **Figure 5.5** corresponds to the string 01010011. By associating the positions of those bits whose value is equal to one with its corresponding octants (**Table 5.14**) it is determined that the four boxes are distributed in the octants $x_1 x_2 \overline{x}_3$, $x_1 \overline{x}_2 \overline{x}_3$, $\overline{x}_1 \overline{x}_2 x_3$ and $\overline{x}_1 \overline{x}_2 \overline{x}_3$.



a)

0	1	0	1	0	0	1	1

b)

FIGURE 5.5

a) A boxes' combination for 3D configuration "j" and b) its binary representation (own elaboration).

TABLE 5.14

Octants occupied by the boxes indicated in a 3D combination's binary string (own elaboration).

Configuration's Binary String	0	1	0	1	0	0	1	1
Positions	0	1	2	3	4	5	6	7
Positions' Binary Representations	000	001	010	011	100	101	110	111
Corresponding Octants	$x_1 x_2 x_3$	$x_1 x_2 \overline{x}_3$	$x_1 \overline{x}_2 x_3$	$x_1 \overline{x}_2 \overline{x}_3$	$\overline{x}_1 x_2 x_3$	$\overline{x}_1 x_2 \overline{x}_3$	$\overline{x}_1 \overline{x}_2 x_3$	$\overline{x}_1 \overline{x}_2 \overline{x}_3$

Definition 5.4: *Let* **C(a, b)** *be the number of bits that* <u>*don't change*</u> *from binary string 'a' respect to binary string 'b'.*

For example:

C(110, 001) = 0	(no bit remains unchanged)
C(1<u>1</u>0, 0<u>1</u>1) = 1	(bit 1 does not change)
C(<u>11</u>0, <u>11</u>1) = 2	(bits 0 and 1 do not change)

By comparing the binary representations of the positions of two boxes we can infer the type of adjacency between them. If $C(a_2, b_2)$ is equal to two, it implies that two bits don't change and therefore these unchanged bits will refer to the positive or negative parts of two main axes which define specifically a shared face. For example, by considering two boxes whose binary positions are a = 101 and b = 100, we have that C(<u>10</u>1, <u>10</u>0) = 2, i.e. there is a shared face (face adjacency) which is defined by the first and second unchanged bits in both binary strings and whose corresponding axes are $\bar{x}_1 x_2$. If $C(a_2, b_2) = 1$ then we have an edge adjacency between two boxes which is defined by the positive or negative part of the axis defined by the unchanged bit. If $C(a_2, b_2) = 0$ then we have a vertex adjacency which only takes place at the origin.

Definition 5.5: *Let* **Adj$_3$(a,b)** *be the function that computes the type of adjacency between two 3D boxes referred through the binary digits that correspond to their respective octants. Then we will have:*

$$Adj_3(a,b) = \begin{cases} \Pi_2 & (face\ \ adjacency) \quad iff \quad C(a,b) = 2 \\ \Pi_1 & (edge\ \ adjacency) \quad iff \quad C(a,b) = 1 \\ \Pi_0 & (vertex\ \ adjaceny) \quad iff \quad C(a,b) = 0 \end{cases}$$

In other words:

$$Adj_3(a,b) = \Pi_{C(a,b)}$$

For example we will determine the adjacencies between the four boxes in the example from **Table 5.14** and **Figure 5.5**:

$Adj_3(\underline{0}1\underline{1}, \underline{0}0\underline{1}) = \Pi_2$ (Face adjacency)

$Adj_3(0\underline{1}1, 1\underline{1}0) = \Pi_1$ (Edge adjacency)

$Adj_3(0\underline{11}, 1\underline{11}) = \Pi_2$ (Face adjacency)

$Adj_3(001, 110) = \Pi_0$ (Vertex adjacency)

$Adj_3(00\underline{1}, 11\underline{1}) = \Pi_1$ (Edge adjacency)

$Adj_3(\underline{11}0, \underline{11}1) = \Pi_2$ (Face adjacency)

5.6.2 Representing n-dimensional Configurations

A nD-OPP's configuration can be represented through a binary string with 2^n bits. These bits will indicate the nD space's hyper-octants. If a bit has a value equal to one then its referred hyper-octant is occupied by a *hyper-box*; otherwise, the hyper-octant is empty. Since we will have 2^n bits in the binary string, the position of each bit can be interpreted as a binary number with n digits ($\underbrace{0\cdots0}_{n}{}_2$ … $\underbrace{1\cdots1}_{n}{}_2$). These n digits will be associated with each one of the nD space's main axes by considering the most significant bit as a reference to the X_1 axis, the subsequent bit as a reference to the X_2 axis, and so forth until we consider the least significant bit as a reference to the X_n axis. Moreover, if a bit is 0 then we

will consider the positive part of the corresponding axis; otherwise, we will consider its negative part. Then, through the binary representation of the position of a bit in the configuration's string we can infer its corresponding octant. For example, if the 22-th bit in a 5D configuration's binary string has a value equal to one, then we infer that there is a *hyper-box* in the hyper-octant $\overline{x}_1 x_2 \overline{x}_3 \overline{x}_4 x_5$ because $22_{10} = 10110_2$. As a specific case we present in **Table 5.15** the correspondences between the bits' positions in a 4D configuration's binary string with their hyper-octants.

TABLE 5.15
Correspondences between bits' positions in a 4D configuration binary string and their hyper-octants (own elaboration).

Bit's Position	**Position's Binary Representation**				**Corresponding Hyper-octant** (Descriptive Axes)
	Most Significant Bit			Least Significant Bit	
0	0	0	0	0	$x_1 x_2 x_3 x_4$
1	0	0	0	1	$x_1 x_2 x_3 \overline{x}_4$
2	0	0	1	0	$x_1 x_2 \overline{x}_3 x_4$
3	0	0	1	1	$x_1 x_2 \overline{x}_3 \overline{x}_4$
4	0	1	0	0	$x_1 \overline{x}_2 x_3 x_4$
5	0	1	0	1	$x_1 \overline{x}_2 x_3 \overline{x}_4$
6	0	1	1	0	$x_1 \overline{x}_2 \overline{x}_3 x_4$
7	0	1	1	1	$x_1 \overline{x}_2 \overline{x}_3 \overline{x}_4$
8	1	0	0	0	$\overline{x}_1 x_2 x_3 x_4$
9	1	0	0	1	$\overline{x}_1 x_2 x_3 \overline{x}_4$
10	1	0	1	0	$\overline{x}_1 x_2 \overline{x}_3 x_4$
11	1	0	1	1	$\overline{x}_1 x_2 \overline{x}_3 \overline{x}_4$
12	1	1	0	0	$\overline{x}_1 \overline{x}_2 x_3 x_4$
13	1	1	0	1	$\overline{x}_1 \overline{x}_2 x_3 \overline{x}_4$
14	1	1	1	0	$\overline{x}_1 \overline{x}_2 \overline{x}_3 x_4$
15	1	1	1	1	$\overline{x}_1 \overline{x}_2 \overline{x}_3 \overline{x}_4$

For example, the string 0100011001110000 will define a 4D configuration with six *hyper-boxes*. By associating the positions of those bits whose value is equal to one with its corresponding hyper-octants (**Table 5.16**) it is determined that the six *hyper-boxes* are distributed in the hyper-octants $x_1 x_2 x_3 \overline{x}_4$, $x_1 \overline{x}_2 x_3 \overline{x}_4$, $x_1 \overline{x}_2 \overline{x}_3 x_4$, $\overline{x}_1 x_2 x_3 \overline{x}_4$, $\overline{x}_1 x_2 \overline{x}_3 x_4$ and $\overline{x}_1 x_2 \overline{x}_3 \overline{x}_4$.

TABLE 5.16
Hyper-octants occupied by the *hyper-boxes* indicated in a 4D configuration's binary string (own elaboration).

Configuration's Binary String	**Positions**	**Positions' binary representation**	**Corresponding Hyper-octants**
0	0	0000	$x_1 x_2 x_3 x_4$
1	1	0001	$x_1 x_2 x_3 \overline{x}_4$
0	2	0010	$x_1 x_2 \overline{x}_3 x_4$
0	3	0011	$x_1 x_2 \overline{x}_3 \overline{x}_4$
0	4	0100	$x_1 \overline{x}_2 x_3 x_4$
1	5	0101	$x_1 \overline{x}_2 x_3 \overline{x}_4$
1	6	0110	$x_1 \overline{x}_2 \overline{x}_3 x_4$
0	7	0111	$x_1 \overline{x}_2 \overline{x}_3 \overline{x}_4$
0	8	1000	$\overline{x}_1 x_2 x_3 x_4$
1	9	1001	$\overline{x}_1 x_2 x_3 \overline{x}_4$
1	10	1010	$\overline{x}_1 x_2 \overline{x}_3 x_4$
1	11	1011	$\overline{x}_1 x_2 \overline{x}_3 \overline{x}_4$
0	12	1100	$\overline{x}_1 \overline{x}_2 x_3 x_4$
0	13	1101	$\overline{x}_1 \overline{x}_2 x_3 \overline{x}_4$
0	14	1110	$\overline{x}_1 \overline{x}_2 \overline{x}_3 x_4$
0	15	1111	$\overline{x}_1 \overline{x}_2 \overline{x}_3 \overline{x}_4$

By comparing the binary representations of the positions of two hyper-boxes in a n-dimensional configuration we can infer the type of adjacency between them. If C(a, b) is equal to n-1 (see **Definition 5.4**), it implies that n-1 bits don't change and therefore these unchanged bits will refer to the positive or negative parts of n-1 main axes which define

specifically a Π_{n-1} shared cell. If C(a,b) = n-2 then we have an (n-2)-D adjacency (a Π_{n-2} shared cell); and so forth until the cases when C(a,b) = 1 (an edge adjacency) and C(a,b)=0 (a vertex adjacency).

Definition 5.6: *Let* **Adj_n(a, b)** *be the function that computes the type of adjacency between two n-dimensional hyper-boxes referred through the binary digits that correspond to their respective hyper-octants. Then we will have:*

$$Adj_n(a,b) = \begin{cases} \Pi_{n-1} & ((n-1)D\ adjacency) & iff & C(a,b) = & n-1 \\ \Pi_{n-2} & ((n-2)D\ adjacency) & iff & C(a,b) = & n-2 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ \Pi_1 & (edge\ adjacency) & iff & C(a,b) = & 1 \\ \Pi_0 & (vertex\ adjacency) & iff & C(a,b) = & 0 \end{cases}$$

In other words:

$$Adj_n(a,b) = \Pi_{C(a,b)}$$

For example, $Adj_4(a,b)$ will be defined as:

$$Adj_4(a,b) = \begin{cases} \Pi_3 & (volume\ adjacency) & iff & C(a,b) = 3 \\ \Pi_2 & (\ face\ adjacency) & iff & C(a,b) = 2 \\ \Pi_1 & (edge\ adjacency) & iff & C(a,b) = 1 \\ \Pi_0 & (vertex\ adjacency) & iff & C(a,b) = 0 \end{cases}$$

Then, the adjacencies between the six hyper-boxes from 4D configuration 67 (see **Appendix A** and **Table 5.16**) are shown in **Table 5.17**.

TABLE 5.17

The adjacencies between the six hyper-boxes of a 4D configuration (own elaboration).

Shared (n-k)-dimensional cell	Shared (n-k)-dimensional cell
$Adj_4(\underline{0}0\underline{0}1, \underline{0}1\underline{0}1) = 3$ Π_3 (Volume)	$Adj_4(010\underline{1}, 101\underline{1}) = 1$ Π_1 (Edge)
$Adj_4(\underline{0}001, \underline{0}110) = 1$ Π_1 (Edge)	$Adj_4(0110, 1001) = 0$ Π_0 (Vertex)
$Adj_4(0\underline{00}1, 1\underline{00}1) = 3$ Π_3 (Volume)	$Adj_4(01\underline{10}, 10\underline{10}) = 2$ Π_2 (Face)
$Adj_4(0\underline{0}01, 1010) = 1$ Π_1 (Edge)	$Adj_4(01\underline{1}0, 10\underline{1}1) = 1$ Π_1 (Edge)
$Adj_4(0\underline{0}0\underline{1}, 1\underline{0}1\underline{1}) = 2$ Π_2 (Face)	$Adj_4(\underline{10}01, \underline{10}10) = 2$ Π_2 (Face)
$Adj_4(\underline{01}01, \underline{01}10) = 2$ Π_2 (Face)	$Adj_4(\underline{10}0\underline{1}, \underline{10}1\underline{1}) = 3$ Π_3 (Volume)
$Adj_4(01\underline{01}, 10\underline{01}) = 2$ Π_2 (Face)	$Adj_4(\underline{101}0, \underline{101}1) = 3$ Π_3 (Volume)
$Adj_4(0101, 1010) = 0$ Π_0 (Vertex)	

The following is an implementation of the procedure to calculate the adjacencies counting for a configuration represented through its binary string:

```
int [ ] analyzeAdjacencies(BinaryString hyperboxes, int dimensions)
{
    int adjacencies[ ] = new int[dimensions];
    for(i = 0; i < 2^dimensions; i++)
        if(hyperboxes[i] == 1)
        {
            BinaryString hyperOctantA = getBinaryRepresentation(i);
            for(j = i + 1; j < 2^dimensions; j++)
                if(hyperboxes[j] == 1)
                {
                    BinaryString hyperOctantB = getBinaryRepresentation(j);
                    int adjacency = Adj(hyperOctantA, hyperOctantB, dimensions);
                    adjacencies[adjacency]++;
                }
        }
    return adjacencies;
}
```

Some considerations for the algorithm are:

- The function *getBinaryRepresentation* has as input an integer and returns its respective binary representation.
- The function *Adj* implements the function from **Definition 5.6**. It has as inputs the positions of two hyper-boxes (indicated by variables *i* and *j* respectively) using their binary representation and the number of *dimensions* (in order to get the appropriate adjacencies' evaluation according to the nD space).
- The array *adjacencies* will store the counting of the adjacencies between all the hyper-boxes in the configuration. Due to function *Adj* returns values in the interval [0, n-1], such values will indicate the position in the array *adjacencies* whose value must be increased. When the main cycle finishes, this array has the final adjacencies counting, the first position will contain the number of vertex adjacencies, the second position will contain the number of edge adjacencies and so forth; until the last position will contain the number of (n-1)-dimensional adjacencies.

5.7 The Hill vs. Aguilera & Pérez's Configurations for the 4D-OPP's

In this section we will discuss some important aspects related to the determination of the configurations for the 4D-OPP's. Specifically we will consider the differences between the procedures for obtaining such configurations according to the methods described by [Hill, 98] and Aguilera & Pérez in [Pérez, 01].

5.7.1 Obtaining the Hill's Configurations for the 4D-OPP's

In the section 4.2.3 we specified in a general way the procedure for obtaining the configurations for the 4D-OPP's. We mentioned that the 65,536 possible combinations from 0 to 16 hyper-boxes can be grouped by applying symmetries and rotations. In [Hill,98] it is considered this same procedure by specifying the application of some of the following transformations:

- Rotations:
  - Around the X_1X_2, X_1X_3, X_1X_4, X_2X_3, X_2X_4 and X_3X_4 planes and with angles 90°, 180° and 270°.
- Symmetries:
  - Reflections respect to the X_1, X_2, X_3 and X_4 axes.

This way we can say that a hyper-boxes' combination Cn_1 will be equivalent to other combination Cn_2 (i.e. they belong to the same equivalence class) if there exists a composition T^n of the mentioned transformations such that:

$$Cn_1 \equiv T^n(Cn_2)$$

For example, in the **Table 5.18** there are shown some examples of 4D combinations Cn_1 and Cn_2 and the transformations to apply in order to obtain their proper equivalencies.

TABLE 5.18

Examples of 4D combinations Cn_1 and Cn_2 and the transformations to apply such that $Cn_1 \equiv T^n(Cn_2)$ (R_1 stands for reflection respect X_1; own elaboration).

Ex.	***Cn_1* (Binary Representation)**	***T^n***	***Cn_2* (Binary Representation)**
1	1001100000000000	$R_{1,4}(180°) \cdot R_{1,2}(180°) \cdot R_{2,3}(90°)$	0010001010000000
2	1110000000000000	$R_{1,4}(180°) \cdot R_{2,4}(180°) \cdot R_{1,2}(180°)$	0000000000000111
3	1001100000000000	$R_{1,4}(180°) \cdot R_{2,4}(270°) \cdot R_{1,2}(180°) \cdot R_{2,3}(90°)$	0011000001000000
4	1001100010000000	$R_{1,4}(270°) \cdot R_{2,4}(270°) \cdot R_1$	1011100000000000
5	0011010010000000	$R_{1,4}(270°) \cdot R_{2,4}(180°) \cdot R_{3,4}(90°) \cdot R_{2,3}(90°) \cdot R_1$	0100100000001001

By this way, we can develop an implementation to find the configurations for the 4D-OPP's. The algorithm will find such configurations in the following way:

- We will have a set *configurations* that contains all the hyper-boxes' sets that represent each one of the identified equivalence classes.
- A *combination* of hyper-boxes which will be evaluated with each one of the hyper-boxes' sets in *configurations* in order to determine if that *combination* can be a representative of a new configuration. If there exists a composition of transformations such that $configuration \in configurations \equiv T^n(combination)$ then the evaluated *combination* has a representative in *configurations*; otherwise, the *combination* of hyper-boxes is the representative of a new configuration and it must be added to the set *configurations*.
- The implementation receives as input the number $0 \leq N \leq 16$ of hyper-boxes whose configurations will be determined. The output will be the set *configurations* that contains all the hyper-boxes' sets that are representative of all the configurations with N hyper-boxes.

```
Vector getConfigurations(int N)
{
    Vector configurations = new Vector( );
    BinaryString combination = 0000000000000000; //Specific for 4D space.
    for(int i = 0; i < 65536; i++)
    {
        if(getNumberOfHyperBoxes(combination) == N)
        {
            for(int j = 0; j < confs; j++)
            {
                BinaryString configuration = configurations.elementAt(j);
                if(existsComposition(combination, configuration) == true)
                    break;
            }
            if(j == confs)
                configurations.add(combination);
        }
        getNextCombination(combination);
    }
    return configurations;
}
```

The function *existsComposition* will have as input the two hyper-boxes' sets that correspond to a combination and a configuration. Its objective is to search exhaustively a composition of transformations (rotations and/or reflections) such that $configuration \equiv T^n(combination)$. A composition of transformations is represented by an array of seven digits whose values are in the interval [0, 4]. We have a total of $4^6 \cdot 5 = 20{,}480$ combinations or compositions. Each position in the array will be related to an specific transformation in the following way:

- Position 0: Rotation around X_1X_4 plane.
- Position 1: Rotation around X_2X_4 plane.
- Position 2: Rotation around X_3X_4 plane.
- Position 3: Rotation around X_1X_2 plane.
- Position 4: Rotation around X_2X_3 plane.
- Position 5: Rotation around X_1X_3 plane.
- Position 6: Reflection.

Moreover, the positions' values will indicate the parameters of the transformations:

- From position 0 to 5, the values 1, 2 or 3 will indicate a rotation angle of 90°, 180° and 270° respectively. Value 0 will indicate that the referred rotation won't be applied.
- In position 6, the values 1, 2, 3 and 4 will indicate a reflection respect to X_1, X_2, X_3 and X_4 axes respectively. Value 0 will indicate that a reflection won't be applied.

See in the **Table 5.19** the array representation for the composition of transformations of the examples presented in **Table 5.18**.

TABLE 5.19

Examples of 4D combinations Cn_1 and Cn_2 and the transformations to apply, in their respective array representation, such that $Cn_1 \equiv T^n(Cn_2)$ (own elaboration).

Example	***Cn_1* (Binary Representation)**	**T^n (Array Representation)**	***Cn_2* (Binary Representation)**
1	1001100000000000	{2, 0, 0, 2, 1, 0, 0}	0010001010000000
2	1110000000000000	{2, 2, 0, 2, 0, 0, 0}	0000000000000111
3	1001100000000000	{2, 3, 0, 2, 1, 0, 0}	0011000001000000
4	1001100010000000	{3, 3, 0, 0, 0, 0, 1}	1011100000000000
5	0011010010000000	{3, 2, 1, 0, 1, 0, 1}	0100100000001001

If there exists a composition such that $configuration \equiv T^n(combination)$ then the function returns true; otherwise, all possible compositions were evaluated and none of them indicate an equivalence between the combination and the configuration from the input.

```
boolean existsComposition(BinaryString combination, BinaryString configuration)
{
    int composition[7] = {0,0,0,0,0,0,0};
    for(i = 1; i < 20480; i++)
    {
        BinaryString cn = combination.clone( );
        applyComposition(composition, cn);
        if(equals(cn, configuration) == true)
            return true;
        getNextComposition(composition);
    }
    return false;
}
```

The function *applyComposition* applies a composition of geometric transformations to a combination of hyper-boxes (*cn* in the code). The application of a set of geometric transformations on a configuration which is represented through a binary string is a very simple process. Each one of the *hyper-boxes* (i.e. occupied hyper-octants) in a configuration is referred through their descriptive axes. Now, each one of the hyper-octant's axes will be associated with a coordinate, if we are considering an axis' negative part then the corresponding coordinate will have a value equal to –1; otherwise, the corresponding coordinate is equal to one. By this way, a hyper-octant will be related with a point in the 4D

space. For example, the corresponding point for the hyper-octant $\overline{x}_1 x_2 \overline{x}_3 x_4$ is (-1,1,-1,1). **Table 5.20** shows the hyper-octants and their corresponding points.

TABLE 5.20
The 4D space's hyper-octants and their corresponding points (own elaboration).

Hyper-octant (Descriptive Axes)	**Corresponding Point**
$x_1 x_2 x_3 x_4$	(1, 1, 1, 1)
$x_1 x_2 x_3 \overline{x}_4$	(1, 1, 1,-1)
$x_1 x_2 \overline{x}_3 x_4$	(1, 1,-1, 1)
$x_1 x_2 \overline{x}_3 \overline{x}_4$	(1, 1,-1,-1)
$x_1 \overline{x}_2 x_3 x_4$	(1,-1, 1, 1)
$x_1 \overline{x}_2 x_3 \overline{x}_4$	(1,-1, 1,-1)
$x_1 \overline{x}_2 \overline{x}_3 x_4$	(1,-1,-1, 1)
$x_1 \overline{x}_2 \overline{x}_3 \overline{x}_4$	(1,-1,-1,-1)
$\overline{x}_1 x_2 x_3 x_4$	(-1, 1, 1, 1)
$\overline{x}_1 x_2 x_3 \overline{x}_4$	(-1, 1, 1,-1)
$\overline{x}_1 x_2 \overline{x}_3 x_4$	(-1, 1,-1, 1)
$\overline{x}_1 x_2 \overline{x}_3 \overline{x}_4$	(-1, 1,-1,-1)
$\overline{x}_1 \overline{x}_2 x_3 x_4$	(-1,-1, 1, 1)
$\overline{x}_1 \overline{x}_2 x_3 \overline{x}_4$	(-1,-1, 1,-1)
$\overline{x}_1 \overline{x}_2 \overline{x}_3 x_4$	(-1,-1,-1, 1)
$\overline{x}_1 \overline{x}_2 \overline{x}_3 \overline{x}_4$	(-1,-1,-1,-1)

To apply the set of required transformations we must then obtain the corresponding points of each occupied hyper-octant and transform them according to the needed rotations and/or reflections. Since the possible rotation's angles are 90°, 180° and 270° and the reflections to apply will preserve the values of the coordinates in –1 or 1, is that the transformed points' coordinates will have a new hyper-octant associated and therefore a *hyper-box* has been placed in a new position; finally a new string that represents the transformed configuration is obtained.

For example, consider the string 0100011001110000 whose *hyper-boxes* are distributed in the 4D space as has been shown in the **Table 5.16**. In the **Table 5.21** we show the points associated to each one of its occupied hyper-octants.

TABLE 5.21
The points associated to the occupied *hyper-octants* of a 4D configuration (from Table 5.16; own elaboration).

Hyper-octants	**Corresponding Points**
$x_1 x_2 x_3 \bar{x}_4$	(1, 1, 1,-1)
$x_1 \bar{x}_2 x_3 \bar{x}_4$	(1,-1, 1,-1)
$x_1 \bar{x}_2 \bar{x}_3 x_4$	(1,-1,-1, 1)
$\bar{x}_1 x_2 x_3 \bar{x}_4$	(-1, 1, 1,-1)
$\bar{x}_1 x_2 \bar{x}_3 x_4$	(-1, 1,-1, 1)
$\bar{x}_1 x_2 \bar{x}_3 \bar{x}_4$	(-1, 1,-1,-1)

Now, we will apply the compositions of transformations {1,1,0,1,0,0,0}, i.e. $R_{1,4}(90°) \cdot R_{2,4}(90°) \cdot R_{1,2}(90°)$. The 90° rotations around X_1X_2, X_1X_3, X_1X_4, X_2X_3, X_2X_4 and X_3X_4 planes and the reflections respect X_1, X_2, X_3 and X_4 axes are easily defined in the **Table 5.22**.

TABLE 5.22
Main rotations around 90° and reflections in the 4D space (own elaboration).

$R_{1,2}$(90°)	**$R_{1,3}$(90°)**	**$R_{1,4}$(90°)**
$X_1' = X_1$ $X_2' = X_2$ $X_3' = X_4$ $X_4' = -X_3$	$X_1' = X_1$ $X_2' = X_4$ $X_3' = X_3$ $X_4' = -X_2$	$X_1' = X_1$ $X_2' = -X_3$ $X_3' = X_2$ $X_4' = X_4$
$R_{2,3}$(90°)	**$R_{2,4}$(90°)**	**$R_{3,4}$(90°)**
$X_1' = -X_4$ $X_2' = X_2$ $X_3' = X_3$ $X_4' = X_1$	$X_1' = X_3$ $X_2' = X_2$ $X_3' = -X_1$ $X_4' = X_4$	$X_1' = -X_2$ $X_2' = X_1$ $X_3' = X_3$ $X_4' = X_4$

R_1	**R_2**	**R_3**	**R_4**
$X_1' = -X_1$ $X_2' = X_2$ $X_3' = X_3$ $X_4' = X_4$	$X_1' = X_1$ $X_2' = -X_2$ $X_3' = X_3$ $X_4' = X_4$	$X_1' = X_1$ $X_2' = X_2$ $X_3' = -X_3$ $X_4' = X_4$	$X_1' = X_1$ $X_2' = X_2$ $X_3' = X_3$ $X_4' = -X_4$

Then by applying the transformations $R_{1,4}(90°) \cdot R_{2,4}(90°) \cdot R_{1,2}(90°)$ on the points of the referred example, we have the new points and therefore the new occupied hyper-octants presented in **Table 5.23**. The new string that represents to the transformed configuration is then 0101001110000010 (the original string was 0100011001110000).

TABLE 5.23
Applying geometric transformations to the points associated to the occupied hyper-octants in a 4D configuration (from Table 5.16; own elaboration).

Points	**Transformations**	**Transformed Points**	**Corresponding Hyper-octants**
(1, 1, 1,-1)	$R_{1,4}(90°) \cdot R_{2,4}(90°) \cdot R_{1,2}(90°)$	(1,-1,-1, 1)	$x_1 \bar{x}_2 \bar{x}_3 x_4$
(1,-1, 1,-1)		(-1,-1,-1, 1)	$\bar{x}_1 \bar{x}_2 \bar{x}_3 x_4$
(1,-1,-1, 1)		(-1, 1, 1, 1)	$\bar{x}_1 x_2 x_3 x_4$
(-1, 1, 1,-1)		(1,-1,-1,-1)	$x_1 \bar{x}_2 \bar{x}_3 \bar{x}_4$
(-1, 1,-1, 1)		(1, 1, 1,-1)	$x_1 x_2 x_3 \bar{x}_4$
(-1, 1,-1,-1)		(1, 1,-1,-1)	$x_1 x_2 \bar{x}_3 \bar{x}_4$

Through this implementation we have found the 402 Hill's configurations for the 4D-OPP's. In the **Table 5.24** is shown the configurations' distribution while in **Appendix B** are shown the hyper-boxes' sets that are representatives of these 402 configurations.

TABLE 5.24
The distribution for the 402 Hill's Configurations in the 4D-OPP's (own elaboration).

4D Hyper-boxes	**Configurations**	**4D Hyper-boxes**	**Configurations**
0	1	16	1
1	1	15	1
2	4	14	4
3	6	13	6
4	19	12	19
5	27	11	27
6	50	10	50
7	56	9	56
8	74		

5.7.2 Obtaining the Aguilera & Pérez's Configurations for the 4D-OPP's

The configurations for the 4D-OPP's determined by Aguilera & Pérez were obtained by a very similar process to the used in the determination of the Hill's configurations (previous section). However, it was observed that the counting of each type of adjacencies for all the combinations that belong to a same configuration is the same. For example, in the **Table 5.25** are shown six 4D configurations whose adjacencies counting (which can be performed through the algorithm presented in section 5.6.2) is the same.

TABLE 5.25
Six 4D combinations with 6 hyper-boxes' whose adjacencies counting is the same (own elaboration).

Hyper-boxes' combinations	Number of Volume Adjacencies	Number of Face Adjacencies	Number of Edge Adjacencies	Number of Vertex Adjacencies
0011110110000000 1001011110000000 0110101011000000 1001101011000000 0101011011000000 1000000111101000	4	6	4	1

By this way we can say that a hyper-boxes' combination Cn_1 will be equivalent to other combination Cn_2 (i.e. they belong to the same equivalence class) if their adjacencies counting is equal, such that:

$$Cn_1 \equiv Cn_2 \quad iff \quad \begin{cases} volume\_adjacencies(Cn_1) & = & volume\_adjacencies(Cn_2) & and \\ face\_adjacencies(Cn_1) & = & face\_adjacencies(Cn_2) & and \\ edge\_adjacencies(Cn_1) & = & edge\_adjacencies(Cn_2) & and \\ vertex\_adjacencies(Cn_1) & = & vertex\_adjacencies(Cn_2) & \end{cases}$$

The implementation to find the configurations for the 4D-OPP's by according this evaluation is similar to the presented in previous section. The algorithm will find the 4D-OPP's configurations in the following way:

- We will have a set *configurations* that contains all the adjacencies counting of each one of the identified equivalence classes.

- The adjacencies counting of a *combination* of hyper-boxes (obtained through the algorithm presented in section 5.6.2) will be evaluated with each one of the adjacencies counting in *configurations* in order to determine if that *combination* can be a representative of a new configuration. If there exists an adjacencies counting in *configurations* equal to the combination's counting then it has a representative in configurations; otherwise, the *combination* of hyper-boxes is the representative of a new configuration and its adjacencies counting must be added to the set *configurations*.

- The implementation receives as input the number N of hyper-boxes whose configurations will be determined. The output will be the set *configurations* that contains all the adjacencies counting of all the configurations with N hyper-boxes.

```
Vector getConfigurations(int N)
{
    Vector configurations = new Vector( );
    BinaryString combination = 0000000000000000; //Specific for 4D space.
    for(int i = 0; i < 65536; i++)
    {
        if(getNumberOfHyperBoxes(combination) == N)
        {
            int combinationAdjacencies[ ] = analyzeAdjacencies(combination, 4);
            for(int j = 0; j < confs; j++)
            {
                int configurationAdjacencies[ ] = configurations.elementAt(j);
                if(equals(combinationAdjacencies, configurationAdjacencies) == true)
                    break;
            }
            if(j == confs)
            configurations.add(combinationAdjacencies);
        }
        getNextCombination(combination);
    }
    return configurations;
}
```

Through this implementation Aguilera & Pérez [Pérez, 01] found 253 configurations for the 4D-OPP's. In the section 4.2.3 is shown the configurations' distribution while in **Appendix A** are shown the adjacencies counting for these 253 configurations. Moreover, the configurations for the 5D and 6D-OPP's found through the 'Test-Box' Heuristic (section 5.2) were determined according to their adjacencies counting.