Chapter 6
Algorithms in the nD-EVM and their Performance

In this chapter we will introduce some basic algorithms that perform operations between nD-OPP's represented through the nD-EVM. We will specify aspects related to their implementation. In [Aguilera98] were described a set of primitive algorithms that should be present in a system that implements the nD-EVM. In our case we will start from those defined algorithms. It is natural to ask about the performance, or in more detailed words, about the time complexity of these algorithms. In this work we will deal with this last topic but from a statistical point of view. The bounds we provide will be obtained from experimental data which were obtained according to procedures we mention with detail in the corresponding sections. Because our algorithms are recursive on the number of dimensions of the input polytopes and in each recursivity level a wide range of situations can be present is that a formal analysis for time complexity is above of the scope of this work. We expect our estimations can be useful in suggesting an expected result for formal temporal complexity analysis, or well, in providing to the reader clues about timings of algorithms when they are implemented.

This chapter is divided in the following sections: **Section 6.1** defines the basic algorithms to be considered for the manipulation of nD-EVM's. As commented before, these procedures were originally proposed in [Aguilera98] and here we are taking them as stating point. The **Section 6.1.1** provides details about the way we are storing and implementing EVM's. Such implementations are essential for the time statistical analysis discussed in **Sections 6.2** to **6.6** where we describe and test experimentally algorithms for Boolean Operations, computing of nD Content and computing of (n-1)D Content of nD-OPP's represented through the nD-EVM.

6.1. Basic Algorithms for the nD-EVM

In **Section 5.4** we stated that in this work we will assume that the coordinates of extreme vertices in the Extreme Vertices Model of an nD-OPP p, $EVM_n(p)$ are sorted according to coordinate X_1, then to coordinate X_2, and so on until coordinate X_n. That is, we are considering the only ordering $X_1 \ldots X_i \ldots X_n$, $1 < i \leq n$. According to **Sections 5.2** to **5.6** we can define the following primitive operations which are based in the functions originally presented in [Aguilera98] (for 2D and 3D cases) and they consider the ordering previously commented:

```
Output: An empty nD-EVM.
Procedure InitEVM( )
{       Returns the empty set.                                                  }
```

```
Input:          An (n-1)D-EVM hvl embedded in nD space.
Input/Output:   An nD-EVM p
Procedure PutHvl(EVM hvl, EVM p)
{       Appends an (n-1)D couplet hvl, which is perpendicular to X1-axis, to p.  }
```

```
Input:  An nD-EVM p
Output: An (n-1)D-EVM embedded in (n-1)D space.
Procedure ReadHvl(EVM p)
{       Extracts next (n-1)D couplet perpendicular to X1-axis from p.           }
```

```
Input:          Two vertices Vb and Ve.
Input/Output:   An nD-EVM p
Procedure PutBrink(Vertex Vb, Vertex Ve, EVM p)
{       Appends to an nD-EVM p a brink defined by its Extreme Vertices Vb and Ve.  }
```

```
Input:          An nD-EVM p
Output:         Two vertices Vb and Ve.
Procedure ReadBrink(EVM p)
{       Reads next brink (or pair of Extreme Vertices) from an nD-EVM p.                    }
```

```
Input:          An nD-EVM p
Output:         A Boolean.
Procedure EndEVM(EVM p)
{       Returns true if the end of p along X1-axis has been reached.                        }
```

```
Input/Output:  An (n-1)D-EVM p embedded in (n-1)D space.
Input:         A coordinate coord of type CoordType
               (CoordType is the chosen type for the vertex coordinates: Integer or Real)
Procedure SetCoord(EVM p, CoordType coord)
{       Sets the X1-coordinate to coord on every vertex of the (n-1)D couplet p.
        For coord = 0, it performs the projection π1(p).                                    }
```

```
Input: An (n-1)D-EVM p embedded in nD space.
Output: A CoordType (CoordType is the chosen type for the vertex coordinates: Integer or Real)
Procedure GetCoord(EVM p)
{       Gets the common X1 coordinate of the (n-1)D couplet p.                              }
```

```
Input: Two nD-EVM's p and q.
Output: An nD-EVM
Procedure MergeXor(EVM p, EVM q)
{  Applies the Exclusive OR operation to the vertices of p and q and returns the resulting set. }
```

Function MergeXor performs an XOR between two nD-EVM's, that is, it keeps all vertices belonging to either $EVM_n(p)$ or $EVM_n(q)$ and discards any vertex that belongs to both $EVM_n(p)$ and $EVM_n(q)$. Since the model is sorted, this function consists on a simple merging-like algorithm, and therefore, it runs on linear time [Aguilera98]. Its complexity is given by $O(Card(EVM_n(p)) + Card(EVM_n(q))$ since each vertex from $EVM_n(p)$ and $EVM_n(q)$ needs to be processed just once. Moreover, according to **Theorem 5.19**, the resulting set corresponds to the regularized XOR operation between p and q since

$$EVM_n(p \otimes^* q) = EVM_n(p) \ \otimes \ EVM_n(q)$$

From the above primitive operations and [Aguilera98], the **Algorithms 6.1** and **6.2** may be easily derived.

```
Input: An (n-1)D-EVM corresponding to section S.
       An (n-1)D-EVM corresponding to couplet hvl.
Output: An (n-1)D-EVM.
Procedure GetSection(EVM S, EVM hvl)
       // Returns the projection of the next section of an nD-OPP whose previous section is S.
       return MergeXor(S, plv)
end-of-procedure
```

Algorithm 6.1. Computing $EVM_{n-1}\left(\pi_1(S_k^i(p))\right)$ as $EVM_{n-1}\left(\pi_1(S_{k-1}^i(p))\right) \otimes EVM_{n-1}\left(\pi_1(\Phi_k^i(p))\right)$ (by **Corollary 5.8**).

```
Input: An (n-1)D-EVM corresponding to section Si.
       An (n-1)D-EVM corresponding to section Sj.
Output: An (n-1)D-EVM.
Procedure GetHvl(EVM Si, EVM Sj)
       // Returns the projection of the couplet between consecutive sections Si and Sj.
       return MergeXor(Si, Sj)
end-of-procedure
```

Algorithm 6.2. Computing $EVM_{n-1}\left(\pi_1(\Phi_k^i(p))\right) = EVM_{n-1}\left(\pi_1(S_{k-1}^i(p))\right) \otimes EVM_{n-1}\left(\pi_1(S_k^i(p))\right)$ (by **Corollary 5.7**).

The **Algorithm 6.3** computes the sequence of sections of an nD-OPP p from its nD-EVM using the previous functions [Aguilera98]. It sequentially reads the projections of the (n-1)D couplets *hvl* of the polytope p. Then it computes the sequence of sections using function *GetSection*. Each pair of sections S_i and S_j (the previous and next sections about the current *hvl*) is processed by a generic processing procedure (called *Process*), which performs the desired actions upon S_i and S_j (Note that some processes may only need one of such sections).

```
Input: An nD-EVM p.
Procedure EVM_to_SectionSequence(EVM p)
        EVM hvl         // Current couplet.
        EVM Si, Sj      // Previous and next sections about hvl.
        hvl = InitEVM( )
        Si = InitEVM( )
        Sj = InitEVM( )
        hvl = ReadHvl(p)
        while(Not(EndEVM(p)))
                Sj = GetSection(Si, hvl)
                Process(Si, Sj)
                Si = Sj
                hvl = ReadHvl(p)              // Read next couplet.
        end-of-while
end-of-procedure
```

Algorithm 6.3. Computing the sequence of sections from an nD-OPP p represented through the nD-EVM.

6.1.1. About the nD-EVM Implementation

6.1.1.1. The Trie Tree Data Structure

Usually procedures as searching in trees are based in comparisons between the values of their keys. A trie tree is a data structure that uses the way the keys are represented, in this case, as sequences of characters or digits, in order to guide procedures as searching through the structure. The name of the trie tree, coined by [Friendkin60], was assigned because it is contained in "information retrieval".

A trie tree is an m-ary tree. The order of a trie is determined by the base used to represent the values of its keys. For example, if its keys are represented through digits then the base and order is 10; if its keys are represented through alphabetical characters then its order is 26. Each node in a trie of order m is, in its original definition by [Friendkin60], an array of m pointers. Each element in the arrays corresponds to one of the elements in the base of the keys. The position of a pointer in the node determines its corresponding value in the base. The height of a trie is determined by the length of its keys. For a node P in the j-th level, in a 10-ary trie, P_i points to a subtree that represents to all the values of keys whose j-th digit is i. For example, P_4, in the sixth level of a 10-ary trie, points to a subtree that represents to all the values of keys whose sixth digit is 4.

Consider the following example of a trie whose keys are numbers in base 4 with four digits. The keys introduced in the structure are 1112, 1113, 2210, 3003 and 3102. See the **Figure 6.1**. The structure is a 4-ary trie and also has a height equal to four; each level is given by the position of each digit in the keys.

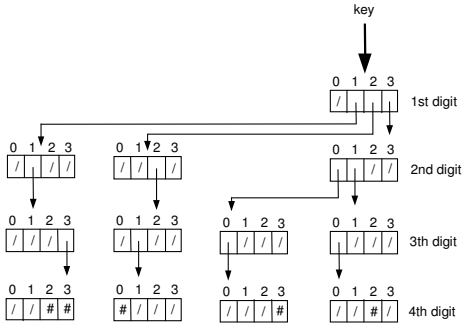


Figure 6.1. A trie tree for storing the keys 1112, 1113, 2210, 3003 and 3102 in base 4.

Originally trie trees were proposed as structures for storing file indexes [Loomis89] but they can be used to store and represent sets of data. In this later case, each leaf node will contain empty positions that indicate the absence of the corresponding value. See in our previous example (**Figure 6.1**), the character '#' is used to indicate the presence of a value in leaf nodes while character '/' indicates the absence of a value or a null pointer in the case of nodes in first, second and third levels.

Searching in a trie must finish in the leaf nodes. To determine the existence of a key in the structure it is required to visit all the levels in the tree. In each level, the ramification to follow is determined by the pertinent digit in the key. Hence, the length of a successful searching is determined by the height of the trie, which is based in the length of the keys. In our example from **Figure 6.1** a successful searching requires to visit four nodes, a value that is independent of the number of keys represented by the trie. In the other hand, a non-successful searching finishes when one of the digits in the key is not present in the structure. In this case, a non-successful searching can finish in any level of the structure. For example, by visiting the root node in our example we can infer that there are no keys whose first digit is zero.

The insertion of new keys in a trie is a direct process. The correct position in a node for representing a new digit is located by direct searching. When the position is located then it is changed from null to pointer, or in the second case, the pointer present in that position is followed to access next level in the structure. When a new pointer is added then a new leaf node is also added in order to direct a searching to it. Consider for example the adding of the key 1320 to our trie from **Figure 6.1**. In the first level we found that there are yet stored keys with first digit equal to one (**Figure 6.2.a**). In the second level we have that position three in the node is null, hence a new pointer is created and a new leaf node in the third level is attached to it (**Figure 6.2.b)**. In the third level obviously all the positions in the new node are null, hence, its position two is modified to store a pointer and a new leaf node is added in level four. Because this is the last digit in the key then the position zero in the new node is modified to indicate the presence of the new value (**Figure 6.2.c).**

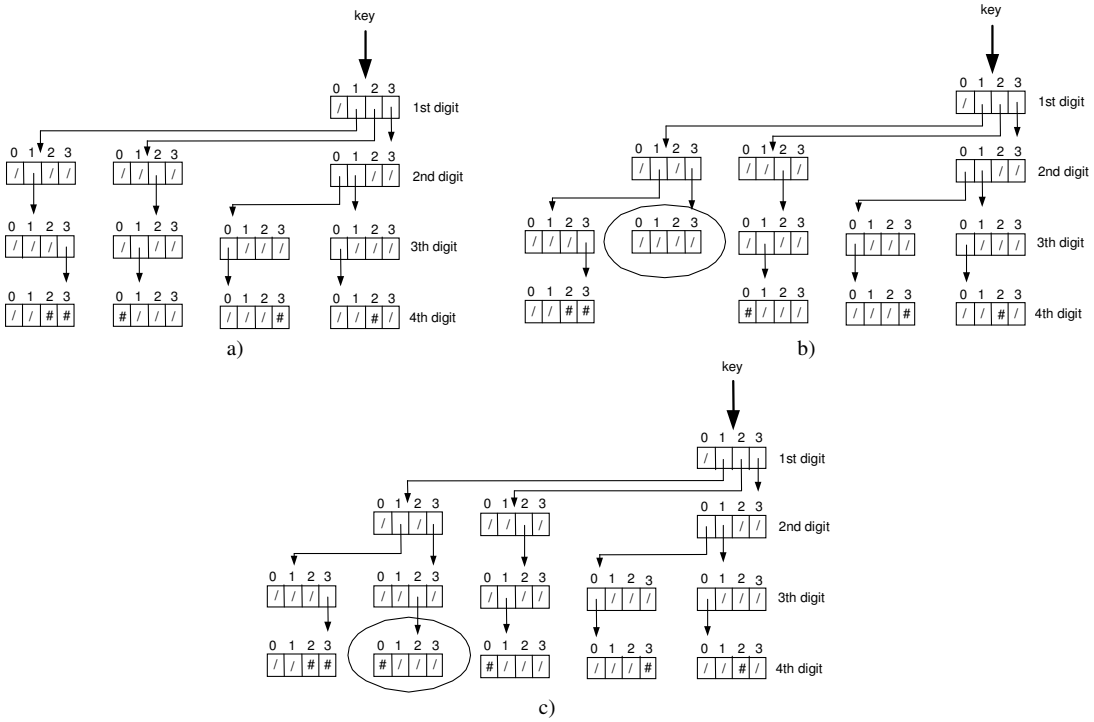


Figure 6.2. Inserting the key 1320 in the trie tree from **Figure 6.1**. b) Adding a new node in the third level. c) Adding the final leaf node in the fourth level.

At this point the reader can have detected a problem related to tries' storage requirements. Consider for example the following case: suppose that we have 250 keys with 9 digits each one in base 10. The trie will be a 10-ary tree with nine levels and potential space for 10^9 keys, but we are using only 0.000025% of these potential positions. Moreover, it can be observed that in our example from **Figure 6.2.c** we have positions in the node pointing to null or with absent values. In this sense, a solution, provided originally in [Maly76], propose to consider each node in the structure not as an array but as a sorted linked list. The elements in such linked lists contain three fields:

- One field contains the value of the key in the corresponding level.
- A pointer to the next element in the list in the same level.
- A pointer to the following level.

In this case the structure contains only the values of the keys that it has stored. There is no space reserved, as in **Figures 6.1** and **6.2**, for potential new keys. If a new key is stored then only the required elements in each level are added. By applying this idea we have the trie presented in **Figure 6.2.c** has now the structure shown in **Figure 6.3**.

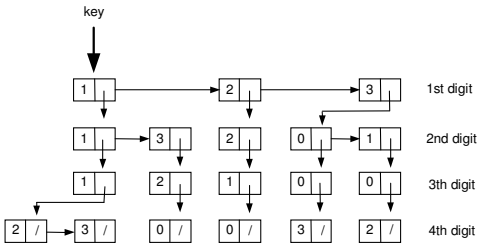


Figure 6.3. The trie tree from **Figure 6.2.c** by considering the ignoring of empty nodes and null pointers.

With the above modification, searching and adding of keys is slightly modified. It is preserved that searching and adding is computed in constant time which depends on the number of the length of its keys [Maly76]. It is important to recall that this bound for time is valid if the length of the keys is constant [Bodon04]. This approach for trie trees reduces the storage requirements by using only the necessary pointers.

6.1.1.2. Representing an nD-EVM in a Trie Tree

An Extreme Vertex can be seen as a key with length n. Each one of its coordinates in this case corresponds to each one of its "digits". The base of the keys is given by the number of distinct coordinates present in the nD-EVM where such vertex is contained. Consider for example the set of extreme vertices in a 4D unit hypercube c. Because we are considering, as stated in **Chapter 5**, that the coordinates of vertices are sorted according to coordinate X_1, then to coordinate X_2, and so on until coordinate X_4, hence we have:

$$EVM_4(c) = \{(0,0,0,0), (0,0,0,1), (0,0,1,0), (0,0,1,1), (0,1,0,0), (0,1,0,1), (0,1,1,0), (0,1,1,1),$$
$$(1,0,0,0), (1,0,0,1), (1,0,1,0), (1,0,1,1), (1,1,0,0), (1,1,0,1), (1,1,1,0), (1,1,1,1)\}$$

Therefore, our keys have length n = 4 and the order is given by m = 2 (the number of distinct coordinates in $EVM_4(c)$). Now we will proceed to introduce these points, or "keys", in a trie tree in such way that each one of its nodes stores their corresponding X_i-coordinate, or "digit". Moreover, that structure have a height given by n = 4 levels. See **Figure 6.4**.

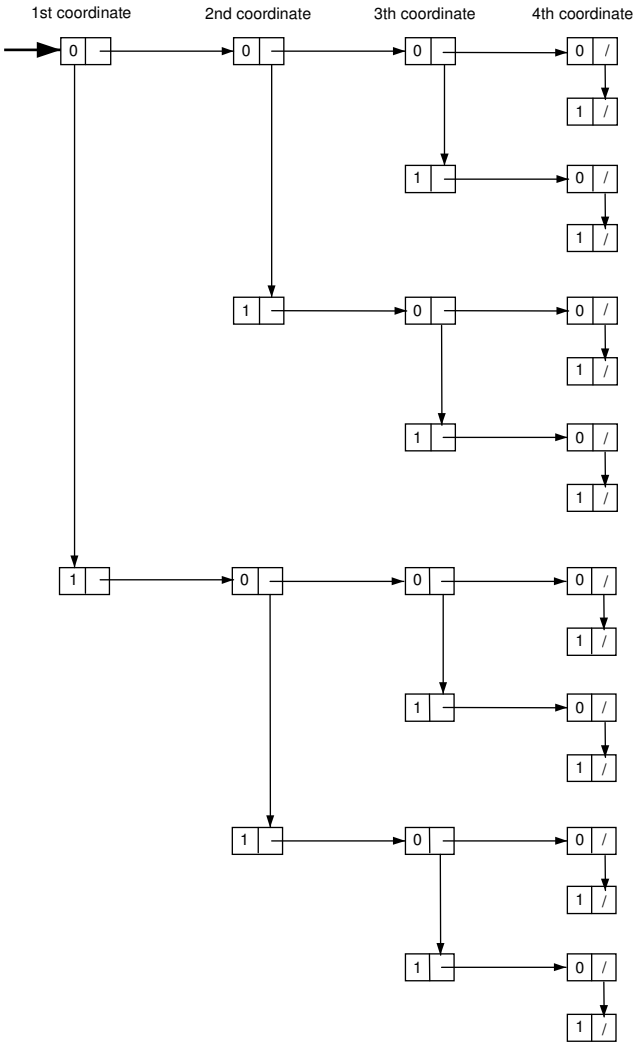


Figure 6.4. The trie tree associated to the EVM of a 4D unit hypercube.

The node 0 in the first level points to a subtree that represents to all the values of extreme vertices whose first coordinate, or "digit", is 0. In a similar way, node 1 in the same level points to the subtree that represents to all the values of extreme vertices whose first coordinate is 1. The first of these two referred subtrees contains the vertices embedded in the first couplet perpendicular to X_1-axis, i.e. $\Phi_1^1(c)$; while the second subtree contains the vertices embedded in the second couplet perpendicular to X_1-axis, i.e., $\Phi_2^1(c)$.

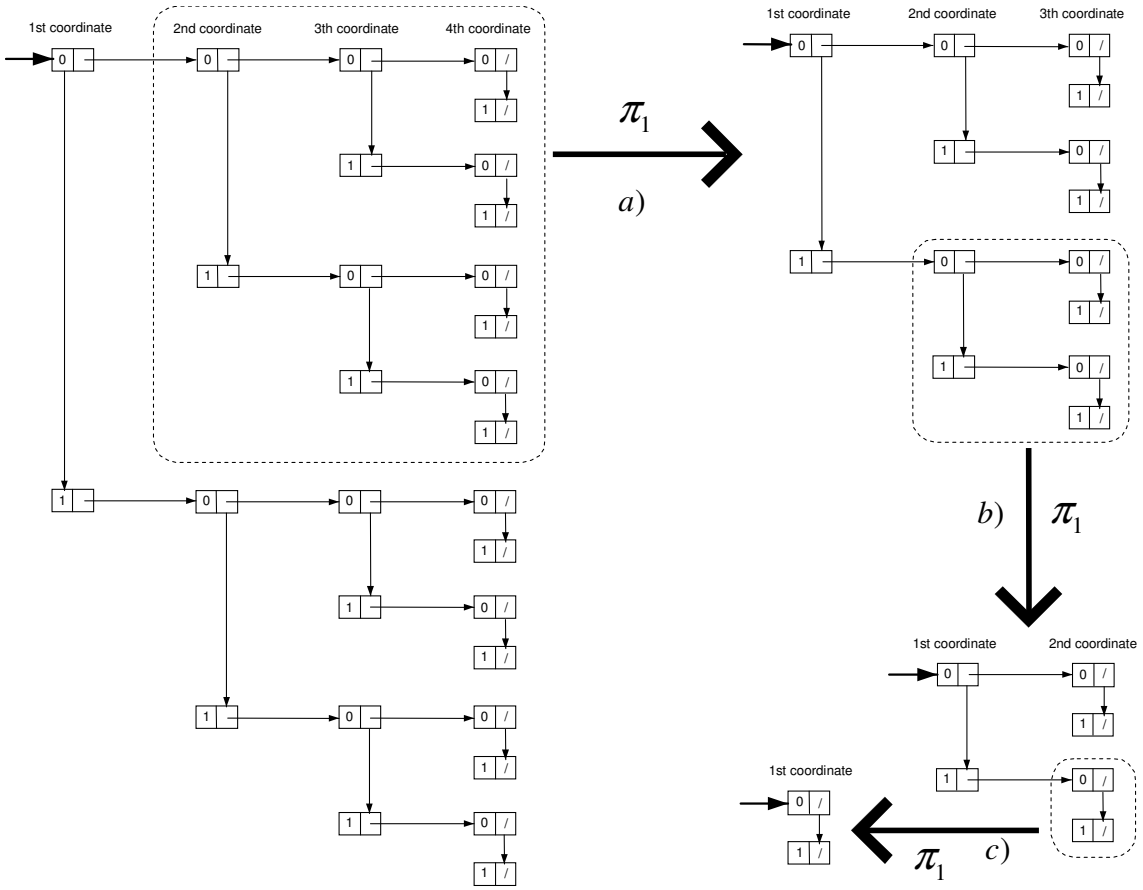


Figure 6.5. Extracting kD couplets from the trie tree associated to the EVM in a 4D unit hypercube.
a) Extracting the projection of a 3D couplet. b) Extracting the projection of a 2D couplet. c) Extracting a brink.

In **Chapter 5** we commented that in fact the couplets are themselves (n-1)D-OPP's, 3D-OPP's in this case, embedded in 4D space. By applying the projection operator π_1 we get the projection of such couplet in a 3D hyperplane perpendicular to X_1-axis, hence $\pi_1(\Phi_1^1(c))$ is a 3D-OPP embedded in 3D space and obviously its points contain three coordinates. The 3D-OPP $\pi_1(\Phi_1^1(c))$ is present in the trie tree from **Figure 6.4**. Consider again node 0 in the trie's first level. We commented before that such node points to the subtree that contains those extreme vertices whose first coordinate is zero. The operation $\pi_1(\Phi_1^1(c))$ suppress precisely that coordinate, hence, by extracting the subtree associated to node 0 first level we get the set of extreme vertices associated to $\pi_1(\Phi_1^1(c))$. Such extracted subtree is now a trie with height 3 (See **Figure 6.5**).

At this point it is clear that the above procedure of extraction of a subtree leads to a process where at the same time it is possible to extract the projection of the couplets from $\pi_1(\Phi_1^1(c))$. This process can descend until we extract subtries with one level which corresponds to extreme vertices associated to brinks. At this point, a trie represents the EVM of a 1D-OPP's and its structure corresponds to a simple connected linked list.

Trie trees as a way for representing nD-EVM's provide us an immediate access to couplets as shown in the previous example. In fact, according to the operation to perform, a copy of an extracted subtrie could be not necessary and only a pointer to the root of the subtrie would be sufficient. In this sense, algorithms PutHvl (appending an couplet), ReadHvl (extracting an couplet), PutBrink (adding a brink) and ReadBrink (extracting a brink) which were presented in **Section 6.1** can be implemented taking in account this tree structure. At this point is important to mention that the way trie trees represent EVM's, and their vertices, was previously identified by [Aguilera98] in the context of data compression schemes for EVM in 3D space.

To perform the Regularized Xor Boolean operation according to **Theorem 5.19**, that is, $EVM_n(p \otimes^* q) = EVM_n(p) \otimes EVM_n(q)$, by assuming that our nD-EVM's are stored in trie trees we can proceed as follows:

- Copy the trie tree associated to $EVM_n(p)$. Such copy trie at the end of the process will correspond to the trie associated to $EVM_n(p \otimes^* q)$.
- Perform a Depth First Search in the trie tree associated to $EVM_n(q)$:
  - When a leaf node is reached we have identified the coordinates of one of the points in $EVM_n(q)$. This point is searched in the trie associated to $EVM_n(p \otimes^* q)$. If it is not present then it is added to the structure, otherwise it is removed from the trie corresponding to $EVM_n(p \otimes^* q)$.

Because the length of our keys is constant, then as mentioned before, searching, adding and deleting a vertex is performed in constant time. Hence, procedure MergeXor, mentioned in **Section 6.1**, can be implemented assuming EVM's are stored through trie trees and its execution time will preserve its linearity.

Consider for example the 3D-OPP's p and q presented in the **Figures 6.6.a** and **b** with their respective EVM's stored through trie trees of height 3 which are shown in **Figures 6.6.c** and **d**. The common points to both EVM's will not be present in the result of Xor operation between p and q. Such common vertices are shown in the trie trees. After performing Xor operation according to the procedure we have described we obtain the 3D-OPP corresponding to $EVM_3(p \otimes^* q)$. Such OPP and its trie tree are shown in **Figure 6.7**.

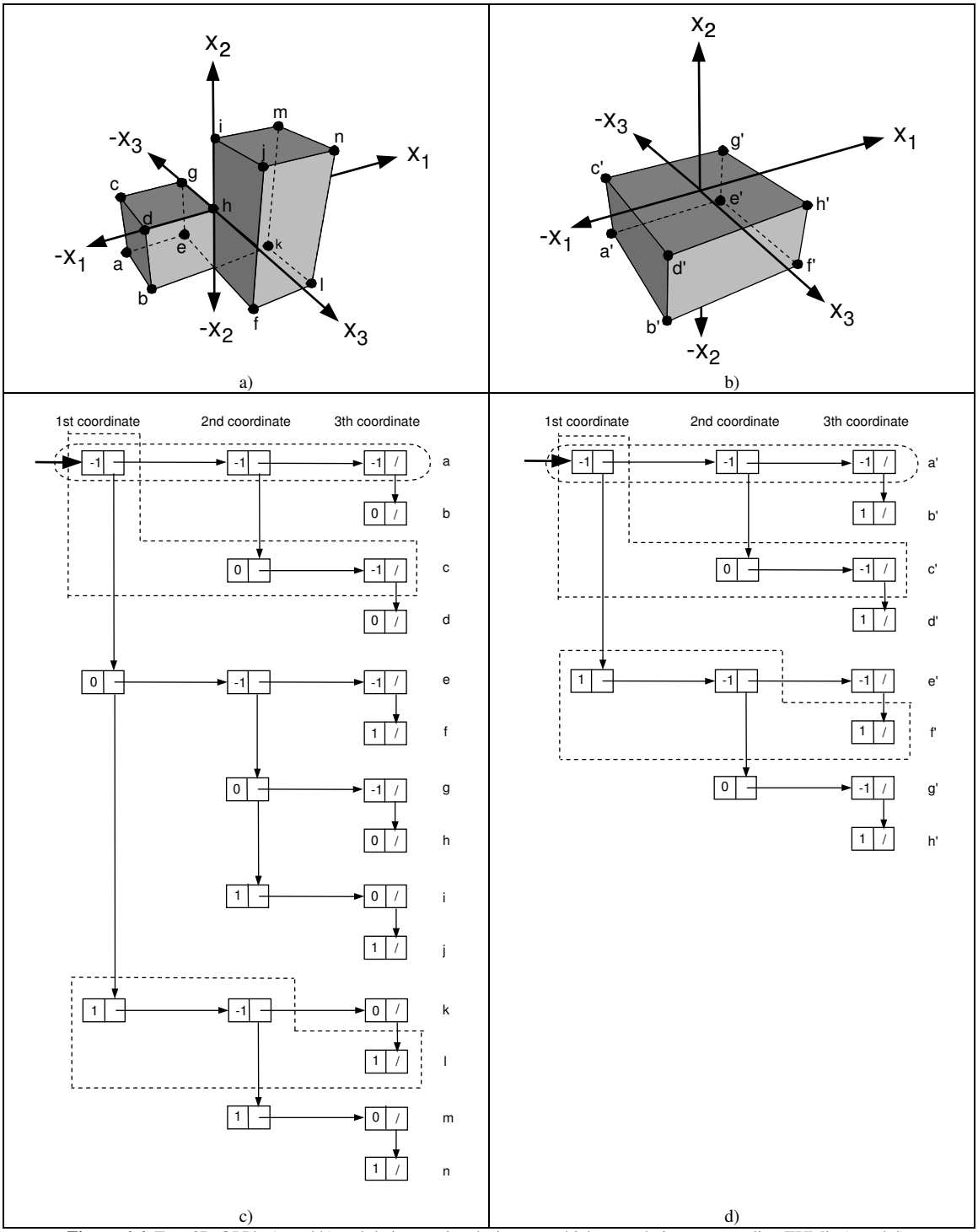


Figure 6.6. Two 3D-OPP's (a and b) and their associated trie trees which store their corresponding EVM's (c and d). The dotted lines indicates common extreme vertices to both OPP's.

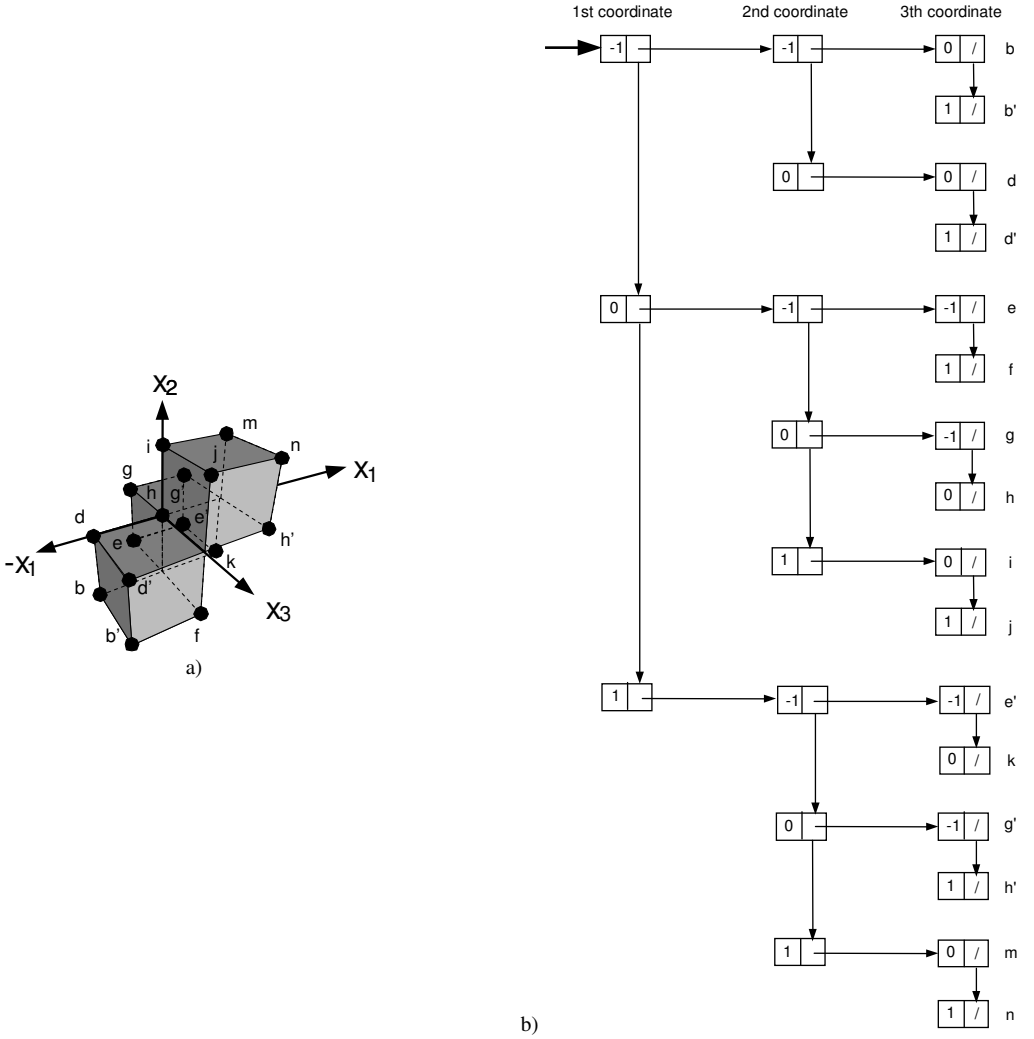


Figure 6.7. a) The 3D-OPP corresponding to $EVM_3(p \otimes^* q)$ in **Figure 6.6**. b) Its associated trie tree.

Following sections in this chapter will describe algorithms under the nD-EVM and we will analyze from a statistical point of view their execution time. The implementations of those procedures are based in the fact that the EVM's are stored using trie trees and the procedures described in this section.

6.2. The Boolean Operations Algorithm for the nD-EVM

This section describes the algorithm originally presented in [Aguilera98] for performing regularized Boolean operations. Let p and q be two nD-OPP's represented through the nD-EVM, and let op* be a Boolean operator in $\{\cup^*, \cap^*, -^*, \otimes^*\}$. The algorithm computes the resulting nD-OPP r = p op* q, and it is based on **Theorem 5.20**. Note that r = p $\otimes^*$ q can also be trivially performed using **Theorem 5.19**. The idea behind this algorithm is the following [Aguilera98]:

- The sequence of sections from p and q, perpendicular to X_i-axis, can be obtained first, based in **Theorem 5.17**.
- Then, according to **Corollary 5.6**, every section of r can recursively be computed as $S_k^i(r) = S_k^i(p)\ op^*\ S_k^i(q)$.
- Finally, r can be obtained from its sequence of sections, perpendicular to X_i-axis, according to **Theorem 5.16**.

Nevertheless, **Algorithm 6.4** does not work in this sequential form. It actually works in a wholly merged form in which it only needs to store one section for each of the operands p and q, and two consecutive sections for the result r. It also considers a unified grid partition $part_1(p \mid q)$ for both operands (See **Section 5.6.1**), assuming virtual couplets as needed.

```
Input: The nD-OPP's p and q represented through the nD-EVM.
       The number n of dimensions and the Boolean operation op.
Output: The output nD-OPP r, such that r = p op* q, codified through the nD-EVM.
Procedure BooleanOperation(EVM p, EVM q, BooleanOperator op, int n)
       EVM sP, sQ            // Current sections of p and q respectively.
       EVM hvl               // I/O couplet.
       boolean fromP, fromQ  // flags for the source of the couplet hvl.
       CoordType coord       // the common coordinate of couplets.
       EVM r, sRprev, sRcurr // nD-OPP r and two of its sections.
       If(n = 1) then // Basic case
              return BooleanOperation1D(p, q, op)
       else
              n = n - 1
              sP = InitEVM( )
              sQ = InitEVM( )
              sRcurr = InitEVM( )
              NextObject(p, q, coord, fromP, fromQ)
              While(Not(EndEVM(p)) and Not(EndEVM(q)))
                     If(fromP = true) then
                            hvl = ReadHvl(p)
                            sP = GetSection(sP, hvl)
                     end-of-if
                     If(fromQ = true) then
                            hvl = ReadHvl(q)
                            sQ = GetSection(sQ, hvl)
                     end-of-if
                     sRprev = sRcurr
                     sRcurr = BooleanOperation(sP, sQ, n, op) // Recursive call
                     hvl = GetHvl(sRprev, sRcurr)
                     SetCoord(hvl, coord)
                     PutHvl(hvl, r)
                     NextObject(p, q, coord, fromP, fromQ)
              end-of-while
              while(Not(EndEVM(p)))
                     hvl = ReadHvl(p)
                     PutBool(hvl, r, op)
              end-of-while
              while(Not(EndEVM(q)))
                     hvl = ReadHvl(q)
                     PutBool(hvl, r, op)
              end-of-while
              return r
       end-of-else
end-of-procedure
```

Algorithm 6.4. Computing Regularized Boolean Operations on the nD-EVM.

We describe some functions not defined in previous section [Aguilera98]:

- Function *BooleanOperation1D* performs 1D Boolean operations between p and q that are two 1D-OPP's.
- Procedure *NextObject* considers both input objects p and q and returns the common *coord* value of the next *hvl* to process, using function *GetCoord*. It also returns two flags, *fromP* and *fromQ*, which signal from which of the operands (both inclusive) is the next *hvl* to come.
- The main loop of procedure *BooleanOperation* gets couplets from p and/or q, using function *GetSection*. These sections are recursively processed to compute, according to **Corollary 5.6**, the corresponding section of r, *sRcurr*. Since two consecutive sections, *sRprev* and *sRcurr*, are kept, then the projection of the resulting *hvl*, is obtained by means of function *GetHvl* and then, it is correctly positioned by procedure *SetCoord*.
- When the end of one of the polytopes p or q is reached then the main iteration finishes, and the remaining couplets of the other polytope are either appended or not to the resulting polytope depending on the Boolean operation considered. Procedure *PutBool* performs this appending process.

6.2.1. Performance of Boolean Operations under the nD-EVM

6.2.1.1. A Note about the Experimental Complexity Analysis

In the following section we will present results related with execution times for **Algorithm 6.4** which performs Boolean Operations between two nD-OPP's represented through the nD-EVM. We proceed as follows:

- Our testing consider n = 2, 3, 4, 5.
- For each n we have generated 16,000 random nD-OPP's according to the following procedures:
  - Given two hypervoxelizations representing nD-OPP's g_1 and g_2 we obtain their respective nD-EVM's namely $EVM_n(g_1)$ and $EVM_n(g_2)$. According to **Theorem 5.1**, if a vertex is surrounded by an odd number of occupied hypervoxels then it is an Extreme Vertex. Thus, a hypervoxelization to nD-EVM conversion consists on collecting every vertex that belongs to and odd number of hypervoxels, and discarding the remaining vertices (In **Section 6.6** we will deal with detail the topic related to the conversion of the nD-EVM from and to other representation schemes).
  - Given the Regularized Boolean Operator op* we perform both g_1 op* g_2 and $EVM_n(g_1$ op* $g_2)$ according to the methodologies described in **Section 2.2.5** and **Section 5.6** respectively.
  - Let $EVM_n(r)$ be the output given by **Algorithm 6.4**, i.e., $EVM_n(r) = EVM_n(g_1$ op* $g_2)$. Let r' be the result provided by Boolean operation op* between hypervoxelizations of nD-OPP's g_1 and g_2. As a mechanism for controlling possible errors in our implementations we obtain $EVM_n(r')$ and verify that all the 16,000 generated nD-OPP's satisfied $EVM_n(r') = EVM_n(r)$. The comparison $EVM_n(r') = EVM_n(r)$ is not considered in the recorded execution times.
- The considered Boolean operations are Regularized Intersection, Union and Xor. In the case corresponding to Xor operation we have tested the same 8,000 pairs of generated nD-OPP's with the Algorithm MergeXor, described in **Section 6.1**, in order to compare its efficiency with **Algorithm 6.4**.
- The units for the time measures presented in **Charts 6.1** to **6.11** are given in nanoseconds.
- The evaluations were performed in a computer with Intel Celeron Processor at 900 Mhz and 256 megabytes in RAM memory. This equipment was isolated from network connections, virus scanners and utilities for the management and maintenance of files. This isolation has the objective of avoiding as possible the execution of additional processes that could affect the execution time of our algorithms.
- The algorithms were implemented using the Java Programming Language under the Software Development Kit 1.5 provided by Sun Microsystems.
- As commented in **Section 6.1.1.2** our EVM's are stored and managed through trie trees. Our implemented algorithms consider this aspect.
- Once the generation of nD-OPP's has finished and the algorithms were evaluated we proceed with a statistical analysis in order to find a trendline of the form $t = ax^b$, where $x = Card(EVM_n(g_1)) + Card(EVM_n(g_2))$, that fits as good as possible to our measures in order to provide an estimation of the temporal complexity of the evaluated algorithms for each value of n. The quality of the approximation curve is assured by computing the R^2 value known as the coefficient of determination. It is well known that $R^2 \in [0, 1]$ and it reveals how closely the estimated values for the trendline correspond to our time measures [Burden04]. According to the literature, our trendlines are most reliable when its R^2 is at or near 1 [Wackerly01].
- In **Section 6.2.1.3**, and starting from the data presented in **Section 6.2.1.2** and their associated trendlines, we will propose an approximation surface for temporal complexity of **Algorithm 6.4** for each considered Boolean operation. Such surface which will be a function of two variables: the number x of Extreme Vertices in the input polytopes and the number n of dimensions.
- The trendlines and their coefficients of determination were computed using software Mathematica version 5.0.1, Wolfram Research. Approximation surfaces and their coefficients of determination were determined through software TableCurve 3D version 4.0.01, Systat Software.

6.2.1.2. The Time Complexity of the Boolean Operations Algorithm for n = 2, 3, 4, 5: An Experimental Analysis

We start by considering the case n = 2. Our generated 8,000 $Card(EVM_2(g_1)) + Card(EVM_2(g_2))$ have the following characteristics:

- $Max(Card(EVM_2(g_1)) + Card(EVM_2(g_2))) = 7{,}580$
- $Min(Card(EVM_2(g_1)) + Card(EVM_2(g_2))) = 136$
- $Mean(Card(EVM_2(g_1)) + Card(EVM_2(g_2))) = 4{,}822.5567$
- $Standard\_Deviation(Card(EVM_2(g_1)) + Card(EVM_2(g_2))) = 1{,}705.3379$

The **Charts 6.1** and **6.2** show the timings of **Algorithm 6.4** under Regularized Union, Intersection and Xor. The **Table 6.1** shows the equations of their associated trendlines. We will discuss our measures at the end of this section.

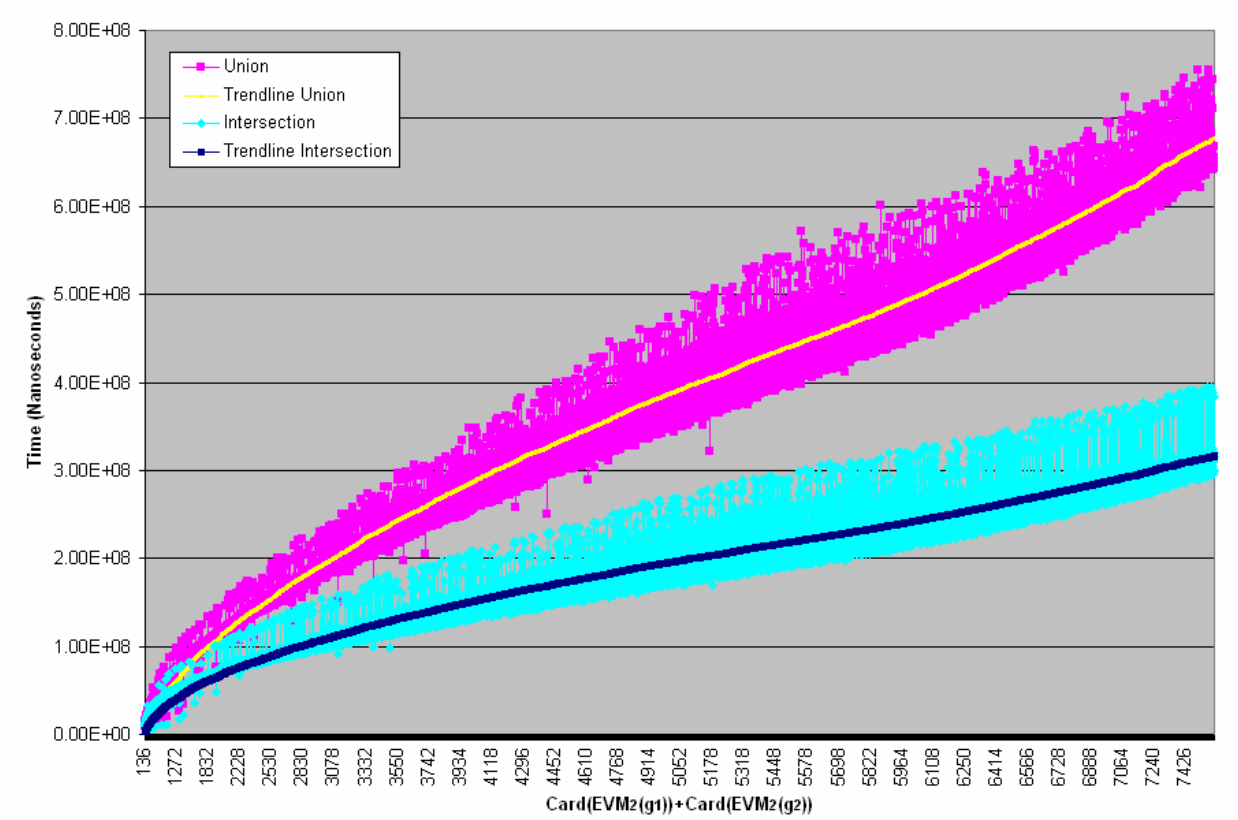


Chart 6.1. Execution times of **Algorithm 6.4** under 2D Regularized Union and Intersection.

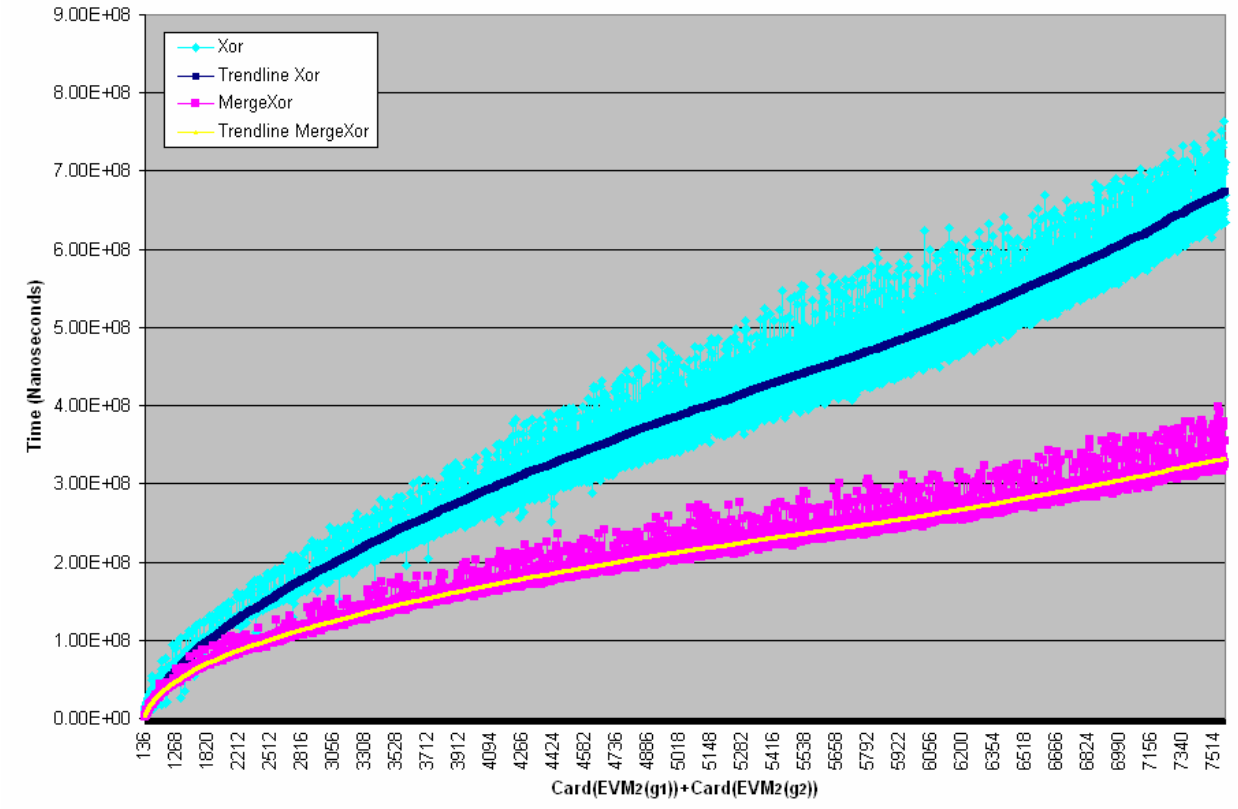


Chart 6.2. Comparing execution times of **Algorithm 6.4** and MergeXor Function under 2D Regularized Xor.

Operation	n	Trendline $t = ax^b$	a	b	R^2
Union	2	$t = 3{,}920.89x^{1.35026}$	3,920.89	1.35026	0.9709
Intersection	2	$t = 10{,}006.7x^{1.15994}$	10,006.7	1.15994	0.9030
Xor	2	$t = 4{,}033.48x^{1.34649}$	4,033.48	1.34649	0.9690
Xor (MergeXor)	2	$t = 20{,}883.7x^{1.08317}$	20,883.7	1.08317	0.9839

Table 6.1. Trendlines approximating the execution times for 2D Regularized Boolean Operations.

Now considering the case n = 3. Our generated 8,000 $Card(EVM_3(g_1)) + Card(EVM_3(g_2))$ have the following characteristics:

- $Max(Card(EVM_3(g_1)) + Card(EVM_3(g_2))) = 8{,}280$
- $Min(Card(EVM_3(g_1)) + Card(EVM_3(g_2))) = 216$
- $Mean(Card(EVM_3(g_1)) + Card(EVM_3(g_2))) = 5{,}599.993$
- $Standard\_Deviation(Card(EVM_3(g_1)) + Card(EVM_3(g_2))) = 1{,}728.84$

The **Charts 6.3** and **6.4** show the timings of **Algorithm 6.4** under 3D Regularized Union, Intersection and Xor. The **Table 6.2** shows the equations of their associated trendlines together with their respective coefficients of determination.

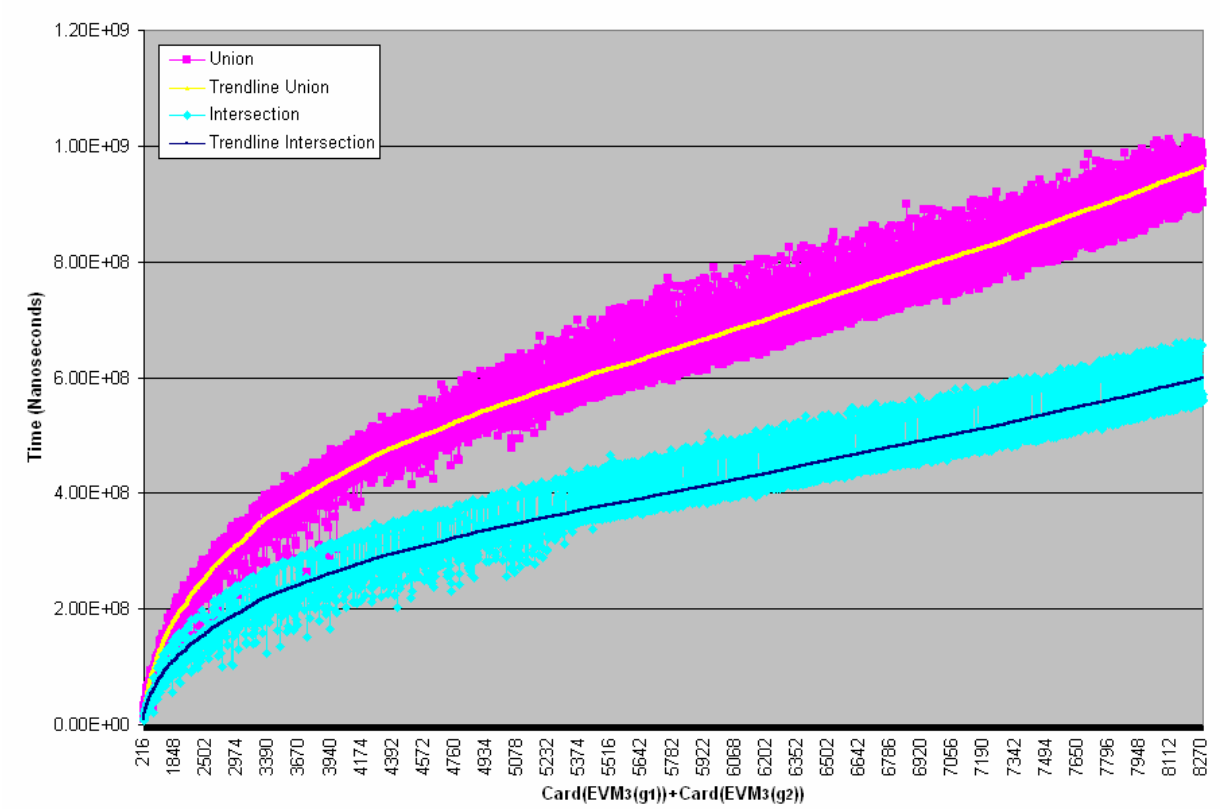


Chart 6.3. Execution times of **Algorithm 6.4** under 3D Regularized Union and Intersection.

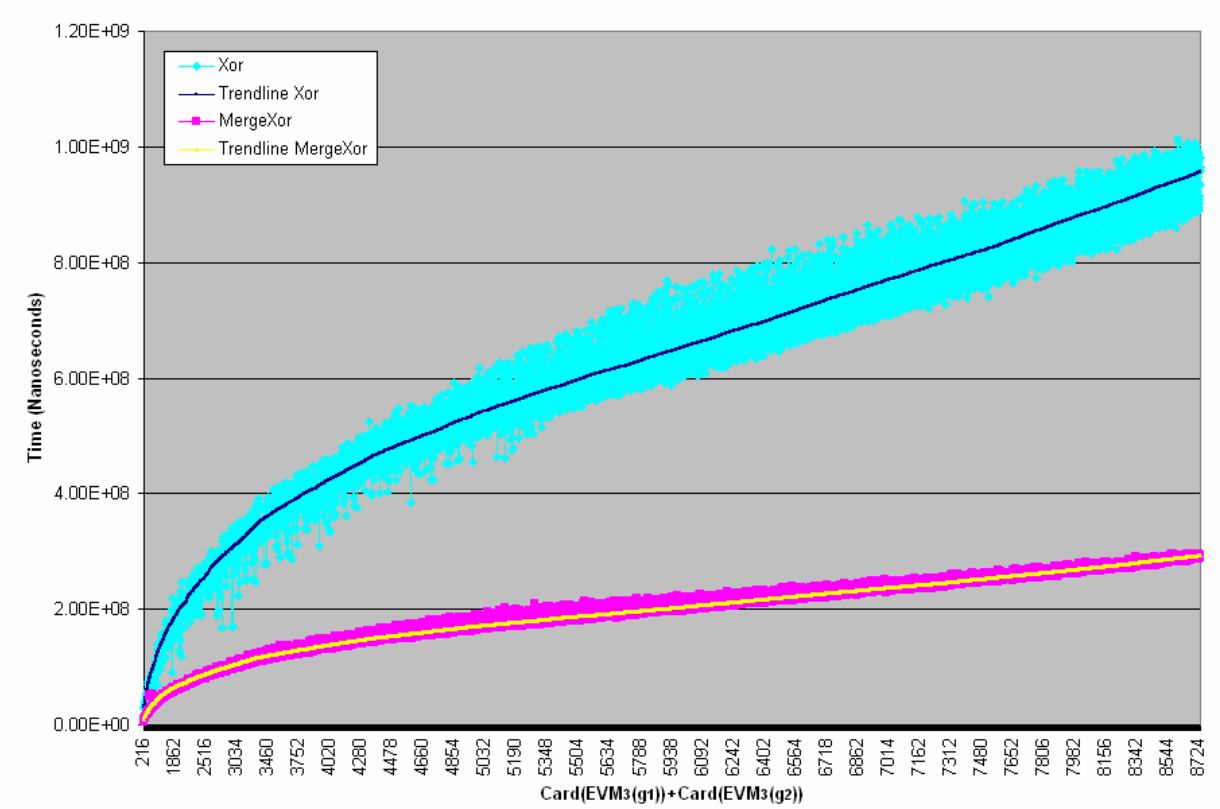


Chart 6.4. Comparing execution times of **Algorithm 6.4** and MergeXor Function under 3D Regularized Xor.

Operation	n	Trendline $t = ax^b$	a	b	R^2
Union	3	$t = 45{,}677.9x^{1.10379}$	45,677.9	1.10379	0.9726
Intersection	3	$t = 24{,}865.6x^{1.11831}$	24,865.6	1.11831	0.9426
Xor	3	$t = 49{,}920.9x^{1.09301}$	49,920.9	1.09301	0.9706
Xor (MergeXor)	3	$t = 46{,}208.3x^{0.96474}$	46,208.3	0.96474	0.9905

Table 6.2. Trendlines approximating the execution times for 3D Regularized Boolean Operations.

In the case n = 4 we have that the generated set of 4D-EVM's has the following values:

- $\text{Max}(\text{Card}(EVM_4(g_1)) + \text{Card}(EVM_4(g_2))) = 8{,}492$
- $\text{Min}(\text{Card}(EVM_4(g_1)) + \text{Card}(EVM_4(g_2))) = 48$
- $\text{Mean}(\text{Card}(EVM_4(g_1)) + \text{Card}(EVM_4(g_2))) = 5{,}792.8812$
- $\text{Standard\_Deviation}(\text{Card}(EVM_4(g_1)) + \text{Card}(EVM_4(g_2))) = 1{,}783.3989$

The **Charts 6.5** and **6.6** show the timings of **Algorithm 6.4** under 4D Regularized Union, Intersection and Xor. The **Table 6.3** shows the equations of their associated trendlines.

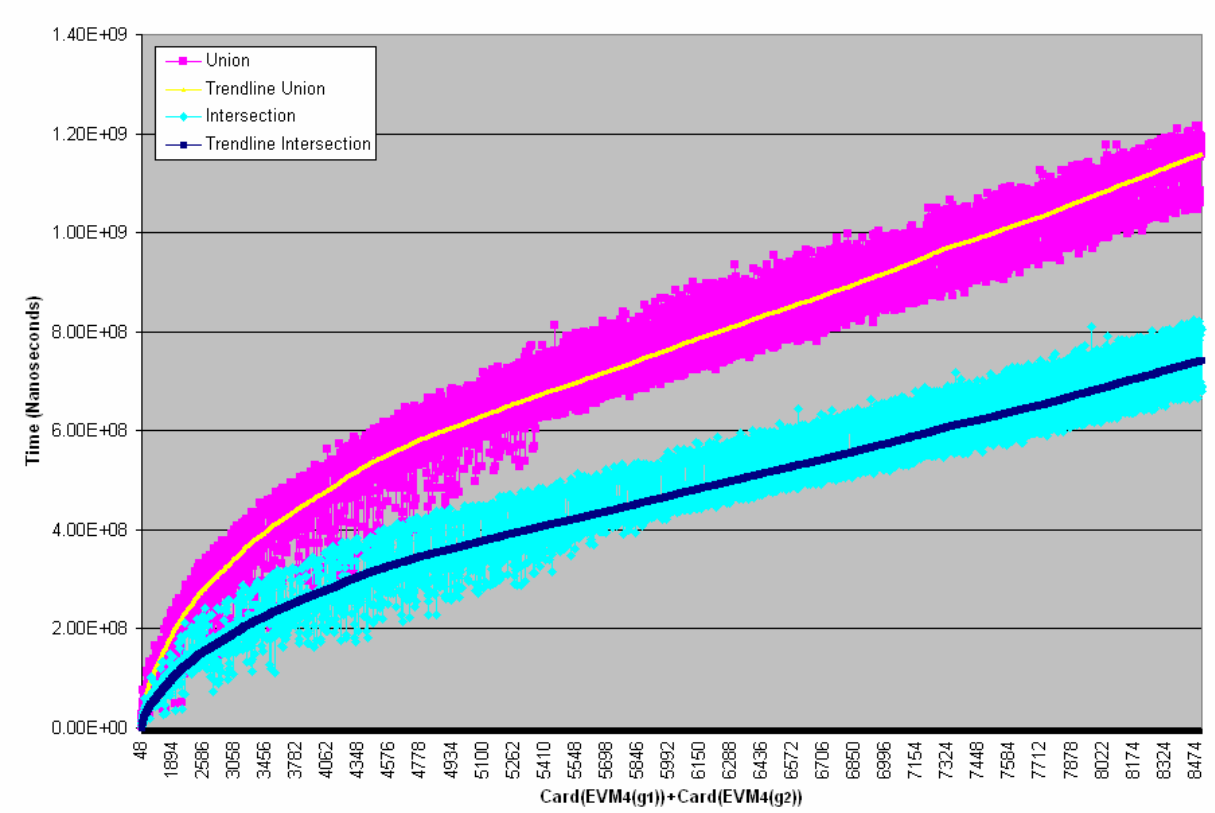


Chart 6.5. Execution times of **Algorithm 6.4** under 4D Regularized Union and Intersection.

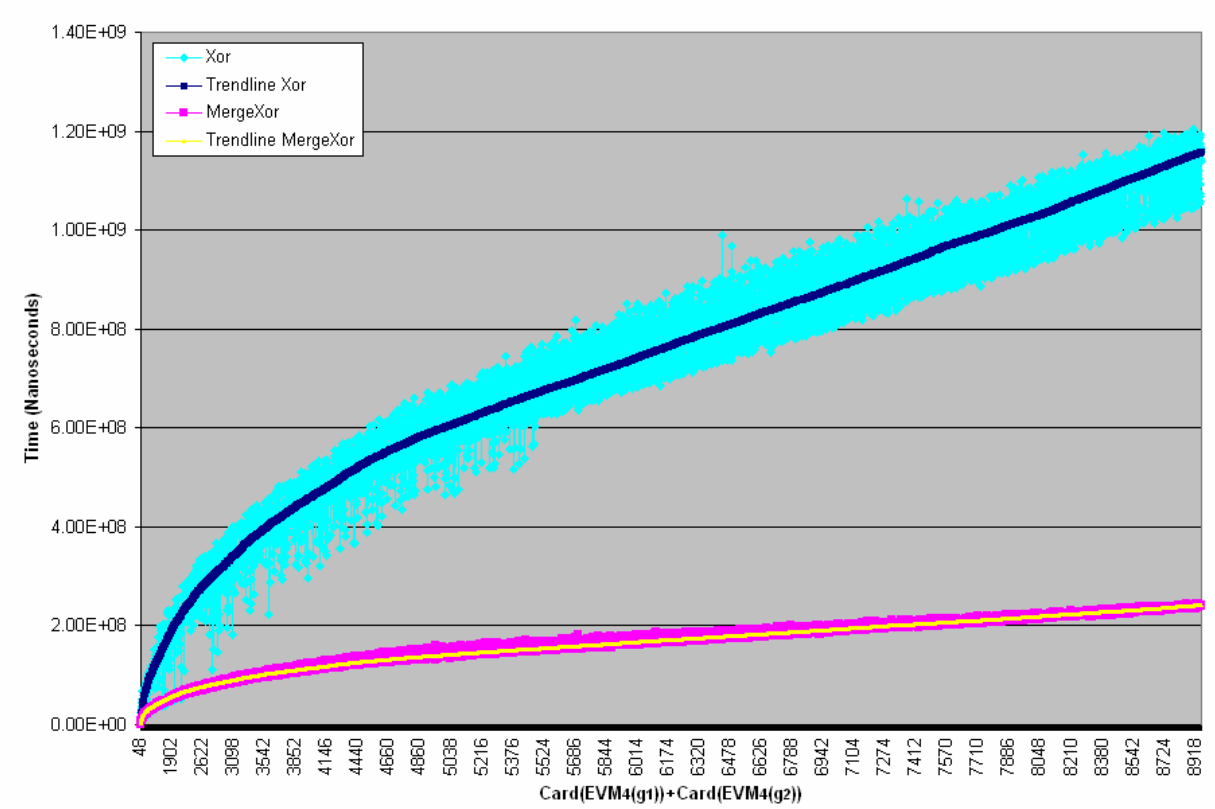


Chart 6.6. Comparing execution times of **Algorithm 6.4** and MergeXor Function under 4D Regularized Xor.

Operation	n	Trendline $y = ax^b$	a	b	R^2
Union	4	$t = 24{,}015.4x^{1.19194}$	24,015.4	1.19194	0.9713
Intersection	4	$t = 4{,}779.91x^{1.32116}$	4,779.91	1.32116	0.9374
Xor	4	$t = 24{,}219.1x^{1.19098}$	24,219.1	1.19098	0.9714
Xor (MergeXor)	4	$t = 50{,}677.2x^{0.93083}$	50,677.2	0.93083	0.9905

Table 6.3. Trendlines approximating the execution times for 4D Regularized Boolean Operations.

Finally we consider case n = 5. The generated 8,000 $Card(EVM_5(g_1)) + Card(EVM_5(g_2))$ presents the following values:

- $Max(Card(EVM_5(g_1)) + Card(EVM_5(g_2))) = 7{,}592$
- $Min(Card(EVM_5(g_1)) + Card(EVM_5(g_2))) = 96$
- $Mean(Card(EVM_5(g_1)) + Card(EVM_5(g_2))) = 4{,}815.6317$
- $Standard\_Deviation(Card(EVM_5(g_1)) + Card(EVM_5(g_2))) = 1668.1757$

The **Charts 6.7** and **6.8** show the timings of **Algorithm 6.4** under 5D Regularized Union, Intersection and Xor. The **Table 6.4** shows the equations of their associated trendlines together with their respective coefficients of determination.

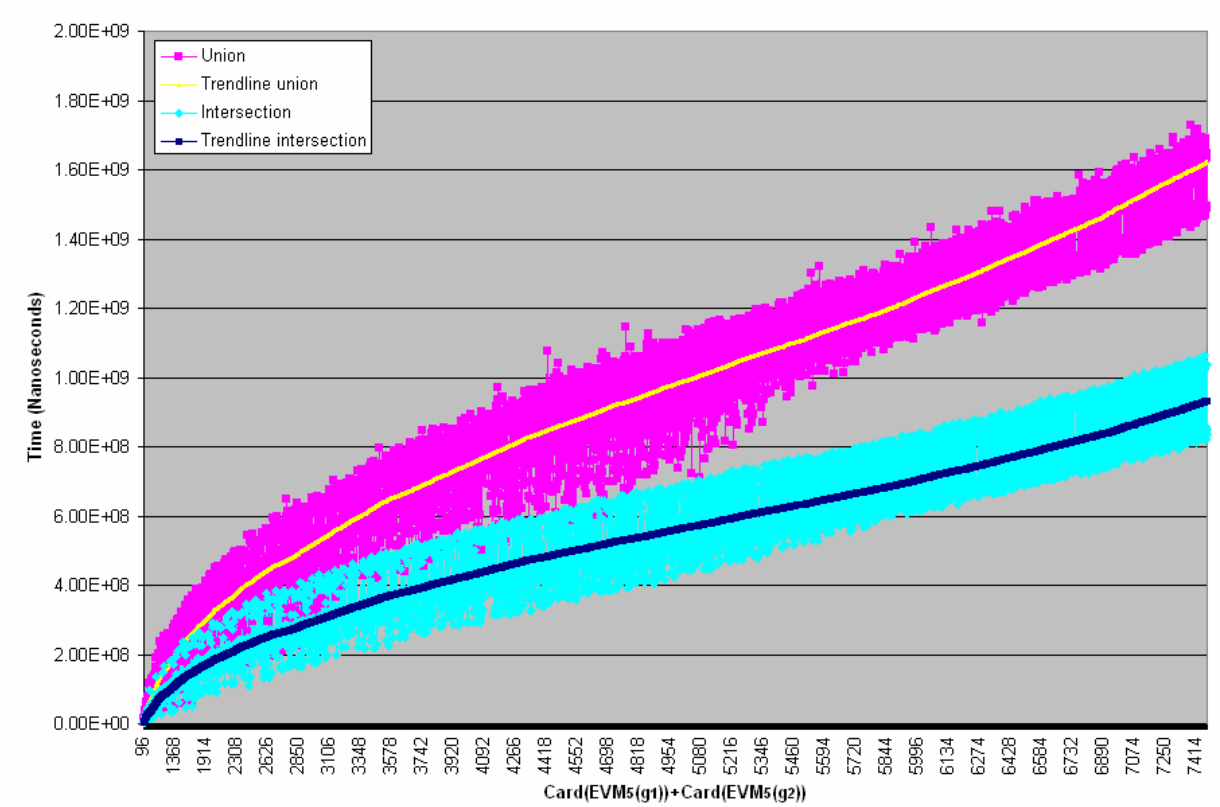


Chart 6.7. Execution times of **Algorithm 6.4** under 5D Regularized Union and Intersection.

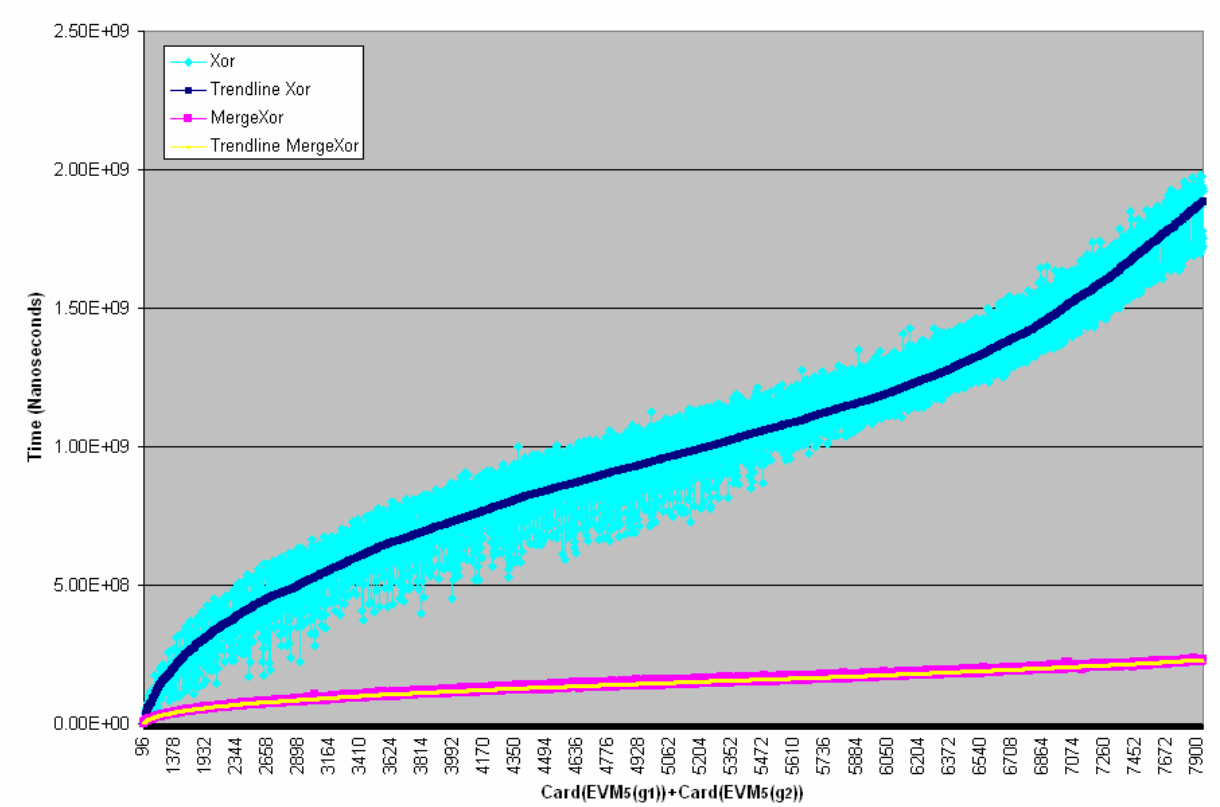


Chart 6.8. Comparing execution times of **Algorithm 6.4** and MergeXor Function under 5D Regularized Xor.

Operation	n	Trendline $y = ax^b$	a	b	R^2
Union	5	$t = 26{,}448.5x^{1.23598}$	26,448.5	1.23598	0.9596
Intersection	5	$t = 13{,}479.3x^{1.24989}$	13,479.3	1.24989	0.9002
Xor	5	$t = 38{,}402.8x^{1.19118}$	38,402.8	1.19118	0.9659
Xor (MergeXor)	5	$t = 32{,}990.5x^{0.98554}$	32,990.5	0.98554	0.9914

Table 6.4. Trendlines approximating the execution times for 5D Regularized Boolean Operations.

According to the results presented in **Charts 6.1** to **6.8** and **Tables 6.1** to **6.4** we have the following observations:

- Performing intersections has a lesser cost respect to unions. This phenomenon was previously identified in [Aguilera98] for the 3D case. Although both operations are performed by the same algorithm, the way the polytopes are processed is distinct. As pointed out by [Aguilera98], **Algorithm 6.4** has three processing stages labeled as stage A, stage B and stage C (see **Figure 6.8**). Only one of the two involved nD-OPP's is present at stages A and C, with trivial recursive calls at stage A, and no recursive calls at stage C. If the involved Boolean operation is an intersection then the result is empty at those stages, thus almost no work is done at stage A, and no work at all is done at stage C. Any way, stage B will deal with both operands, but the recursive calls at this stage will also have stages A, B and C. Unions, on the other hand, produce Boolean results at all three stages [Aguilera98].
- Performing Regularized Xor operation is more efficient by using MergeXor function instead of **Algorithm 6.4**. We have commented previously that MergeXor has a linear complexity execution time because it considers extreme vertices in both input polytopes and discards those vertices present in both polytopes, as established in **Theorem 5.19** (The **Table 6.5** also shows this linearity in experimental way). Moreover, execution time of MergeXor is not affected by the dimensionality of the input polytopes. As seen in **Chart 6.9** we have 2D, 3D, 4D and 5D-OPP's with 0 to approximately 9,000 extreme vertices and although its dimensionality is distinct, its cardinality is the same. In this same Chart can be observed that execution times of **Algorithm 6.4** were always greater than those from function MergeXor.
- The time complexity of **Algorithm 6.4** increases according to the dimensionality of the input nD-OPP's. This situation is easy to deduce because the number of recursivity levels depends of the number of dimensions and it is visualized in **Charts 6.9** to **6.11**.

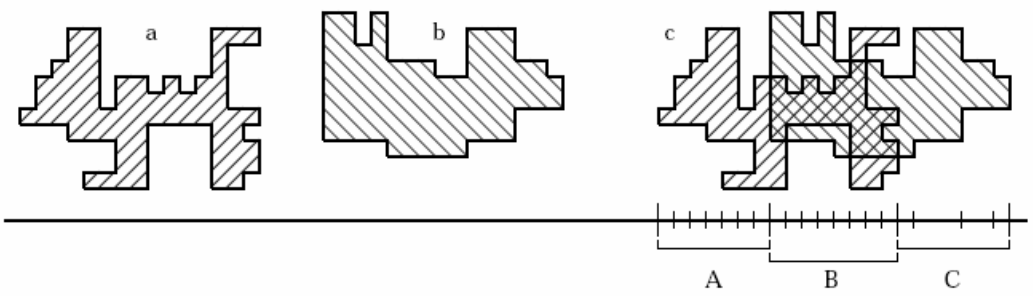


Figure 6.8. Boolean Operations between two 2D-OPP's a and b.
c) The three processing stages (A, B and C) of the Boolean Operations Algorithm (figure taken from [Aguilera98]).

n	Trendline $t = ax^b$	a	b	R^2
2	$t = 20{,}883.7x^{1.08317}$	20,883.7	1.08317	0.9839
3	$t = 46{,}208.3x^{0.96474}$	46,208.3	0.96474	0.9905
4	$t = 50{,}677.2x^{0.93083}$	50,677.2	0.93083	0.9905
5	$t = 32{,}990.5x^{0.98554}$	32,990.5	0.98554	0.9914

Table 6.5. Showing the linearity of execution time in MergeXor function by experimental way.

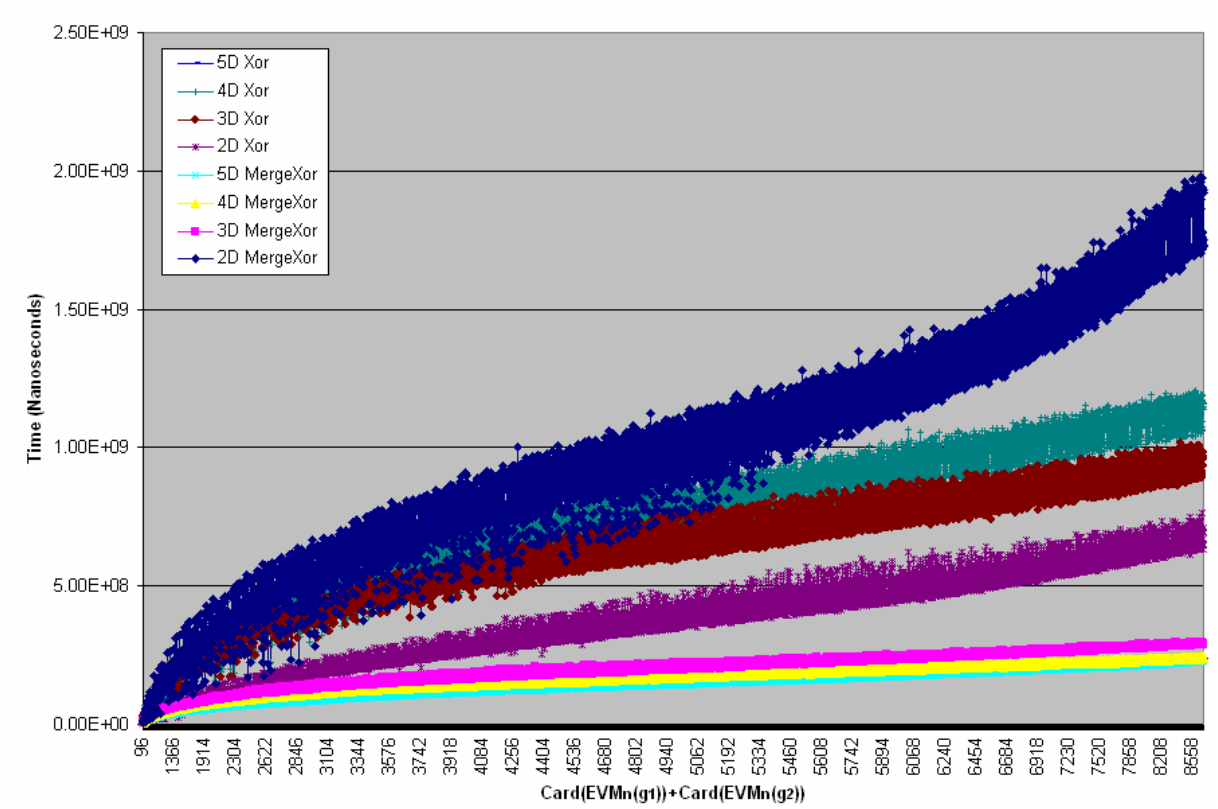


Chart 6.9. Comparing execution times for **Algorithm 6.4** and MergeXor function for nD-OPP's with n = 2, 3, 4, 5 under Regularized Xor.

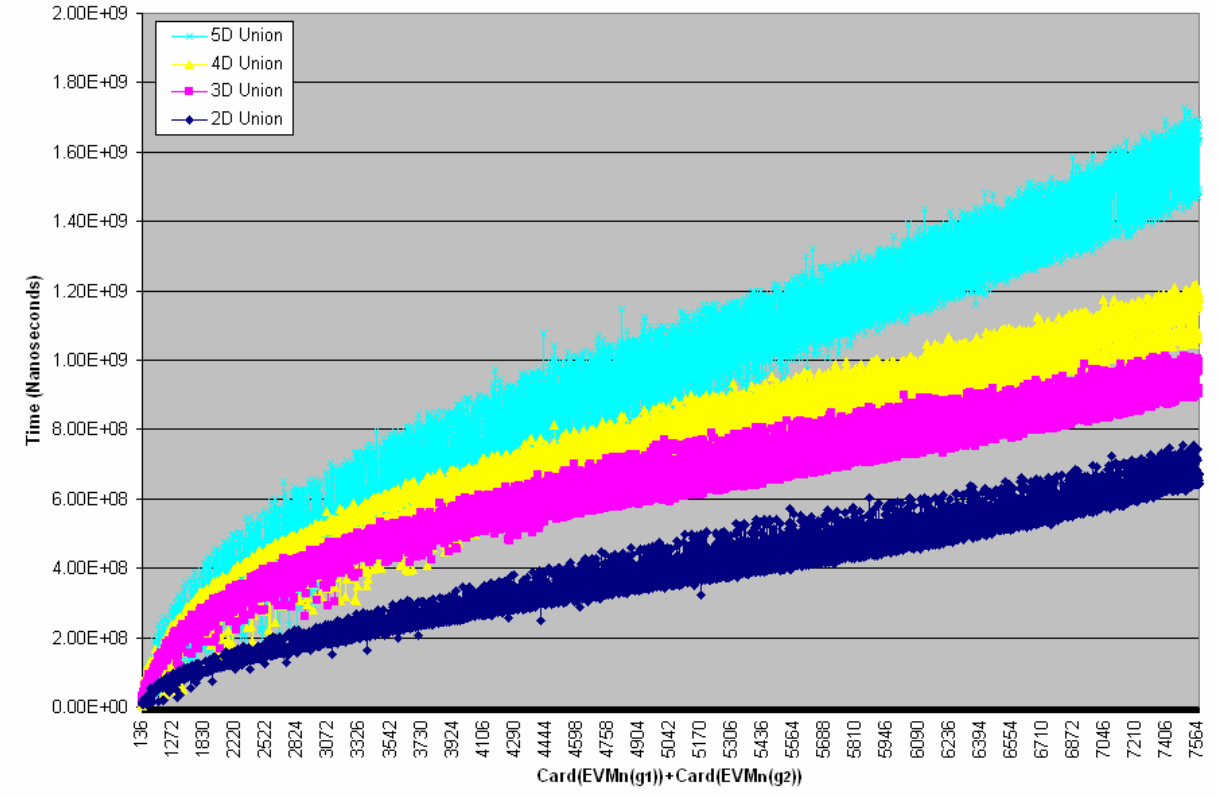


Chart 6.10. Comparing execution times for **Algorithm 6.4** for nD-OPP's with n = 2, 3, 4, 5 under Regularized Union.

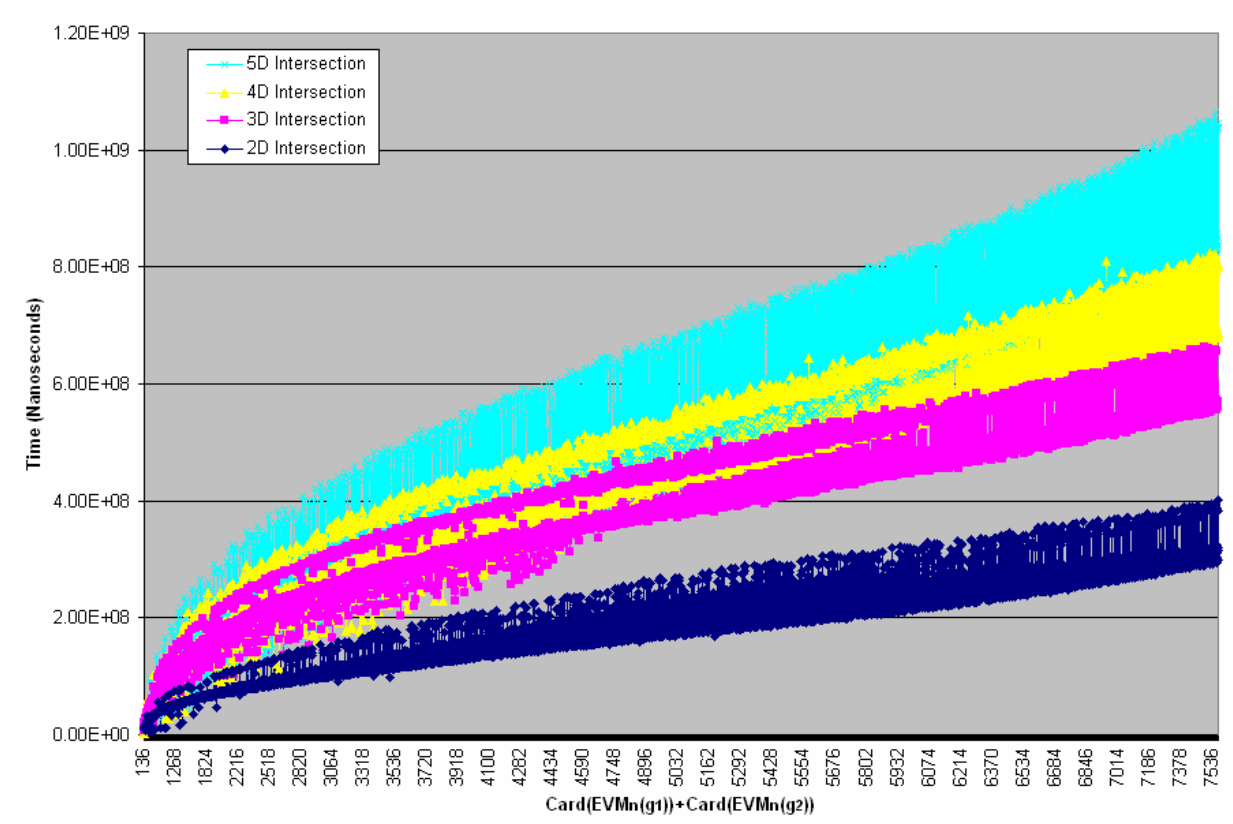


Chart 6.11. Comparing execution times for **Algorithm 6.4** for nD-OPP's with n = 2, 3, 4, 5 under Regularized Intersection.

6.2.1.3. Putting all Together: Providing an Statistical Approximation for Execution Time of Boolean Operations Algorithm under the nD-EVM

According to the results obtained in previous section is natural to expect that execution time of **Algorithm 6.4** depends on two variables: the cardinality x of the nD-EVM's associated to the input polytopes, and the number n of dimensions. Using the measures obtained in previous sections we compute approximation surfaces, i.e., functions from $\mathbb{N}^2$ to $\mathbb{R}$, that provide us an estimation of the execution time to expect given the number of extreme vertices and the number of dimensions. In **Table 6.6** we present approximation surfaces of the form $t = ax^bn^c$ for Intersection and Union operations and their respective coefficients of determination; in the case for Xor operation we present a function of the form $t = ax^bn^b + c$ (The function $t = 4506.37\ x^{1.1819}n^{1.4462}$ was also found for Xor operation, however its coefficient of determination was 0.8357. We decided to propose an alternative form for this specific case in order to provide a more precise estimation).

Operation	Approximation Surface	a	b	c	R^2
Intersection	$t = 4{,}271.11x^{1.1737}n^{1.0862}$	4,271.11	1.1737	1.0862	0.9234
Union	$t = 16{,}698.63x^{1.0821}n^{1.0607}$	16,698.63	1.0821	1.0607	0.9221
Xor	$t = 483.17\ x^{1.4161}n^{1.4161} + 108{,}263{,}080$	483.1716	1.4161	108,263,080	0.9260

Table 6.6. Approximation surfaces for estimating execution times for Boolean operations using **Algorithm 6.4**.

In **Figure 6.9.a** we can visualize in three-dimensional space the approximation surfaces and the way they are related between them. The **Figure 6.9.b** shows another perspective. We have plotted a range of extreme vertices from 0 to 10,000 and the number of dimensions from 0 to 10. It can be observed in these figures that surface approximating execution time for Intersection operation preserves the property identified before: Its execution time is lesser that execution time of Union operation and even Xor operation.

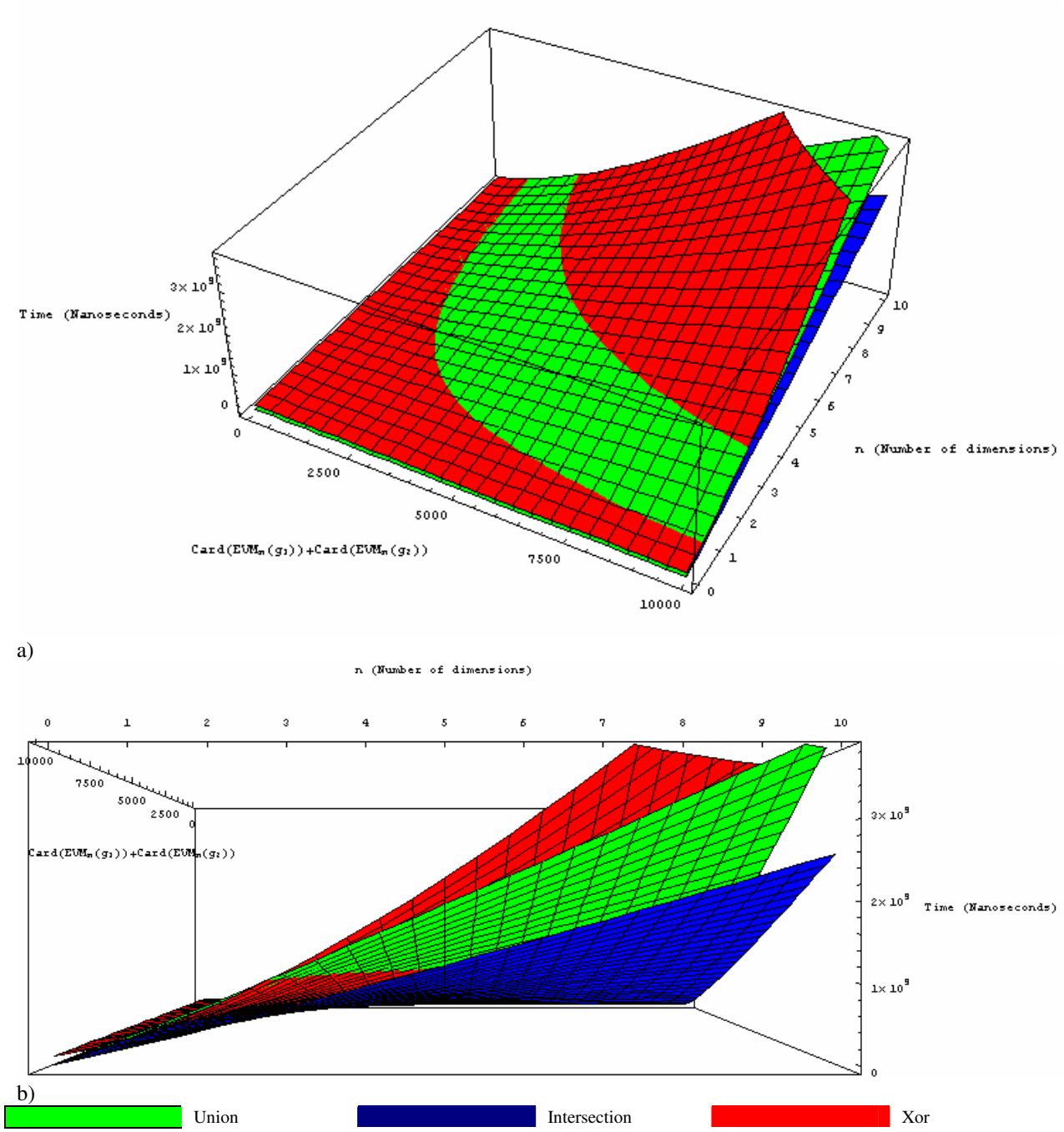


Figure 6.9. Plotting approximation surfaces for execution times of **Algorithm 6.4** under Regularized Boolean Union, Intersection and Xor, $0 \leq \text{Card}(\text{EVM}_n(g_1))+\text{Card}(\text{EVM}_n(g_2)) \leq 10{,}000$; $0 \leq n \leq 10$.

The next logical step to consider is the prediction of execution times for cases $n > 4$ under nD-EVM Boolean Operations Algorithm. This inference can be made through the approximation surfaces we have presented. It is obvious that by fixing the number of dimensions in the equations from **Table 6.6** we obtain a function which depends of the number of input extreme vertices. In **Tables 6.7**, **6.8** and **6.9** we present our predictions for executions times in the cases with $n = 6, 7$ by starting from our approximation surfaces. Our tables show both the trendlines obtained in **Section 6.2.1.2** and the trendlines obtained by fixing n value in the corresponding approximation surface for the referred Boolean operation. Moreover, in such cases we show the coefficients of determination showed in **Section 6.2.1.2** and the coefficients of determination that show how closely the estimated values for the new trendlines correspond to our time measures with $n = 2, 3, 4, 5$.

n	Trendline $t = ax^b$ (Section 6.2.1.2)	R^2	Trendline $t = ax^b$ (by fixing n in approximation surface)	R^2
2	$t = 3{,}920.89\ x^{1.35026}$	0.9709	$t = 34{,}832.4\ x^{1.0821}$	0.8500
3	$t = 45{,}677.9\ x^{1.10379}$	0.9726	$t = 53{,}550.5\ x^{1.0821}$	0.9634
4	$t = 24{,}015.4\ x^{1.19194}$	0.9713	$t = 72{,}658.4\ x^{1.0821}$	0.7679
5	$t = 26{,}448.5\ x^{1.23598}$	0.9596	$t = 92{,}061.6\ x^{1.0821}$	0.9079
6			$t = 111{,}717.5\ x^{1.0821}$	
7			$t = 131{,}563.8\ x^{1.0821}$	

Table 6.7. Fixing the n value in surface approximation for Union operation, under **Algorithm 6.4**, in order to predict trendlines for n > 4.

n	Trendline $t = ax^b$ (Section 6.2.1.2)	R^2	Trendline $t = ax^b$ (by fixing n in approximation surface)	R^2
2	$t = 10{,}006.7\ x^{1.15994}$	0.9030	$t = 9{,}068.18\ x^{1.1737}$	0.9028
3	$t = 24{,}865.6\ x^{1.11831}$	0.9426	$t = 14{,}086.08\ x^{1.1737}$	0.8833
4	$t = 4{,}779.91\ x^{1.32116}$	0.9374	$t = 19{,}253.01\ x^{1.1737}$	0.8573
5	$t = 13{,}479.3\ x^{1.24989}$	0.9002	$t = 24{,}533.7\ x^{1.1737}$	0.8861
6			$t = 29{,}909.9\ x^{1.1737}$	
7			$t = 35{,}362.1\ x^{1.1737}$	

Table 6.8. Fixing the n value in surface approximation for Intersection operation, under **Algorithm 6.4**, in order to predict trendlines for n > 4.

n	Trendline $t = ax^b$ (Section 6.2.1.2)	R^2	Trendline $t = ax^b + c$ (by fixing n in approximation surface)	R^2
2	$t = 4{,}033.48\ x^{1.34649}$	0.9690	$t = 1289.4\ x^{1.4161} + 108263080$	0.72955
3	$t = 49{,}920.9\ x^{1.09301}$	0.9706	$t = 2289.5\ x^{1.4161} + 108263080$	0.9219
4	$t = 24{,}219.1\ x^{1.19098}$	0.9714	$t = 3440.9\ x^{1.4161} + 108263080$	0.7379
5	$t = 38{,}402.8\ x^{1.19118}$	0.9659	$t = 4719.6\ x^{1.4161} + 108263080$	0.9559
6			$t = 6110.1\ x^{1.4161} + 108263080$	
7			$t = 7600.5\ x^{1.4161} + 108263080$	

Table 6.9. Fixing the n value in surface approximation for Xor operation, under **Algorithm 6.4**, in order to predict trendlines for n > 4.

6.3. Computing the Content of an nD-OPP

The 1D content of a segment is its perimeter; the 2D content of a polygon is its area; the 3D content of a polyhedron is its volume, and so on. In this section, we will show a procedure to compute the nD content enclosed by an nD-OPP. An nD hyperprism is generated by the parallel motion of an (n-1)D polytope; it is bounded by the (n-1)D polytope in its initial and final positions and by several (n-1)D hyperprisms [Sommerville58] (a special case of an nD hyperprism is an nD unit hypercube generated according to the procedure by Claude Bragdon [Rucker84], shown in **Figure 6.10**). Consider an nD hyperprism P_n whose base is an (n-1)D polytope P_{n-1} of content C_{n-1}. If h_n is the distance between its bases, i.e. the height of the hyperprism, then its content is given by [Sommerville58]:

$$\text{Content}(P_n) = \text{Content}(P_{n-1}) \cdot h_n = C_{n-1} \cdot h_n \qquad \textbf{(Equation 6.1)}$$

If is the case where P_{n-1} is an (n-1)D hyperprism with height h_{n-1} generated by the parallel motion of an (n-2)D polytope P_{n-2} (as the 4D hypercube in **Figure 6.10.d**) then C_{n-1} is given by the expression $C_{n-1} = C_{n-2} \cdot h_{n-1}$ where C_{n-2} is the content of P_{n-2}. This last expression yields to rewrite **Equation 6.1** as:

$$\text{Content}(P_n) = (\text{Content}(P_{n-2}) \cdot h_{n-1}) \cdot h_n = (C_{n-2} \cdot h_{n-1}) \cdot h_n$$

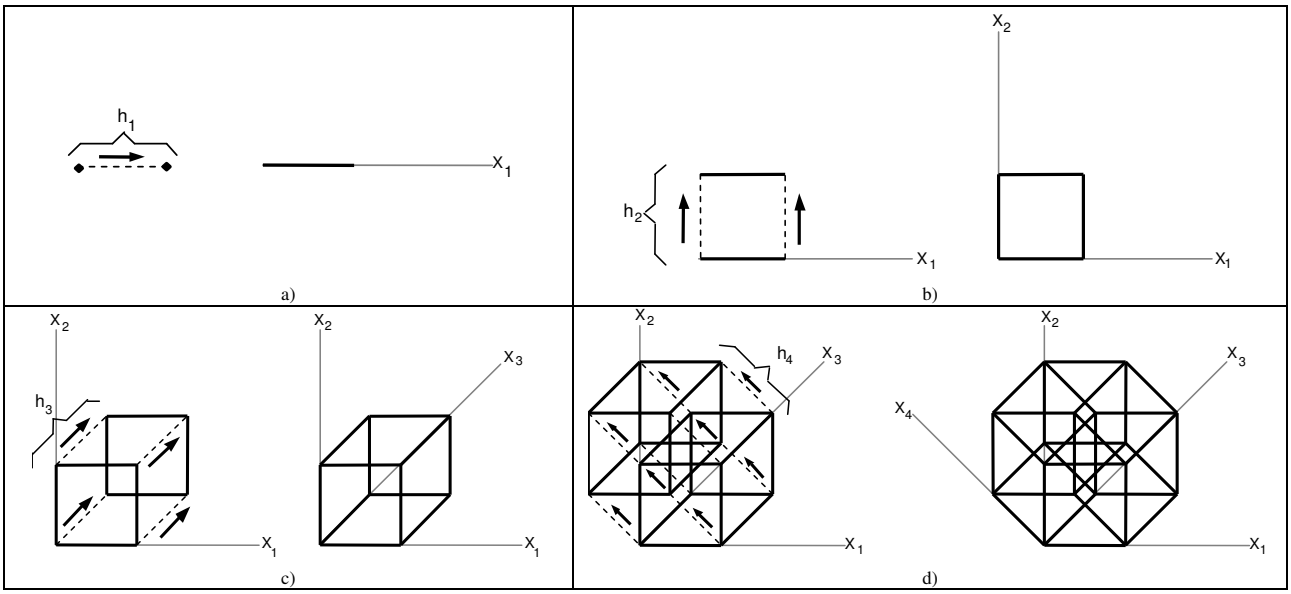


Figure 6.10. The Claude Bragdon process for generating the 4D hypercube. a) A 1D segment is generated by the motion of a point along X_1-axis. b) A 2D square is generated by the motion of a segment along X_2-axis. c) A 3D cube is generated by the motion of a square along X_3-axis. d) A 4D hypercube is generated by the motion of a cube along X_4-axis. The values h_1 to h_4 denote the heights of the hyperprisms generated in each step of Bragdon's sequence.

By considering that each (n-k)D hyperprism P_{n-k} is generated by the parallel motion of an (n-k-1)D hyperprism P_{n-k-1}, where k = 0, 1, 2, …, n-1 (as the cases of the 3D, 2D and 1D cubes from **Figures 6.10.c**, **b** and **a** respectively) then we have that the content of P_n can be computed according to

$$Content(P_n) = \begin{cases} h_1 & n = 1 \\ Content(P_{n-1}) \cdot h_n & n > 1 \end{cases}$$

where h_n is the height of hyperprism P_n when n > 1. In the basic case where n = 1 we have that the content of a segment is given directly by its "height", i.e., the distance between its two boundary points.

Now, we will extend the previous idea in order to compute the content of nD space enclosed by an nD-OPP. In this case we will consider the partition induced by its Slices. A Slice from an nD-OPP can be seen as a set of one or more disjoint nD hyperprisms whose (n-1)D base is the slice' section. As pointed out in [Aguilera98] the volume of a 3D-OPP p can be computed as the sum of the volumes of its 3D slices, where the volume of a $Slice_k^i(p)$ is given by the product between the content of its respective section $S_k^i(p)$ (the 2D base of $Slice_k^i(p)$) and the distance between $\Phi_k^i(p)$ and $\Phi_{k+1}^i(p)$ (the height of the 3D prism $Slice_k^i(p)$). Now let q = $S_k^i(p)$. The area of the 2D-OPP q (see **Figure 6.11** for an example) can be computed as the sum of the areas of its 2D slices, where the area of a $Slice_k^i(q)$ is given by the product between the content of its respective section $S_k^i(q)$ (the 1D base of $Slice_k^i(q)$) and the distance between $\Phi_k^i(q)$ and $\Phi_{k+1}^i(q)$ (the height of the "2D prism" $Slice_k^i(p)$). Finally let r = $S_k^i(q)$. In the basic case the length of the 1D-OPP r is computed as the sum of the lengths of its brinks.

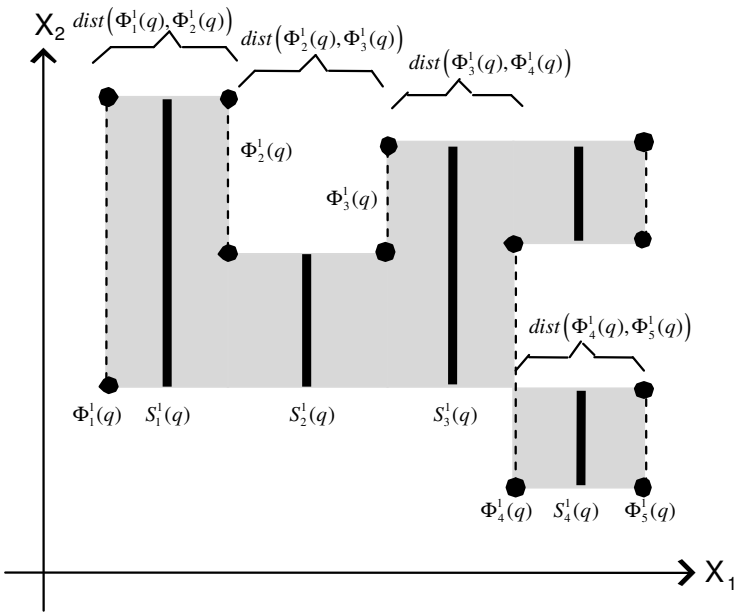


Figure 6.11. A 2D-OPP q whose area is being computed: The total area of q is the sum of the areas of its slices. The area of $Slice_i^1(q)$ is given by the product of the length of its respective section $S_i^1(q)$ and the distance between $\Phi_i^1(q), \Phi_{i+1}^1$.

Let p be an nD-OPP. The nD space enclosed by p, denoted by Content$_{(n)}$(p), can be computed as the sum of the contents of its nD slices:

$$Content_{(n)}(p) = \begin{cases} Length(p) & n = 1 \\ \sum_{k=1}^{np_i - 1} Content_{(n-1)}\left(S_k^i(p)\right) \cdot dist\left(\Phi_k^i(p), \Phi_{k+1}^i(p)\right) & n > 1 \end{cases} \quad \textbf{(Equation 6.2)}$$

Where np_i is the number of couplets of p perpendicular to X_i-axis; $S_k^i(p)$ is the k-th section of the nD-OPP p which is perpendicular to X_i-axis and it is between couplets $\Phi_k^i(p), \Phi_{k+1}^i(p)$.

Algorithm 6.5 implements **Equation 6.2** in order to compute the content of nD space enclosed by a nD-OPP p expressed through the EVM-nD.

```
Input: An nD-EVM p.
       The number n of dimensions.
Output: The content of nD space enclosed by p.
Procedure Content(EVM p, int n)
       real cont = 0.0        // Variable cont will store the content of nD space enclosed by p.
       EVM hvl1, hvl2         // Couplets between a slice of p.
       EVM s                  // Current section of p.
       if(n = 1) then
              return Length(p)
       else
              n = n - 1
              hvl1 = InitEVM( )
              hvl2 = InitEVM( )
              s = InitEVM( )
              hvl1 = ReadHvl(p)
              while(Not(EndEVM(p)))
                     hvl2 = ReadHvl(p)
                     s = GetSection(s, hvl1)
                     cont = cont + Content(s, n) * dist(hvl1, hvl2)      // Recursive Call.
                     hvl1 = hvl2
              end-of-while
              return cont
       end-of-else
end-of-procedure
```

Algorithm 6.5. Computing the content of nD space enclosed by p.

6.3.1. Performance of the Algorithm

We will proceed with a statistical analysis in order to determine execution time of **Algorithm 6.5**. This analysis share some aspects respect to the study described in **Section 6.2.1**. In this case we will describe only the key points related to the analysis to be applied over **Algorithm 6.5**:

- Our testing consider n = 2, 3, 4, 5.
- For each n we have generated 10,000 random nD-OPP's according to the following procedures:
  - Given a hypervoxelization representing nD-OPP's g we obtain their respective nD-EVM, $EVM_n(g)$.
  - Let C be the content of nD space enclosed by the polytope represented through $EVM_n(g)$. Such content is computed through **Algorithm 6.5**.
  - Let C' be the content of nD space enclosed by the polytope g represented through a hypervoxelization. Such computing is straightforward.
  - As a mechanism for controlling possible errors in our implementations we verified that all the 10,000 generated nD-OPP's satisfied C = C'.

The **Table 6.10** shows some information related to our generated data. In **Chart 6.12** it can be visualized the behavior of **Algorithm 6.5** with our set of nD-OPP's. In the same chart can be also visualized the associated trendline for each value of n.

n	Max	Min	Mean	Standard Deviation
2	5,418	0	2,788.999	1,549.5579
3	5,352	0	3,148.462	1,485.0206
4	5,464	0	3,210.922	1,488.6948
5	5,332	0	2,743.690	1,457.3235

Table 6.10. Some statistical characteristics of the set of 10,000 random nD-OPP's for testing of **Algorithm 6.5**.

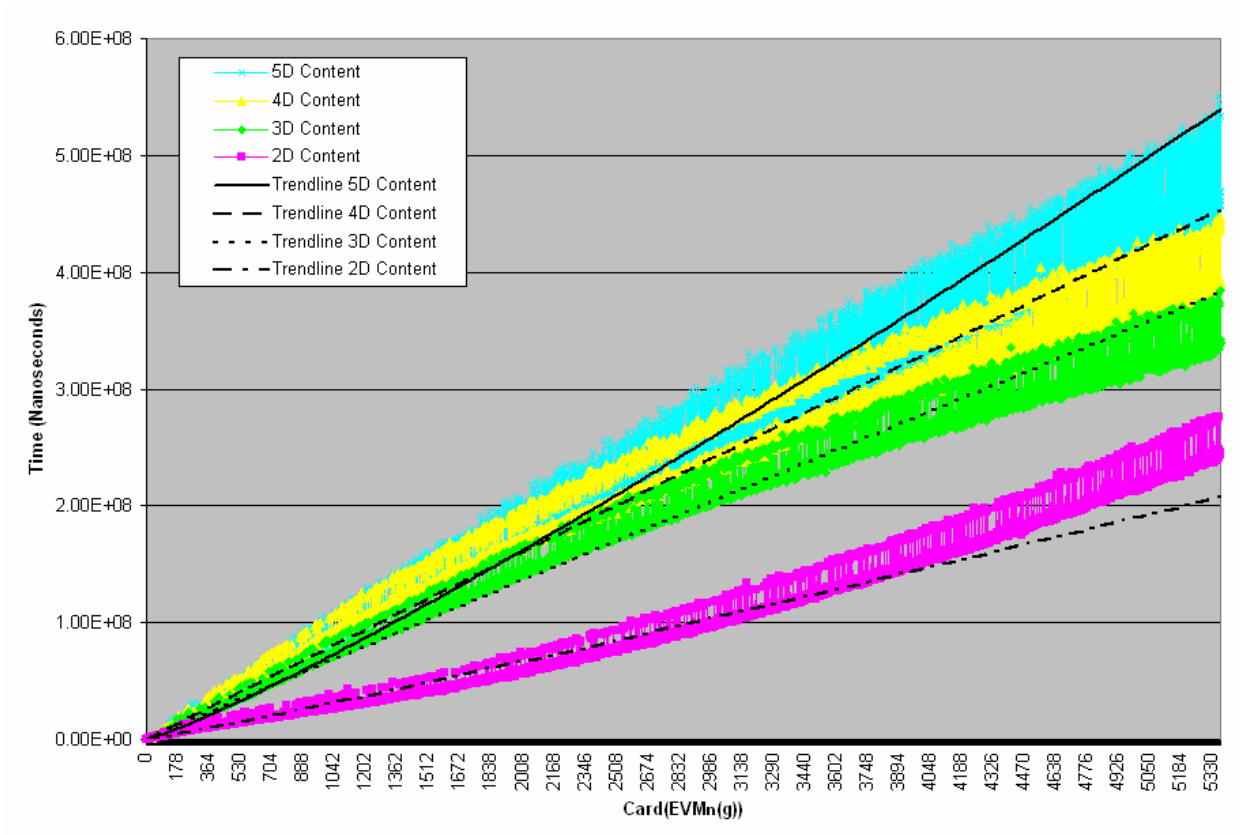


Chart 6.12. Comparing execution times for **Algorithm 6.5** for nD-OPP's with n = 2, 3, 4, 5.

n	Trendline $t = ax^b$	a	b	R^2
2	$t = 9{,}715.6x^{1.0823}$	9,715.6	1.0823	0.9844
3	$t = 44{,}031x^{0.9848}$	44,031	0.9848	0.9878
4	$t = 49{,}267x^{0.9910}$	49,267	0.9910	0.9665
5	$t = 13{,}955x^{1.1469}$	13,955	1.1469	0.9479

Table 6.11. Equations associated to the trendlines that describe execution time of **Algorithm 6.5** in the cases with n = 2, 3, 4, 5.

An interesting aspect to be inferred from **Table 6.11** yields to make the observation that all the exponents in the obtained equations are almost linear. This property is preserved when we determine an approximation surface for execution time of **Algorithm 6.5**. The associated equation, as previously commented in **Section 6.2**, is a function from $\mathbb{N}^2$ to $\mathbb{R}$ which has as arguments the number x of extreme vertices in the input polytope g, i.e., $x = \text{Card}(EVM_n(g))$ and the number of dimensions n. According to our analysis we have that the approximation surface is given by

$$t = 4{,}763.939\ x^{1.1894}n^{0.8390}$$

In this case we have identified a coefficient of determination $R^2 = 0.9803$. The **Figure 6.12** shows the plotting of the above function and shows graphically an estimation of the execution time of **Algorithm 6.5** when the number of input extreme vertices is from 0 to 10,000 and when the number of dimensions is between 0 and 10.

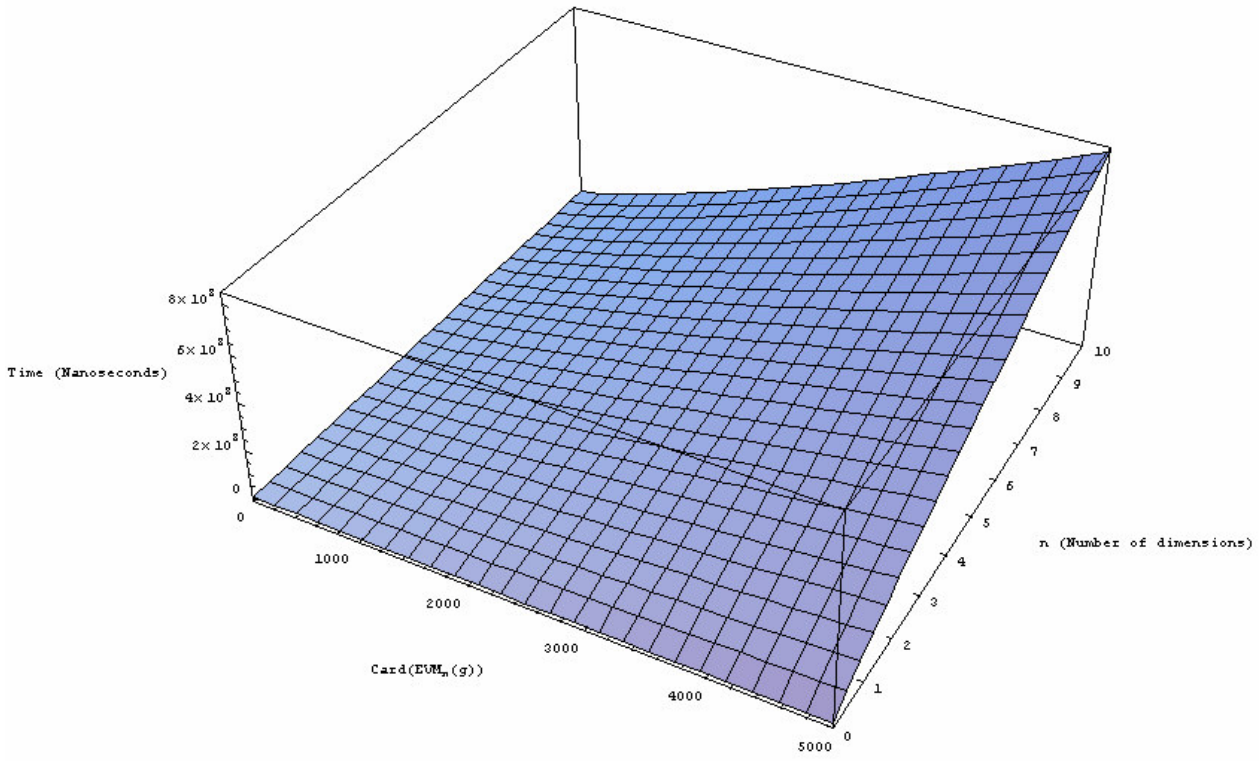


Figure 6.12. Plotting the approximation surface for execution time of **Algorithm 6.5**, $0 \leq \text{Card}(EVM_n(g)) \leq 10{,}000$; $0 \leq n \leq 10$.

The prediction of execution time for **Algorithm 6.5** based in our approximation surface can be performed by fixing the value of n in its associated equation. In this case, we have obtained a good approximation by our trendlines presented in **Table 6.11** and now by considering the new trendlines obtained from the equation of our approximation surface. See **Table 6.12** where we present our estimations for execution time of **Algorithm 6.5** in the cases n = 6, 7.

n	Trendline $t = ax^b$	R^2	Trendline $t = ax^b$ (by fixing n in approximation surface)	R^2
2	$t = 9{,}715.6x^{1.0823}$	0.9844	$t = 8521.78\ x^{1.1894}$	0.9702
3	$t = 44{,}031x^{0.9848}$	0.9878	$t = 11{,}974.9\ x^{1.1894}$	0.9745
4	$t = 49{,}267x^{0.9910}$	0.9665	$t = 15{,}243.8\ x^{1.1894}$	0.9747
5	$t = 13{,}955x^{1.1469}$	0.9479	$t = 18{,}382.4\ x^{1.1894}$	0.9785
6			$t = 21{,}420.8\ x^{1.1894}$	
7			$t = 24{,}378.3\ x^{1.1894}$	

Table 6.12. Fixing the n value in surface approximation for **Algorithm 6.5** in order to predict trendlines for n > 4.

6.4. Computing the Content of the Boundary of an nD-OPP

Consider an nD hyperprism P_n whose base is an (n-1)D polytope P_{n-1} of (n-1)D content C_{n-1}. Let h_n be the distance between its bases, i.e. the height of the hyperprism. Because our nD hyperprism P_n is generated by the parallel motion of P_{n-1} we have that the intersection between P_n and an (n-1)D hyperplane parallel to P_{n-1} always generates an (n-1)D polytope P'_{n-1} with the same characteristics that P_{n-1}. Computing the (n-2)D content BN_{n-2} of the boundary of P'_{n-1} implies to compute each one the (n-2)D contents of its boundary elements. By multiplying each term in BN_{n-2} by the height h_n of P_n we get the (n-1)D content of each one of the (n-1)D hyperprisms perpendicular to the bases of P_n. Trough this reasoning we get the following expression:

$$\text{BoundaryContent}(P_n) = 2\cdot\text{Content}(P_{n-1}) + \text{BoundaryContent}(P_{n-1})\cdot h_n = 2\cdot C_{n-1} + BN_{n-2}\cdot h_n$$

In analogous way, respect to previous section, we have that this last expression descends recursively in the number of dimensions where the basic case is reached when n = 2 where the perimeter (1D content of the boundary) of a rectangle P_2 is directly computed as the sum of the lengths of its four edges:

$$BoundaryContent(P_n) = \begin{cases} Perimeter(P_2) & n = 2 \\ 2\cdot Content(P_{n-1}) + BoundaryContent(P_{n-1})\cdot h_n & n > 2 \end{cases}$$

Now, we will extend the previous idea in order to compute the content of (n-1)D space enclosed by the boundary of an nD-OPP. [Aguilera98] points out that the surface of a 3D-OPP p (see **Figure 6.13** for an example) can be computed as the sum of the areas of its 2D couplets, where the area of a $\Phi_k^i(p)$ is given by $Content_2\left(\Phi_k^i(p)\right)$ (**Equation 6.2**). To this sum must be added the sum of the areas of the faces between $\Phi_k^i(p)$ and $\Phi_{k+1}^i(p)$. These areas are found by the product between the perimeter of the section $S_k^i(p)$ and the distance between $\Phi_k^i(p)$ and $\Phi_{k+1}^i(p)$ (the height of the 3D prism $Slice_k^i(p)$). Now let q = $S_k^i(p)$, we have reached the basic case. The perimeter of the 2D-OPP q can be computed as [Aguilera98]:

$$Perimeter(q) = x_1 Sum(q) + x_2 Sum(q)$$

where $x_i Sum(q)$ is the sum of the lengths of all brinks parallel to X_i-axis.

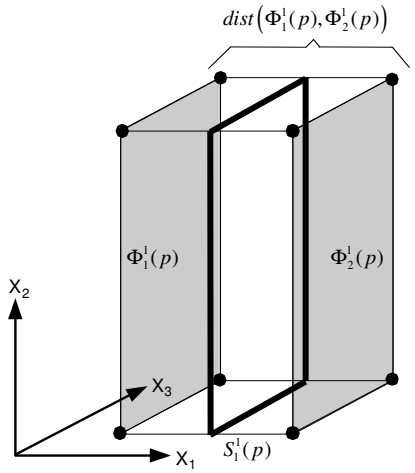


Figure 6.13. Computing the content of the boundary in a 3D prism: part of the total area is found by computing Content$_2$($\Phi_1^1(p)$) and Content$_2$($\Phi_2^1(p)$) through **Algorithm 6.5**. The area of the remaining four faces is determined through the product of the perimeter of the section $S_1^1(p)$ and the distance between $\Phi_1^1(p)$ and $\Phi_2^1(p)$.

Let p be an nD-OPP. The (n-1)D space enclosed by p, denoted by BoundaryContent$_{(n-1)}$(p), can be computed as follows (**Equation 6.3**):

$$BoundaryContent_{(n-1)}(p) = \begin{cases} x_1 Sum(p) + x_2 Sum(p) & n = 2 \\ \sum_{k=1}^{np_i} Content_{(n-1)}\left(\Phi_k^i(p)\right) + \sum_{k=1}^{np_i-1} BoundaryContent_{(n-1)}\left(S_k^i(p)\right) \cdot dist\left(\Phi_k^i(p), \Phi_{k+1}^i(p)\right) & n > 2 \end{cases}$$

Algorithm 6.6 implements **Equation 6.3** in order to compute the content of (n-1)D space enclosed by the boundary of p expressed through the EVM-nD.

```
Input: An nD-EVM p.
       The number n of dimensions.
Output: The content of (n-1)D space enclosed by the boundary of p.
Procedure BoundaryContent(EVM p, int n)
       real cont = 0.0        // cont stores the content of (n-1)D space enclosed by the boundary of p.
       EVM hvl1, hvl2         // Couplets between a slice of p.
       EVM s                  // Current section of p.
       hvl1 = InitEVM( )
       hvl2 = InitEVM( )
       s = InitEVM( )
       If(n = 2) then
              return cont = x1Sum(p) + x2Sum(p)
       else
              n = n - 1
              hvl1 = ReadHvl(p)
              while(Not(EndEVM(p)))
                     hvl2 = ReadHvl(p)
                     s = GetSection(s, hvl1)
                     // Call to algorithm Content and recursive call.
                     cont = cont + Content(hvl1, n) + BoundaryContent(s, n) * dist(hvl1, hvl2)
                     hvl1 = hvl2
              end-of-while
              cont = cont + Content(hvl1, n) // hvl1 contains the last couplet of p.
              return cont
       end-of-else
end-of-procedure
```

Algorithm 6.6. Computing the content of (n-1)D space enclosed by the boundary of p.

6.4.1. Performance of the Algorithm

The key points of the statistical analysis for execution time of **Algorithm 6.6** are very similar to the analysis described in **Section 6.3.1**:

- Our testing consider n = 2, 3, 4, 5.
- For each n we have generated 10,000 random nD-OPP's according to the following procedures:
  - Given a hypervoxelization representing nD-OPP's g we obtain their respective nD-EVM, $EVM_n(g)$.
  - Let BC be the content of the boundary of the polytope represented through $EVM_n(g)$. Such content is computed through **Algorithm 6.6**.
  - Let BC' be the content of the boundary enclosed by the polytope g represented through a hypervoxelization. Such computing is performed in a straightforward way.
  - As a mechanism for controlling possible errors in our implementations we verified that all the 10,000 generated nD-OPP's satisfied BC = BC'.

The **Table 6.13** shows some information related to our generated data. In **Chart 6.13** it can be visualized the behavior of **Algorithm 6.6** with our set of nD-OPP's. In the same chart can be also visualized the associated trendline for each value of n whose associated equations are shown in **Table 6.14**.

n	Max	Min	Mean	Standard Deviation
2	5,442	0	2,789.028	1,549.5244
3	5,382	0	3,148.203	1484.8637
4	5,482	0	3,212.393	1,489.3084
5	5,278	0	2,743.906	1,457.0870

Table 6.13. Some statistical characteristics of the set of 10,000 random nD-OPP's for testing of **Algorithm 6.6**.

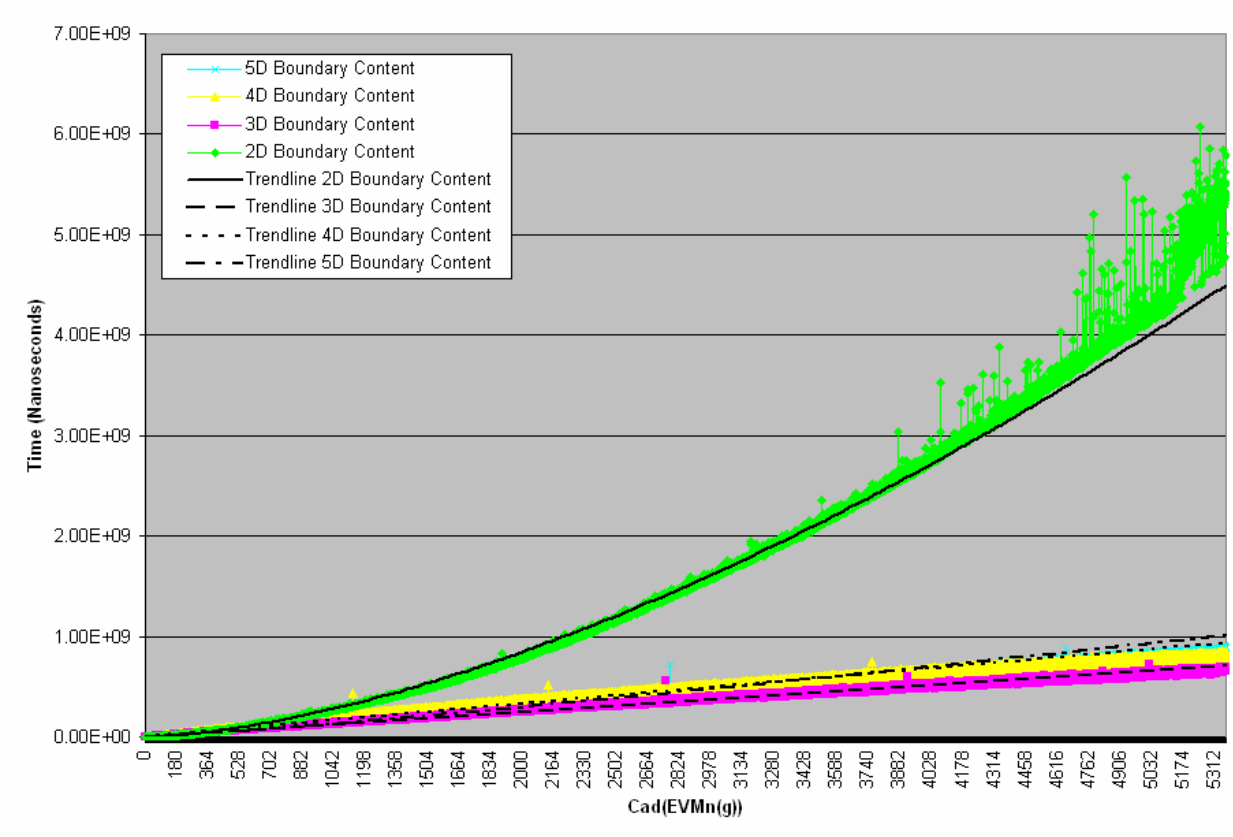


Chart 6.13. Comparing execution times for **Algorithm 6.6** for nD-OPP's with n = 2, 3, 4, 5.

n	Trendline $t = ax^b$	a	b	R^2
2	$t = 2{,}034.8x^{1.5862}$	2,034.8	1.5862	0.9976
3	$t = 69{,}566x^{1.0018}$	69,566	1.0018	0.9889
4	$t = 105{,}588x^{0.9867}$	105,588	0.9867	0.9571
5	$t = 21{,}943x^{1.1658}$	21,943	1.1658	0.9321

Table 6.14. Equations associated to the trendlines that describe execution time of **Algorithm 6.6** in the cases with n = 2, 3, 4, 5.

As seen in **Chart 6.13,** and observing equations from trendlines in **Table 6.14**, execution time of **Algorithm 6.6** in the 2D case is above execution times of cases with n = 3, 4, 5. We will expose the reasons behind this behavior. Consider nD-OPP's from **Figures 6.14.a**, **d** and **g**. All of them contain 16 extreme vertices. **Figure 6.14.a** is a 2D-OPP an according to **Algorithm 6.6** it is in the basic case. Due to the ordering we are considering for its extreme vertices the lengths of the brinks parallel to X_2-axis are directly computed (**Figure 6.14.b**). In the other hand, in order to compute the lengths of the brinks parallel to X_1-axis a sorting must be applied (see **Figure 6.14.c**). In the particular case of this 2D-OPP the sorting considers 16 vertices. In the case related to **Figure 6.14.d**, the 3D-OPP p, whose 2D couplets perpendicular to X_1-axis are shown in **Figure 6.14.e**, has two sections, perpendicular to X_1-axis, with 6 extreme vertices in each one (**Figure 6.14.f**). Those sections reach the basic case of **Algorithm 6.6** and hence two sets of 6 extreme vertices must be sorted. Finally, the 4D hypercube from **Figure 6.14.g** has one 3D section (**Figure 6.14.h**), and in the 2D case, one section with 4 extreme vertices which are also sorted (**Figure 6.14.i**). It is clear that all 2D-OPP's reach the basic case of **Algorithm 6.6** and therefore all its extreme vertices must be sorted for computing the lengths of the brinks parallel to X_1-axis. According to **Chart 6.13**, this situation is relaxed according the dimensionality increases and execution times in cases for n = 3, 4, 5 increase, but not at the same order than 2D case.

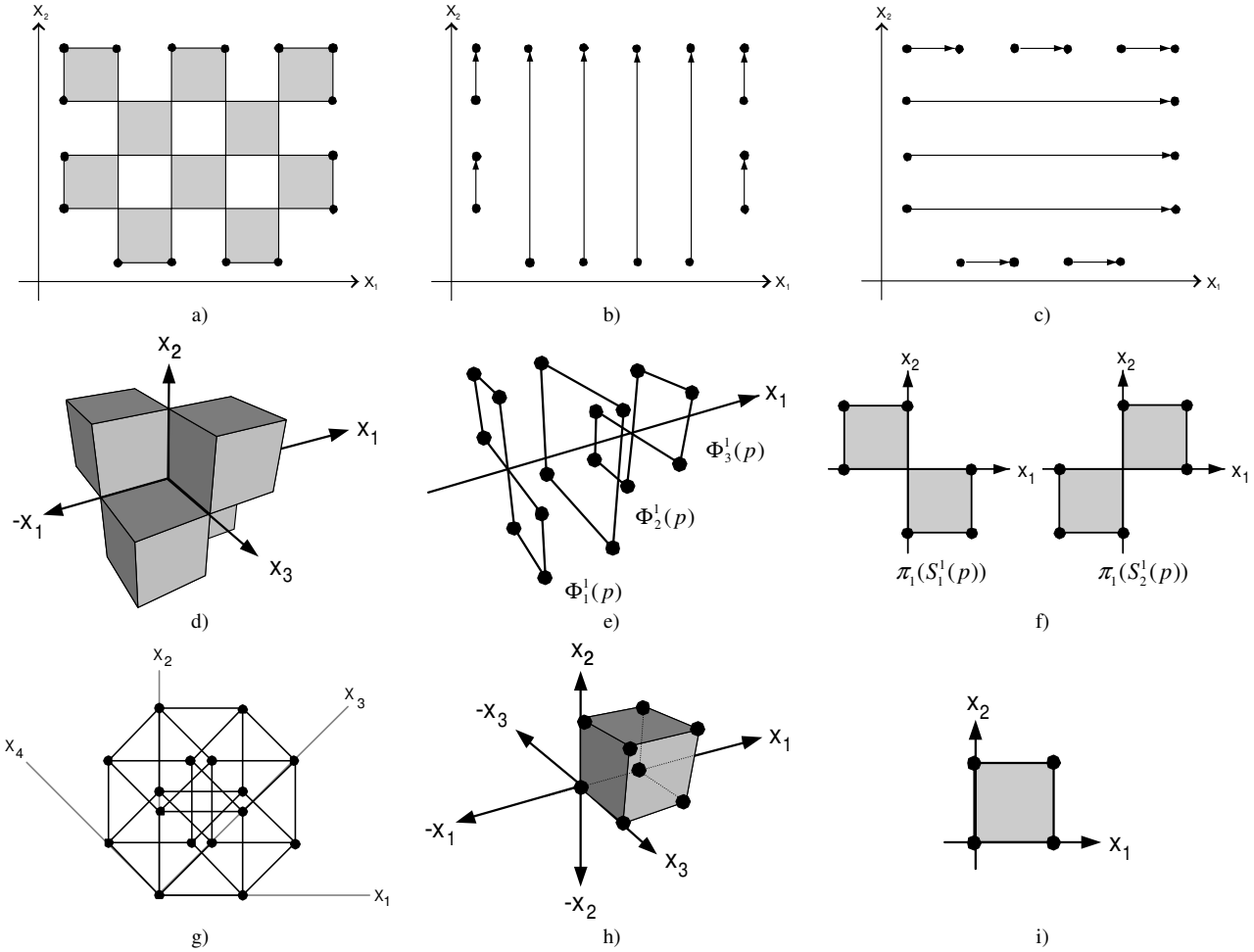


Figure 6.14. Three nD-OPP's with 16 extreme vertices which shown the behavior behind execution time of **Algorithm 6.6**.
a) A 2D-OPP, d) a 3D-OPP and g) a 4D hypercube (see text for details).

Now we determine an approximation surface for execution time of **Algorithm 6.6**. The associated equation is a function from $\mathbb{N}^2$ to $\mathbb{R}$ which has as arguments the number x of extreme vertices in the input polytope g, i.e., x = Card($EVM_n(g)$) and the number of dimensions n. According to our analysis the best approximation surface we have found is given by

$$t = \frac{1454430n + 12921.02x - 7196150}{0.0212336n^3 - 0.247936n^2 + 0.92776n - 1}$$

Whose coefficient of determination is $R^2 = 0.9963$. The **Figure 6.15** shows the plotting of the above function and shows graphically an estimation of the execution time of **Algorithm 6.6** when the number of input extreme vertices is from 0 to 10,000 and when the number of dimensions is between 2 and 10.

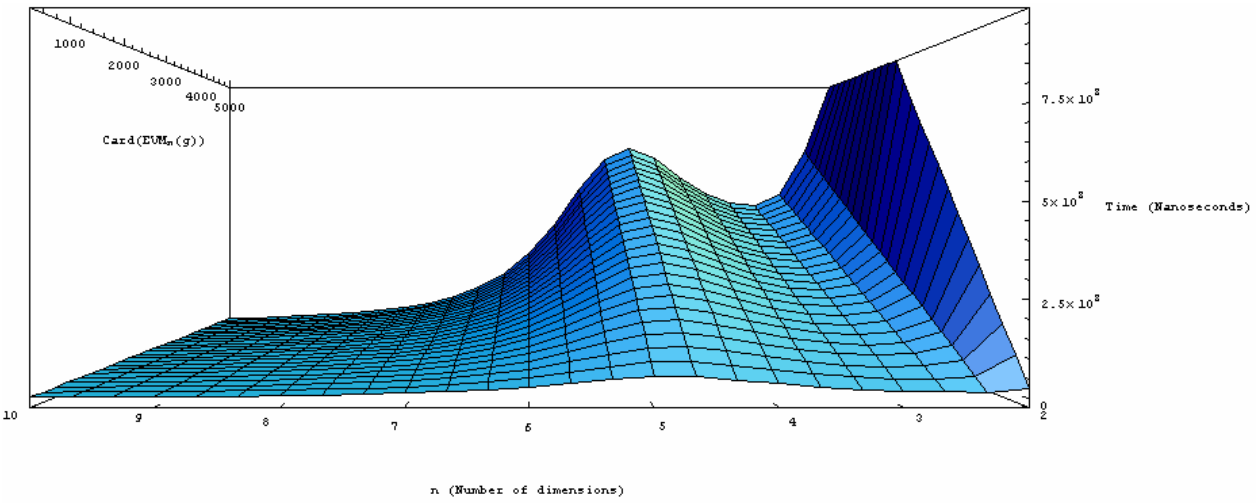


Figure 6.15. Plot of the approximation surface for execution time of **Algorithm 6.6**, $0 \leq Card(EVM_n(g)) \leq 10{,}000$; $2 \leq n \leq 10$.

As shown in previous algorithms, the prediction of execution time for **Algorithm 6.6** based in our approximation surface can be performed by fixing the value of n in its associated equation. In this case, we have obtained the new trendlines presented in **Table 6.15** (the second and third columns show trendlines and coefficients of determination obtained from the data shown in **Chart 6.13**). A special mention is given to the new trendline obtained for the case n = 2 where we identify a coefficient $R^2 = 0.2875$. This situation is present because we consider all data for n = 2, 3, 4, 5 when we determined our approximation surface and, moreover, we mentioned before that the case n = 2 is special, respect to cases with n > 2, because the behavior of the **Algorithm 6.6** in the basic case. See **Table 6.15** where we present our estimations for execution time of **Algorithm 6.6**. in the cases n = 6, 7.

n	Trendline $t = ax^b$	R^2	Trendline $t = ax + b$ (by fixing n in approximation surface)	R^2
2	$t = 2{,}034.8\ x^{1.5862}$	0.9976	$384{,}039.59\ x - 1.27427 \times 10^8$	0.2875
3	$t = 69{,}566\ x^{1.0018}$	0.9889	$103{,}232.89\ x - 2.26333 \times 10^7$	0.8883
4	$t = 105{,}588\ x^{0.9867}$	0.9571	$125{,}427.79\ x - 1.33809 \times 10^7$	0.8681
5	$t = 21{,}943\ x^{1.1658}$	0.9321	$136{,}582.83\ x + 803{,}287.66$	0.8774
6			$56{,}840.3\ x + 6{,}732{,}440$	
7			$20{,}555.9\ x + 4{,}748{,}570$	

Table 6.15. Fixing the n value in surface approximation for **Algorithm 6.6** in order to predict trendlines for n > 4.

6.5. Computing Forward and Backward Differences of an nD-OPP

According to **Theorem 5.18**, in an nD-OPP p, forward differences $FD_k^i(p)$ are the (n-1)D cells on $\Phi_k^i(p)$ whose normal vectors point to the positive side of the coordinate axis X_i which is perpendicular to $\Phi_k^i(p)$, while backward differences $BD_k^i(p)$ are the (n-1)D cells on $\Phi_k^i(p)$ whose normal vectors point to the negative side of the coordinate axis X_i which is perpendicular to $\Phi_k^i(p)$. Through **Definition 5.25** we have that a forward difference $FD_k^i(p)$ and a backward difference $BD_k^i(p)$ are computed according to $\left(\pi_i\left(S_{k-1}^i(p)\right) -^* \pi_i\left(S_k^i(p)\right)\right)$ and $\left(\pi_i\left(S_k^i(p)\right) -^* \pi_i\left(S_{k-1}^i(p)\right)\right)$ respectively. Hence, an algorithm for computing forward and backward differences consists in obtaining projections of sections of the input polytope and processing them through **Definition 5.25** and **Algorithm 6.4** by computing Regularized difference between two consecutive sections, in order to obtain their corresponding forward and backward differences. **Algorithm 6.7** implements the above ideas in order to compute the forward and backward differences in an nD-OPP p represented through the nD-EVM. The output of the proposed algorithm will consist of two sets: the first set FD contains the (n-1)D-EVM's corresponding to forward differences in p, that is FD = $\left\{EVM_{n-1}\left(FD_1^i(p)\right),...,EVM_{n-1}\left(FD_{np_i}^i(p)\right)\right\}$; while the second set BD contains the (n-1)D-EVM's corresponding to backward differences in p, i.e. BD = $\left\{EVM_{n-1}\left(BD_1^i(p)\right),...,EVM_{n-1}\left(BD_{np_i}^i(p)\right)\right\}$.

```
Input: An nD-EVM p.
       The number n of dimensions.
Output: A set FD containing the (n-1)D-EVM's of forward differences in p.
       A set BD containing the (n-1)D-EVM's of backward differences in p.
Procedure GetForwardBackwardDifferences(EVM p, int n)
       FD = ∅          // FD will store (n-1)D-EVM's corresponding to forward differences.
       BD = ∅          // BD will store (n-1)D-EVM's corresponding to backward differences.
       EVM hvl         // Current couplet.
       EVM Si, Sj      // Previous and next sections about hvl.
       EVM FDcurr      // Current forward difference.
       EVM BDcurr      // Current backward difference.
       hvl = InitEVM( )
       Si = InitEVM( )
       Sj = InitEVM( )
       while(Not(EndEVM(p)))
              hvl = ReadHvl(p)              // Read next couplet.
              Sj = GetSection(Si, hvl)
              FDcurr = BooleanOperation(Si, Sj, DifferenceOperator, n-1)  //Call to Algorithm 6.4.
              BDcurr = BooleanOperation(Sj, Si, DifferenceOperator, n-1)  //Call to Algorithm 6.4.
              FD = FD ∪ FDcurr      // The new computed forward difference is added to set FD.
              BD = BD ∪ BDcurr      // The new computed backward difference is added to set BD.
              Si = Sj
       end-of-while
       return FD, BD
end-of-procedure
```

Algorithm 6.7. Computing the forward and backward differences in a polytope p represented through an nD-EVM.

Algorithm 6.7 will be useful when we describe our procedure for extracting the boundary of an nD-OPP which is represented through the nD-EVM. Such procedure will be described in **Section 6.6**.

6.5.1. Performance of the Algorithm

The following key points define the conditions under which the execution time of **Algorithm 6.7** was measured:

- Our testing consider n = 2, 3, 4, 5.
- For each n we have generated 10,000 random nD-OPP's according to the following procedures:
  - Given a hypervoxelization representing nD-OPP's g we obtain their respective nD-EVM, that is $EVM_n(g)$.
  - According to **Theorem 5.13** $\pi_i\left(EVM_n(g)\right) = \bigcup_{k=1}^{np_i} EVM_{n-1}\left(\pi_i\left(\Phi_k^i\right)\right)$, and by **Theorem 5.16** such expression can be rewritten as $\pi_i\left(EVM_n(g)\right) = \bigcup_{k=1}^{np_i} EVM_{n-1}\left(\pi_i(S_{k-1}^i(g)) \otimes^* \pi_i(S_k^i(g))\right)$. By **Property 5.9** we have

$$\pi_i\left(S_{k-1}^i(g)\right) \otimes^* \pi_i\left(S_k^i(g)\right) = \left(\pi_i\left(S_{k-1}^i(g)\right) -^* \pi_i\left(S_k^i(g)\right)\right) \cup^* \left(\pi_i\left(S_k^i(g)\right) -^* \pi_i\left(S_{k-1}^i(g)\right)\right)$$

Hence, and by applying definition of backward and forward differences, we obtain:

$$\pi_i\left(EVM_n(g)\right)=\bigcup_{k=1}^{np_i} EVM_{n-1}\left(FD_k^i(g)\cup *BD_k^i(g)\right)$$

- We can compute the set $\pi_i\left(EVM_n(g)\right)$ by applying our projection operator (**Definition 5.10**) in a straightforward way to the set of extreme vertices in the EVM associated to polytope g. In the order hand, the set $\bigcup_{k=1}^{np_i} EVM_{n-1}\left(FD_k^i(g)\cup *BD_k^i(g)\right)$ is computed through our algorithm for Boolean operations (**Algorithm 6.4**) where $FD_k^i(p)$ and $BD_k^i(p)$ are included in the sets FD and BD which are the output of **Algorithm 6.7**. As a mechanism for identifying possible errors in our implementation of **Algorithm 6.7** we verify if $\pi_i\left(EVM_n(g)\right)$ and $\bigcup_{k=1}^{np_i} EVM_{n-1}\left(FD_k^i(g)\cup *BD_k^i(g)\right)$ contain exactly the same vertices. If this is the case then we store the operation's results and proceed to generate a new random nD-OPP for testing.

The **Table 6.16** shows some information related to our generated data. In **Chart 6.14** can be visualized the behavior of **Algorithm 6.7** with our set of nD-OPP's. In the same chart can be also visualized the associated trendline for each value of n whose associated equations are shown in **Table 6.17**.

n	Max	Min	Mean	Standard Deviation
2	5,472	0	2,788.70	1,549.56
3	5,368	0	3,148.00	1,485.30
4	5,458	0	3,210.50	1,488.60
5	5,288	0	2,743.87	1,457.05

Table 6.16. Some statistical characteristics of the set of 10,000 random nD-OPP's for testing of **Algorithm 6.7**.

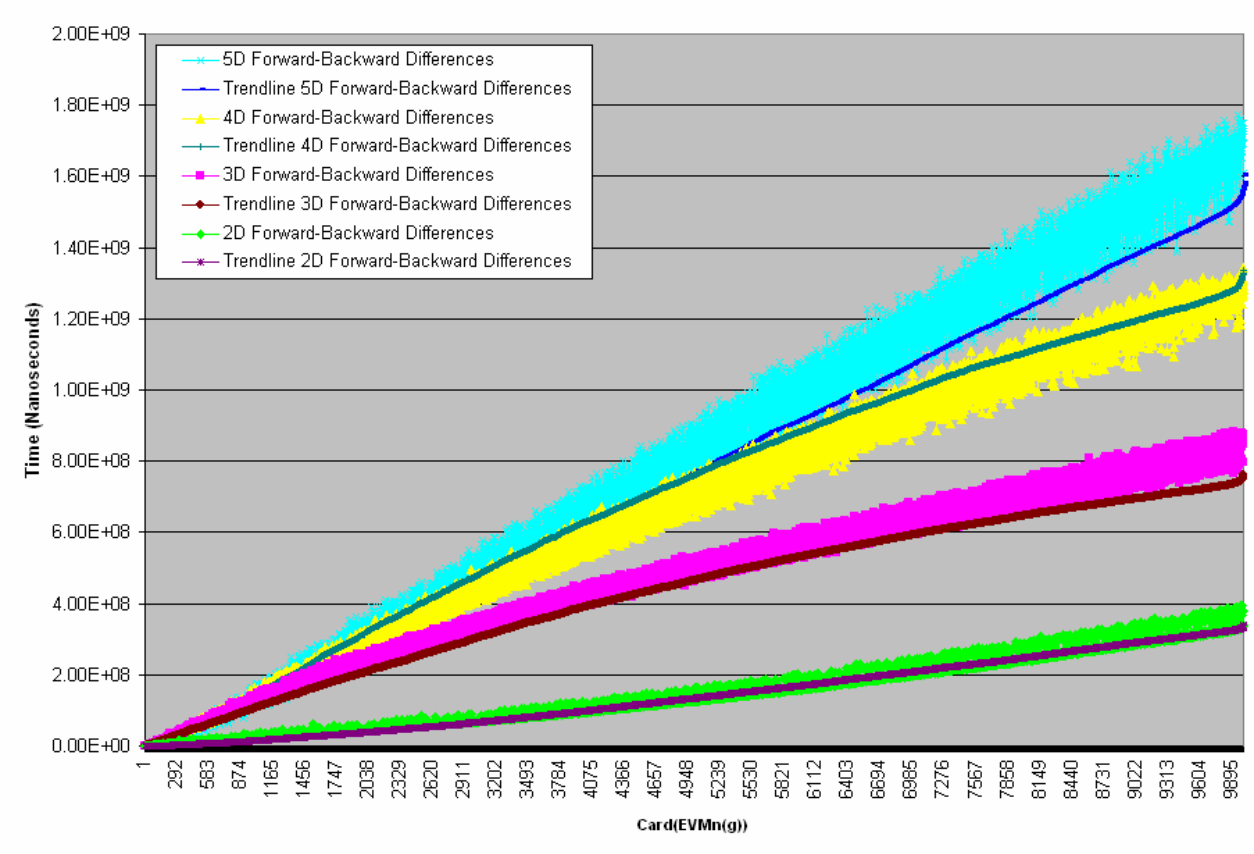


Chart 6.14. Comparing execution times for **Algorithm 6.7** for nD-OPP's with n = 2, 3, 4, 5.

n	Trendline $t = ax^b$	a	b	R^2
2	$t = 1{,}668.39\ x^{1.42}$	1,668.39	1.42	0.9859
3	$t = 65{,}377.04\ x^{1.09}$	65,377.04	1.09	0.9584
4	$t = 28{,}506.68\ x^{1.25}$	28,506.68	1.25	0.9859
5	$t = 54{,}637.82\ x^{1.20}$	54,637.82	1.20	0.9664

Table 6.17. Equations associated to the trendlines that describe execution time of **Algorithm 6.7** in the cases with n = 2, 3, 4, 5.

As we have proceeded in previous algorithms, now we determine an approximation surface for execution time of **Algorithm 6.7**. The associated equation is a function from $\mathbb{N}^2$ to $\mathbb{R}$ which has as arguments the number x of extreme vertices in the input polytope g, i.e., $x = \text{Card}(EVM_n(g))$, and the number of dimensions n. According to our analysis we have that the approximation surface is given by

$$t = 1{,}849.27\ x^{1.3} n^{1.64855}$$

In this case we have identified a coefficient of determination $R^2 = 0.9797$. The **Figure 6.16** shows the plotting of the above function and shows graphically an estimation of the execution time of **Algorithm 6.7** when the number of input extreme vertices is from 0 to 10,000 and when the number of dimensions is between 0 and 10.

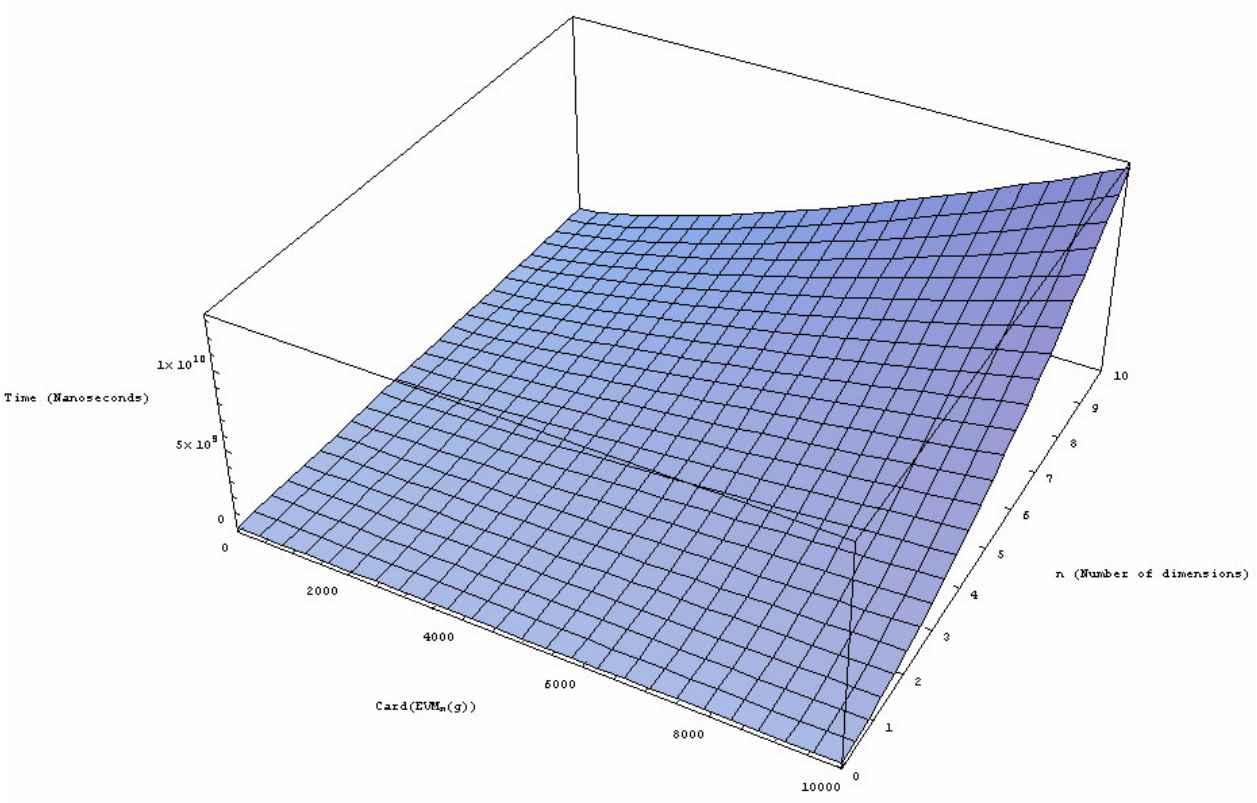


Figure 6.16. Plot of the approximation surface for execution time of **Algorithm 6.7**, $0 \leq \text{Card}(EVM_n(g)) \leq 10{,}000$; $0 \leq n \leq 10$.

Through the proposed approximation surface we have obtained the new trendlines presented in **Table 6.18** (the second and third columns show trendlines and coefficients of determination obtained from the data shown in **Chart 6.14**). Although the new trendline obtained for the case n = 2 has a coefficient $R^2 = 0.7454$, the remaining trendlines for cases n = 3, 4, 5 have coefficients above 0.94, which lead us to expect that given estimations for cases n = 7, 8 are good bounds for execution times of **Algorithm 6.7**.

n	Trendline $t = ax^b$	R^2	Trendline $t = ax^b$ (by fixing n in approximation surface)	R^2
2	$t = 1{,}668.39\ x^{1.42}$	0.9859	$t = 5{,}797.8\ x^{1.3}$	0.7454
3	$t = 65{,}377.04\ x^{1.09}$	0.9584	$t = 11{,}312.5\ x^{1.3}$	0.9443
4	$t = 28{,}506.68\ x^{1.25}$	0.9859	$t = 18{,}177.2\ x^{1.3}$	0.9935
5	$t = 54{,}637.82\ x^{1.20}$	0.9664	$t = 26{,}259.6\ x^{1.3}$	0.9899
6			$t = 35{,}466.8\ x^{1.3}$	
7			$t = 45{,}728.5\ x^{1.3}$	

Table 6.18. Fixing the n value in surface approximation for **Algorithm 6.7** in order to predict trendlines for n > 4.

6.6. Algorithms for Converting the nD-EVM To and From Other Schemes

This section deals with the process of converting the nD-EVM to and from other schemes for representing orthogonal polytopes. The **Sections 6.6.1** and **6.6.2** deal with converting the nD-EVM to and from Boundary Representations (**Section 2.2.3**), respectively. The **Section 6.6.3** covers conversions from Hyperspatial Occupancy Enumeration Models to the nD-EVM (see **Sections 2.2.5** and **2.2.6**). This work does not include conversions from the nD-EVM to Hyperspatial Occupancy Enumeration Models, because a general nD-OPP does not always decompose into identical cells arranged in a fixed regular grid.

6.6.1. The n-Dimensional Boundary Representations to nD-EVM conversion

A boundary representation of an nD-OPP p, must be able to provide, either directly or indirectly the set of (n-1)D cells incident to each edge of p. According to **Theorem 4.7**, if n is odd then an odd edge of p has an even number of incident (n-1)D cells; in the other hand, if n is even then an odd edge of p has an odd number of incident (n-1)D cells. Then, all those vertices that have exactly n perpendicular odd edges in p, by **Theorem 5.7** (see **Section 5.3**) will be Extreme Vertices. Thus, a Boundary Representation to nD-EVM algorithm would be as simple as collecting every vertex that belongs to n perpendicular odd edges, and discarding the remaining ones.

Any way, in this conversion process, [Aguilera98] points out that we must be aware of the boundary representation must represent a valid orthogonal pseudo-polytope p, otherwise the obtained result (if any) will not be meaningfull at all. Moreover, once the conversion has been performed, the potential set $EVM_n(p)$ should be validated using **Theorem 5.21** [Aguilera98].

6.6.2. The nD-EVM to n-Dimensional Boundary Representation Conversion

In **Section 2.2.3** we commented that a boundary representation can be seen as a **Boundary Tree** [Putnam86]. In the tree, each node is split into a component for each element that it bounds. An element (vertex, edge, etc.) will be represented several times inside the tree, one for each boundary that it belongs to. See **Figure 6.17** for a cube's boundary tree.

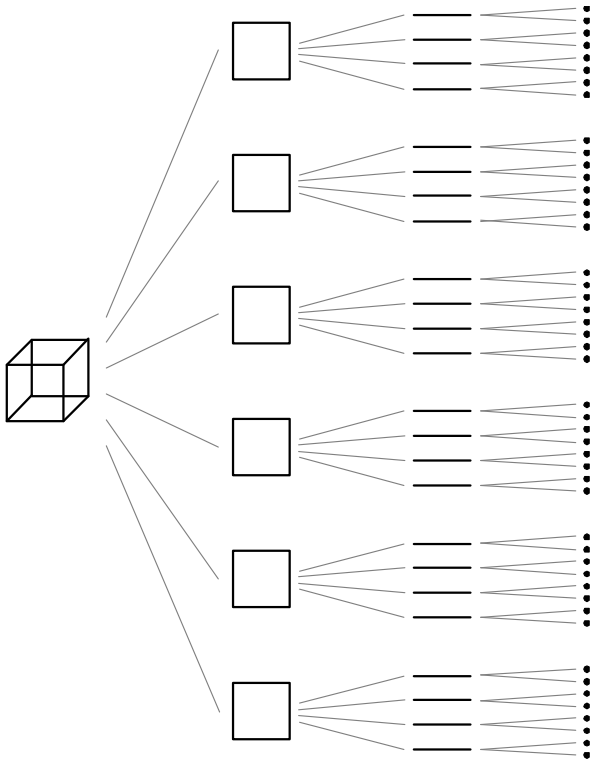


Figure 6.17. The boundary tree associated to a 3D cube.

The way we convert an nD-OPP represented through the nD-EVM to a boundary representation will consider the reconstruction of the boundary tree associated to such nD-OPP. According to **Theorem 5.18**, in an nD-OPP p, forward differences $FD_k^i(p)$ are the (n-1)D cells on $\Phi_k^i(p)$ whose normal vectors point to the positive side of the coordinate axis X_i which is perpendicular to $\Phi_k^i(p)$, while backward differences $BD_k^i(p)$ are the (n-1)D cells on $\Phi_k^i(p)$ whose normal vectors point to the negative side of the coordinate axis X_i which is perpendicular to $\Phi_k^i(p)$. Such forward and backward differences can be computed through **Algorithm 6.7**. In fact, all $FD_k^i(p)$ and $BD_k^i(p)$ from an nD-OPP are (n-1)D-OPP's embedded in (n-1)D space because by **Definition 5.21** $FD_k^i(p) = \left(\pi_i\left(S_{k-1}^i(p)\right) -^* \pi_i\left(S_k^i(p)\right)\right)$ and $BD_k^i(p) = \left(\pi_i\left(S_k^i(p)\right) -^* \pi_i\left(S_{k-1}^i(p)\right)\right)$. If such forward and backward differences were computed through our proposed algorithm then they are expressed as $EVM_{n-1}(FD_k^i(p))$ and $EVM_{n-1}(BD_k^i(p))$. If we apply again **Algorithm 6.7** to such (n-1)D-OPP's we will get new forward and backward differences that correspond to the (n-2)D oriented cells on the boundary of such (n-1)D-OPP's. These new forward and backward differences are themselves (n-2)D-OPP's represented through the EVM. Hence, by applying again **Algorithm 6.7** to them we obtain their associated (n-3)D oriented cells grouped as forward and backward differences. This procedure generates a recursive process which descends in the number of dimensions. In each recursivity level we obtain forward and backward differences associated to the input (n-k)D-OPP's. The basic case is present when n = 1. In this situation the boundary of a 1D-OPP is described by the beginning and ending extreme vertices of each one of its composing segments. Forward differences in a 1D-OPP are composed by the ending vertices while backward differences are composed by the beginning vertices. In **Figure 6.18** we present the extraction of forward and backward differences according to the procedure we have described. Because **Algorithm 6.7** considers such extraction only for differences perpendicular to the first coordinate of the input EVM then in our example we will consider such situation.

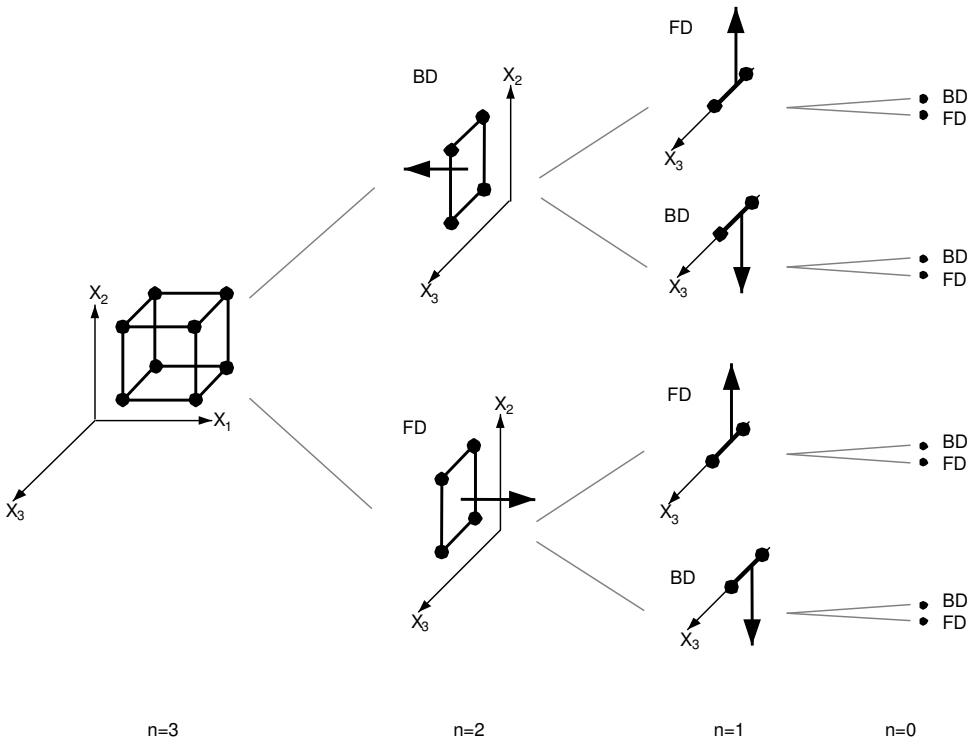


Figure 6.18. Computing forward and backward differences for a cube and some of its boundary elements (See text for details).

In **Figure 6.18** we compute first forward and backward differences, perpendicular to X_1-axis, in the 3D cube. By assuming that such differences were computed through **Algorithm 6.7** then we have that the output set BD contains only the face whose normal points to the negative side of X_1-axis, while set FD contains only the face whose normal points to the positive side of X_1-axis. By applying again **Algorithm 6.7** over such pair of faces we have in each case forward and backward differences perpendicular to X_2-axis (assuming that the next coordinate in the EVM associated to the cube is X_2). As seen in **Figure 6.18**, the set FD contains an edge whose normal vector points to the positive side of X_2-axis and the set BD contains an edge whose normal vector points to the negative side of X_2-axis. By computing forward and backward differences associated to such edges we get the extreme vertices shown at the right side of **Figure 6.18**. As seen in our example, the tree we have obtained has the characteristic that each one of its nodes is split into a component for the elements that it bound. We say in this case that we have obtained a Differences Tree associated to a cube originally expressed in the 3D-EVM.

A recursive procedure can be performed in order to build the Differences Tree associated to an nD-OPP represented through the EVM. In fact, such Differences Tree can be associated to a tree data structure where a node, which corresponds to a boundary element, contains pointers to boundary elements that it bound. Moreover, additional information or processing, according to the application, can be added or performed to the nodes in the tree. For example, the normal vector could be added as a field in a node in order to indicate the orientation of the referred boundary cell corresponding to the node. The **Algorithm 6.8** implements the above proposed ideas. Input parameters for our algorithm require the EVM associated to an nD-OPP p, the number n of dimensions, and a reference (pointer) to the tree data structure associated to the boundary tree. By the moment, when we refer to a Differences Tree we denote a tree generated according **Algorithm 6.8**. That is, as pointed previously, the new algorithm depends on **Algorithm 6.7** which performs the extraction of forward and backward differences perpendicular to the first coordinate of the input EVM. In **Section 6.6.2.2** we will discuss methodologies for the extraction of backward and forward differences perpendicular to remaining main axes.

```
Input: An nD-EVM p.
       The number n of dimensions.
       A pointer to Differences Tree t.
Procedure GetDifferencesTree(EVM p, int n, Tree t)
       EVM FDcurr     // Current forward difference.
       EVM BDcurr     // Current backward difference.
       FD = ∅         // FD stores (n-1)D-EVM's corresponding to forward differences in p.
       BD = ∅         // BD stores (n-1)D-EVM's corresponding to backward differences in p.
       Tree tn        // A leaf to be added to Differences Tree t
       if(n = 1) then
              {FD, BD} = GetForwardBackwardDifferences(p, 1)      // Call to Algorithm 6.7
              for each vertex v in FD do
                     Initialize(tn)
                     Process(tn, v)
                     Link(t, tn)
              end-of-for
              for each vertex v in BD do
                     Initialize(tn)
                     Process(tn, v)
                     Link(t, tn)
              end-of-for
       else
              {FD, BD} = GetForwardBackwardDifferences(p, n)      // Call to Algorithm 6.7
              // We process Differences Trees for each forward difference in p.
              for each forward difference in FD do
                     FDcurr = FD.next( )
                     // Check if FDcurr is not empty to avoid adding empty Differences subtrees.
                     if(Not(EndEVM(FDcurr))) then
                            Initialize(tn)
                            Process(tn, FDcurr)
                            GetDifferencesTree(FDcurr, n-1, tn)         // Recursive call
                            Link(t, tn)
                     end-of-if
              end-of-for
              // We process Differences Trees for each backward difference in p.
              for each backward difference in BD do
                     BDcurr = BD.next( )
                     // Check if BDcurr is not empty to avoid adding empty Differences subtrees.
                     if(Not(EndEVM(BDcurr))) then
                            Initialize(tn)
                            Process(tn, BDcurr)
                            GetDifferencesTree(BDcurr, n-1, tn)         // Recursive call
                            Link(t, tn)
                     end-of-if
              end-of-for
       end-of-else
end-of-procedure
```

Algorithm 6.8. Processing the Differences Tree of an nD-OPP p through forward and backward differences
(Forward and Backward differences are perpendicular to the axis associated to the first coordinate in the extreme vertices of p).

Our algorithm proceeds as follows when n > 1:

- We compute forward and backward differences perpendicular to the axis associated to the first coordinate of the vertices in the input EVM.
- Once the differences have been computed through **Algorithm 6.7**, we proceed to process each one of them. In each iteration of the first loop, a non empty forward difference is extracted from set FD and a new leaf *tn* to be added to the Differences Tree is initialized. Such leaf is associated to the current forward difference *FDcurr*. According to the needs of the application, the leaf *tn* and the difference *FDcurr* are processed through a generic process (called *Process*) which performs the desired actions upon *tn* and *FDcurr*. Because *FDcurr* is a (n-1)D-OPP then a recursive call to the algorithm is performed in order to compute its corresponding forward and backward differences. After returning from the recursive call we proceed to link the current node *tn* to the input Differences Tree *t*. Depending of the recursivity level, the node *tn* can be pointing to the Differences subtree associated to the current forward difference.

- Once we have processed forward differences in the set FD, we proceed to process each one of the backward differences in the set BD in the same way as the previous loop. Such processes are performed in the second loop of the algorithm.

In the basic case, when n = 1, we call **Algorithm 6.7** in order to get forward and backward differences associated to the input 1D-OPP. Set FD contains the ending vertices of each one of the segments that compose to the input 1D-OPP. Such vertices are processed with the leaf node *tn* which is then added to the input Differences Tree *t.* The set BD contains the beginning vertices of each one of the segments that compose to the input 1D-OPP. These vertices are processed with their corresponding leaf node and it is added to the input tree *t*.

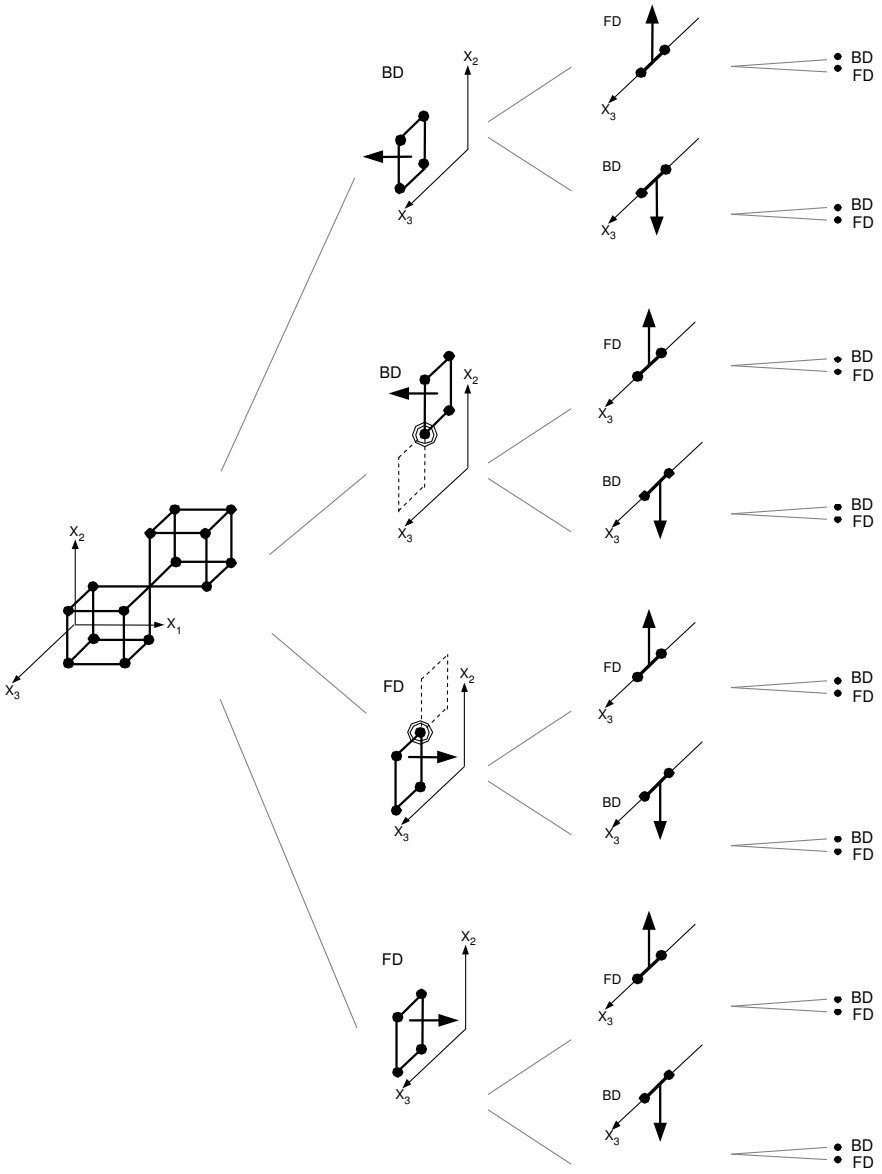


Figure 6.19. Differences tree associated to a 3D-OPP q composed by two cubes sharing a vertex (See text for details).

When we compute the Differences Tree of a polytope through forward and backward differences some situations should be observed. Consider the 3D-OPP q shown in **Figure 6.19**. Such 3D-OPP q is composed by two cubes that share a vertex. Such shared vertex is not included in $EVM_3(q)$ because it is a non-manifold vertex with six incident odd edges. When we compute backward differences perpendicular to X_1-axis we can observe that the EVM of the backward difference on the couplet where the vertex adjacency takes place

contains precisely the projection of such shared vertex. It is indicated in **Figure 6.19** by a double circle. In the same figure, the face on that couplet but with opposite orientation is shown in dotted lines. Such face is not included in the backward difference but in the forward difference, where also the projection of the shared vertex by the cubes is present in the EVM associated to such forward difference. **Figure 6.19** exemplifies a situation where projections of non-manifold vertices are obtained after computing backward and forward differences, and therefore, they are included in the Differences Tree. The reason behind this phenomenon arises from the fact that the faces on the couplet where the non-extreme vertex in embedded have opposite orientations. In the **Figure 6.20** we have a situation where a 3D-OPP r is composed by two cubes sharing an edge.

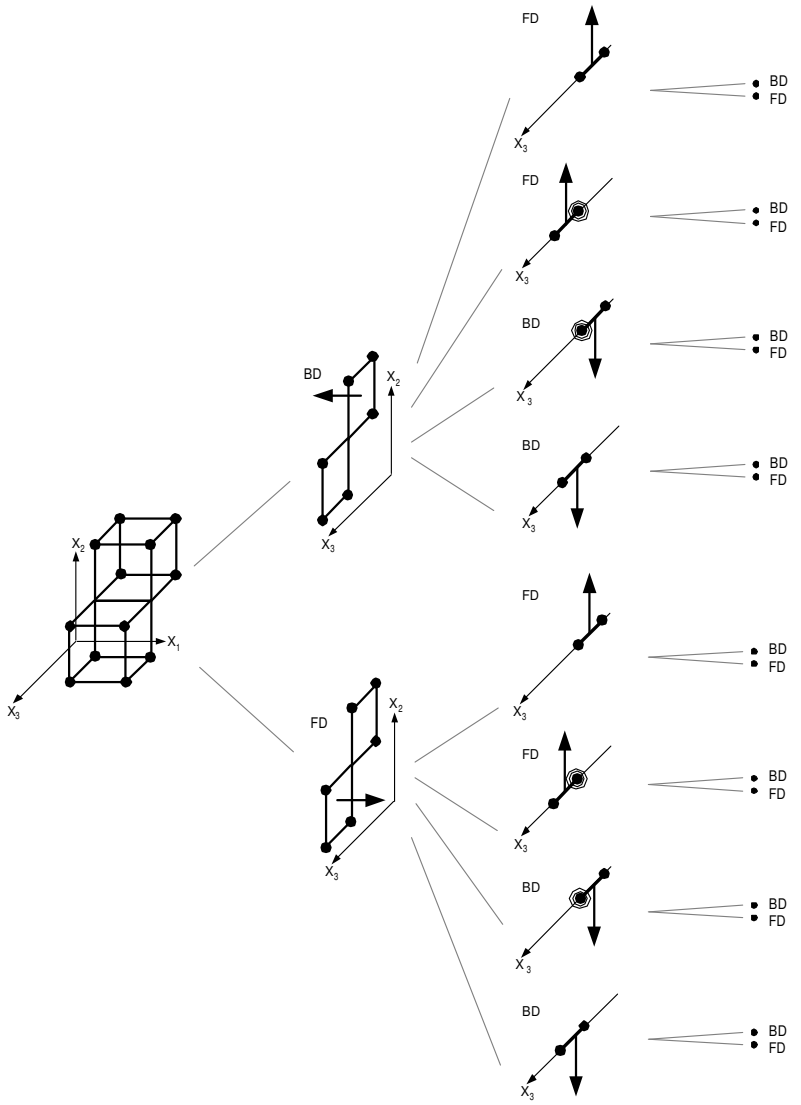


Figure 6.20. Differences Tree associated to a 3D-OPP r composed by two cubes sharing an edge (See text for details).

As seen in **Figure 6.20** the vertices included in the edge adjacency are not included in EVM(r). The pair of faces on the two couplets perpendicular to X_1 axis have the same orientation, hence, the 2D-EVM in the backward difference consider both of them and the situation is the same with the 2D-EVM in the forward difference. The projections of the vertices included in the edge adjacency between the two cubes lead to a 2D non-manifold vertex and therefore the projected vertex is not included in both 2D-EVM's. Consider the couplets perpendicular to X_2-axis in the 2D forward and backward differences. Edges included in such couplets have opposite orientations hence its 1D forward and backward differences contain only one segment. The projection of the non-manifold vertex is obtained after computing the differences, and therefore, it is included in the boundary tree. It is indicated in **Figure 6.20** by a double circle.

Because of the foundations behind the nD-EVM we have that non-extreme vertices do not belong to the EVM. However, projections of such vertices can be obtained by successively computing, as seen in the above two examples, kD forward and backward differences, k = n-1, n-2, …, 0. By this way, the last level in the Differences Tree contains the projections of all the vertices included in an nD-OPP, with some of them duplicated. Computing successive forward and backward differences provides us a new methodology for obtaining non-extreme vertices from the EVM associated to an nD-OPP. The first methodology was presented in **Theorem 5.9**. One of the advantages of our new methodology is that it provides us the Differences Tree of an nD-OPP.

Let's consider a third case to analyze and which can be possibly present in the Differences Trees we build according to **Algorithm 6.8**. See **Figure 6.21**.

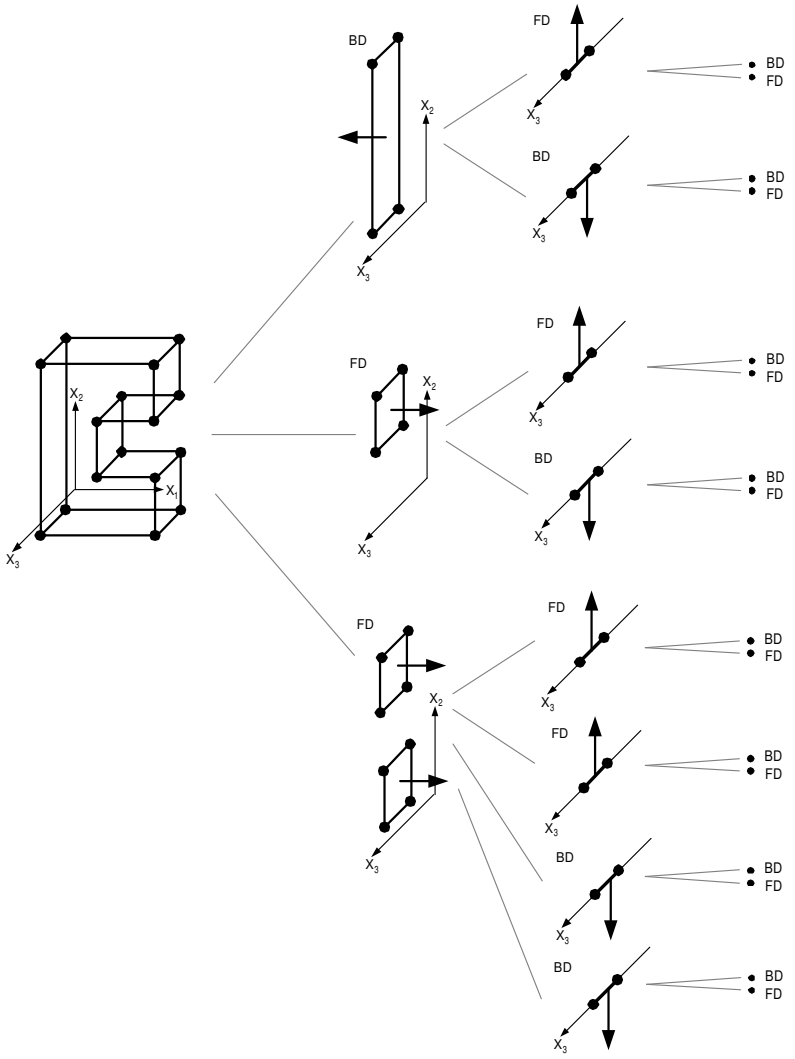


Figure 6.21. Differences Tree associated to a 3D-OPP. One of the nodes in the structure has two disjoint faces with the same orientation (See text for details).

As can be observed in **Figure 6.21** we have the case when one of the nodes in the tree has associated two (or more) disjoint cells with the same orientation. This kind of situation can be present in any level of the tree. In this work we will deal with nodes with two or more disjoint cells taking no action when they are present because our algorithms presented in the following sections are not affected by them.

6.6.2.1. Performance of the Algorithm

In this section we present some results related to the measured execution times for **Algorithm 6.8** in the cases for n = 2, 3, 4, 5. As previously proceeded, we generated 10,000 random nD-OPP's for each considered value of n. The **Table 6.19** shows some statistical characteristics of the sets of generated nD-OPP's.

n	Max	Min	Mean	Standard Deviation
2	5,432	0	2,788.98	1,549.72
3	5,364	0	3,148.42	1,485.08
4	5,528	0	3,211.01	1,488.91
5	5,328	0	2,744.30	1,456.52

Table 6.19. Some statistical characteristics of the set of 10,000 random nD-OPP's for testing of **Algorithm 6.8**.

In **Chart 6.15** can be visualized the behavior of **Algorithm 6.8** with our set of nD-OPP's. In the same chart can be also visualized the associated trendline for each value of n whose associated equations are shown in **Table 6.20**.

n	Trendline $t = ax^b$	a	b	R^2
2	$t = 2{,}065.31\ x^{1.39}$	2,065.31	1.39	0.9623
3	$t = 92{,}413.80\ x^{1.09}$	92,413.80	1.09	0.9693
4	$t = 23{,}904.62\ x^{1.35}$	23,904.62	1.35	0.9715
5	$t = 65{,}509.94\ x^{1.28}$	65,509.94	1.28	0.9644

Table 6.20. Equations associated to the trendlines that describe execution time of **Algorithm 6.8** in the cases with n = 2, 3, 4, 5.

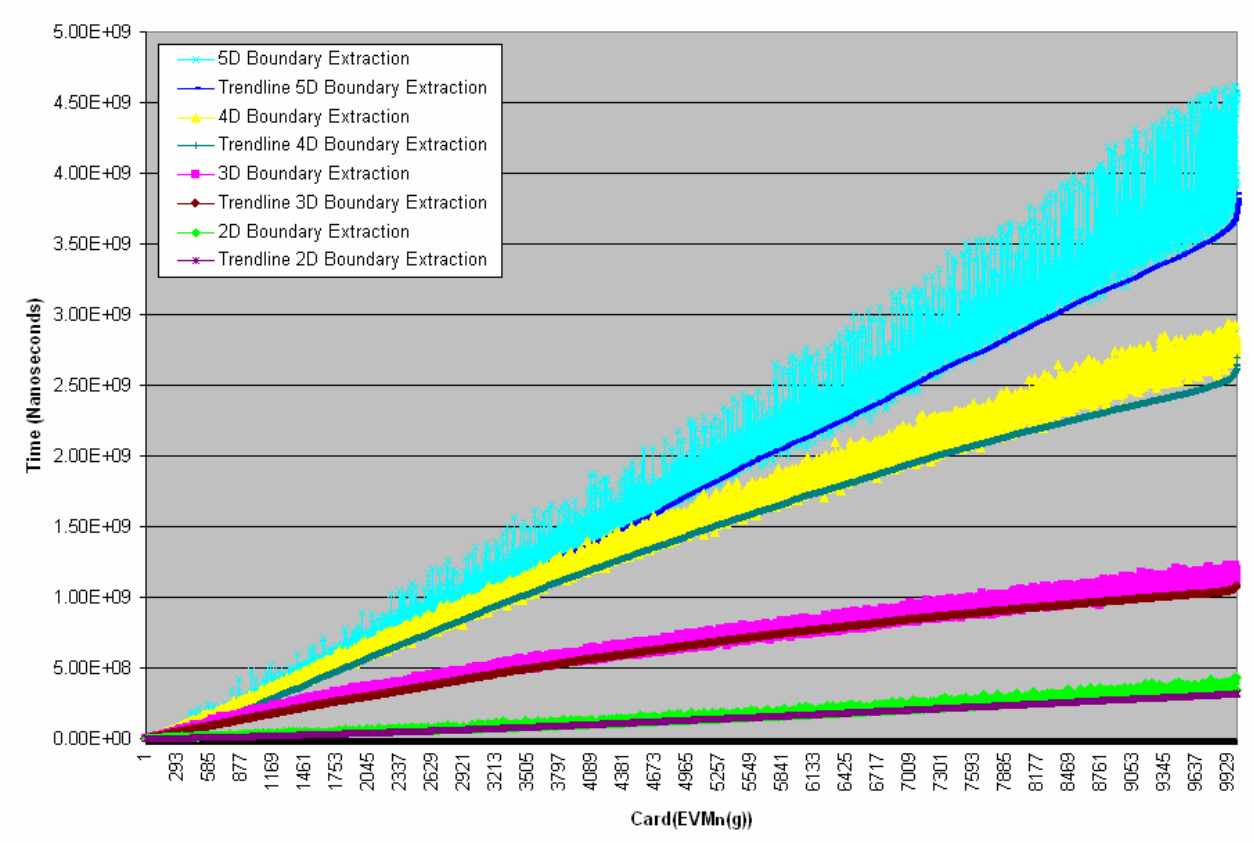


Chart 6.15. Comparing execution times for **Algorithm 6.8** for nD-OPP's with n = 2, 3, 4, 5.

As we have proceeded in previous algorithms, we now determine an approximation surface for execution time of **Algorithm 6.8**. The associated equation is a function from $\mathbb{N}^2$ to $\mathbb{R}$ which has as arguments the number x of extreme vertices in the input polytope g, i.e., $x = \text{Card}(EVM_n(g))$ and the number of dimensions n. According to our analysis we have that the approximation surface is given by

$$t = 1{,}160.7\ x^{1.3189} n^{2.3720}$$

In this case we have identified a coefficient of determination $R^2 = 0.9871$. The **Figure 6.22** shows the plotting of the above function and shows graphically an estimation of the execution time of **Algorithm 6.8** when the number of input extreme vertices is from 0 to 10,000 and when the number of dimensions is between 0 and 10. Through the

proposed approximation surface we have obtained the new trendlines presented in **Table 6.21** (the second and third columns show trendlines and coefficients of determination obtained from the data shown in **Chart 6.15**). Although the new trendline obtained for the case n = 2 has a coefficient $R^2 = 0.2566$, the remaining trendlines for cases n = 3, 4, 5 have coefficients above 0.95, which lead us to expect that given estimations for cases n = 7, 8 are good bounds for execution times of **Algorithm 6.8** (when we analyzed execution times for **Algorithm 6.7** we had a similar situation with the new trendline for n = 2).

n	Trendline $t = ax^b$	R^2	Trendline $t = ax^b$ (by fixing n in approximation surface)	R^2
2	$t = 1{,}668.39\ x^{1.42}$	0.9859	$t = 6{,}008.46\ x^{1.3189}$	0.2566
3	$t = 65{,}377.04\ x^{1.09}$	0.9584	$t = 15{,}719.96\ x^{1.3189}$	0.9501
4	$t = 28{,}506.68\ x^{1.25}$	0.9859	$t = 31{,}103.28\ x^{1.3189}$	0.9731
5	$t = 54{,}637.82\ x^{1.20}$	0.9664	$t = 52{,}805.21\ x^{1.3189}$	0.9844
6			$t = 81{,}375.70\ x^{1.3189}$	
7			$t = 117{,}298.52\ x^{1.3189}$	

Table 6.21. Fixing the n value in surface approximation for **Algorithm 6.8** in order to predict trendlines for n > 4.

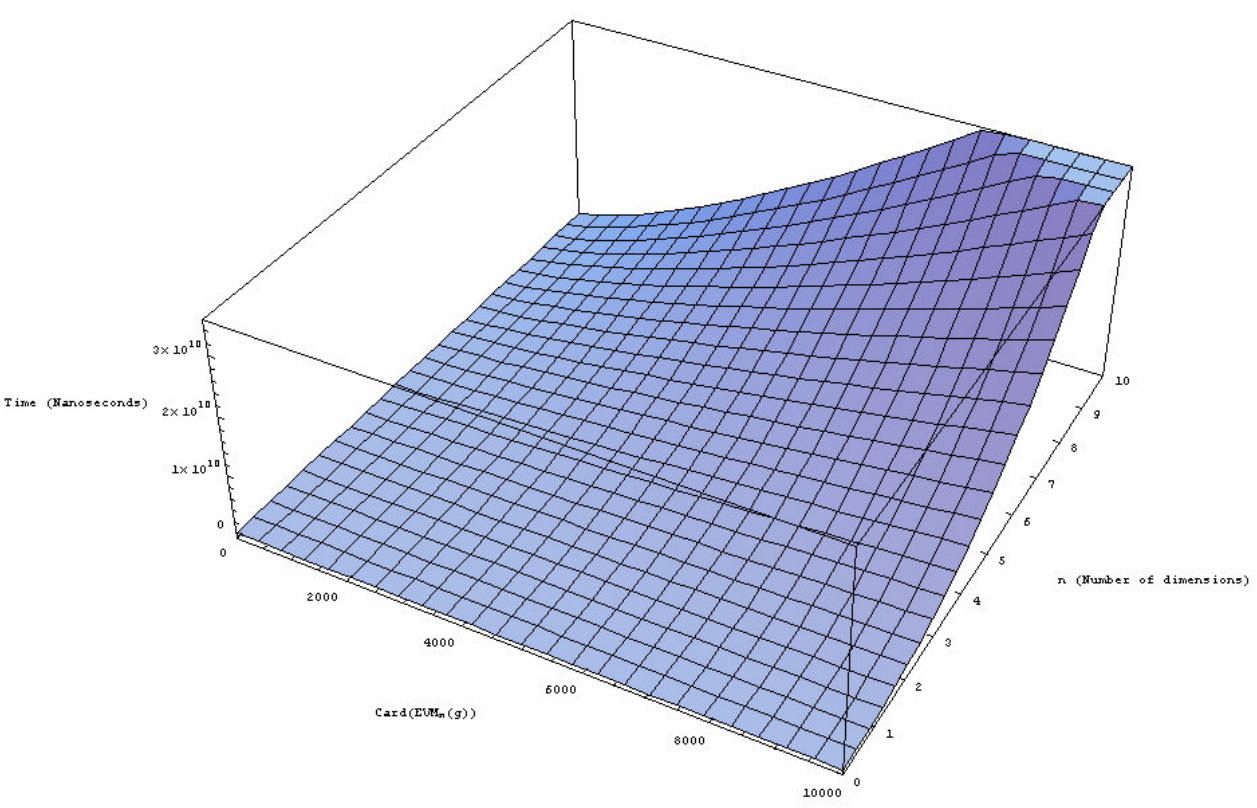


Figure 6.22. Plot of the approximation surface for execution time of **Algorithm 6.8**, $0 \leq \text{Card}(EVM_n(g)) \leq 10{,}000$; $0 \leq n \leq 10$.

6.6.2.2. Computing the Complete Differences Tree of an nD-OPP in the nD-EVM

Algorithm 6.8 extracts, from an nD-OPP p, the (n-1)D backward/forward differences perpendicular to X_1-axis, by assuming the coordinates of extreme vertices have the ordering $X_1X_2\cdots X_{n-1}X_n$. Then, for all $FD_k^1(p)$ and $BD_K^1(p)$, it extracts their (n-2)D forward/backward differences perpendicular from X_2-axis, then the (n-3)D forward/backward differences perpendicular from X_3-axis, and so on until the basic case is reached. As pointed out in **Section 6.6.2**, the **Algorithm 6.8** provides a Differences Tree whose nodes refer to forward/backward differences perpendicular to the first coordinate in the input EVM's (See **Figures 6.18**, **6.19**, **6.20** and **6.21**). Consider the cube shown in **Figure 6.18**. Its EVM has the ordering $X_1X_2X_3$, hence, **Algorithm 6.8** provides at the main call 2D forward/backward differences perpendicular to X_1-axis. In the first recursive call it computes 1D forward/backward

differences perpendicular to X_2-axis. In the second recursive call, where it reaches the basic case, returns vertices along X_3-axis. Now, by sorting coordinates in the cube's vertices as $X_2X_3X_1$ and applying **Algorithm 6.8** we have the Differences Tree shown in **Figure 6.23**.

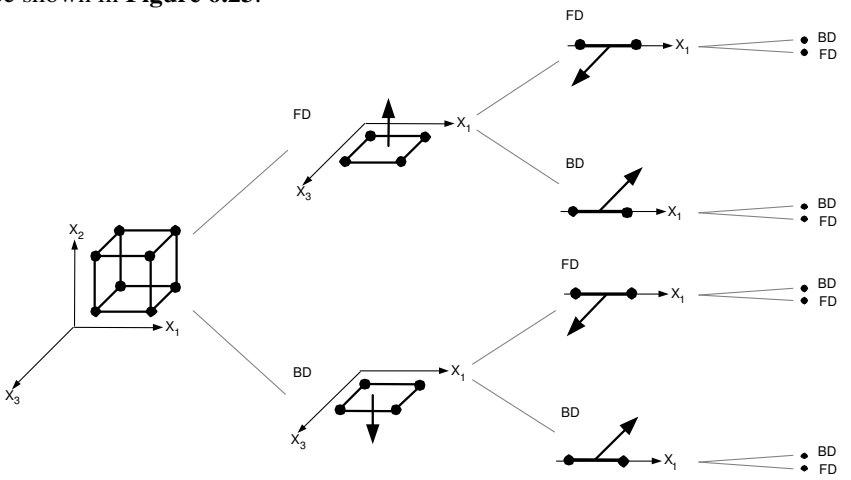


Figure 6.23. Computing the Differences Tree for a cube whose extreme vertices have the coordinates ordering $X_2X_3X_1$ (See text for details).

Through the ordering $X_2X_3X_1$ we have access, according to **Figure 6.23**, to forward/backward differences perpendicular to X_2-axis. Then, we have access to 1D forward/backward differences perpendicular to X_3-axis, and finally, in the basic case of **Algorithm 6.8**, we found vertices along X_1-axis. Now consider coordinates ordering $X_3X_1X_2$, hence, we have the Differences Tree in **Figure 6.24**. In this sense, we have access to the cube's oriented faces perpendicular to X_3-axis, edges perpendicular to X_1-axis and finally to vertices along X_2-axis.

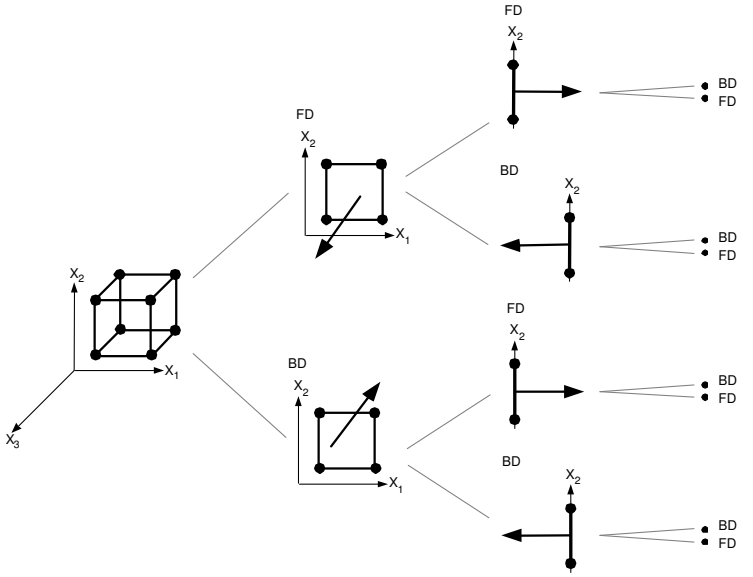


Figure 6.24. Computing the Differences Tree for a cube whose extreme vertices have the coordinates ordering $X_3X_1X_2$ (See text for details).

As seen in **Figures 6.18**, **6.23** and **6.24**, the three Differences Trees have the same root, but their associated subtrees differ according to the coordinates ordering. Now, we will define the Cube's Complete Differences Tree as the union of the boundary trees each one obtained through the coordinates ordering $X_1X_2X_3$, $X_2X_3X_1$ and $X_3X_1X_2$ and the respective application of **Algorithm 6.8**. See **Figure 6.25**.

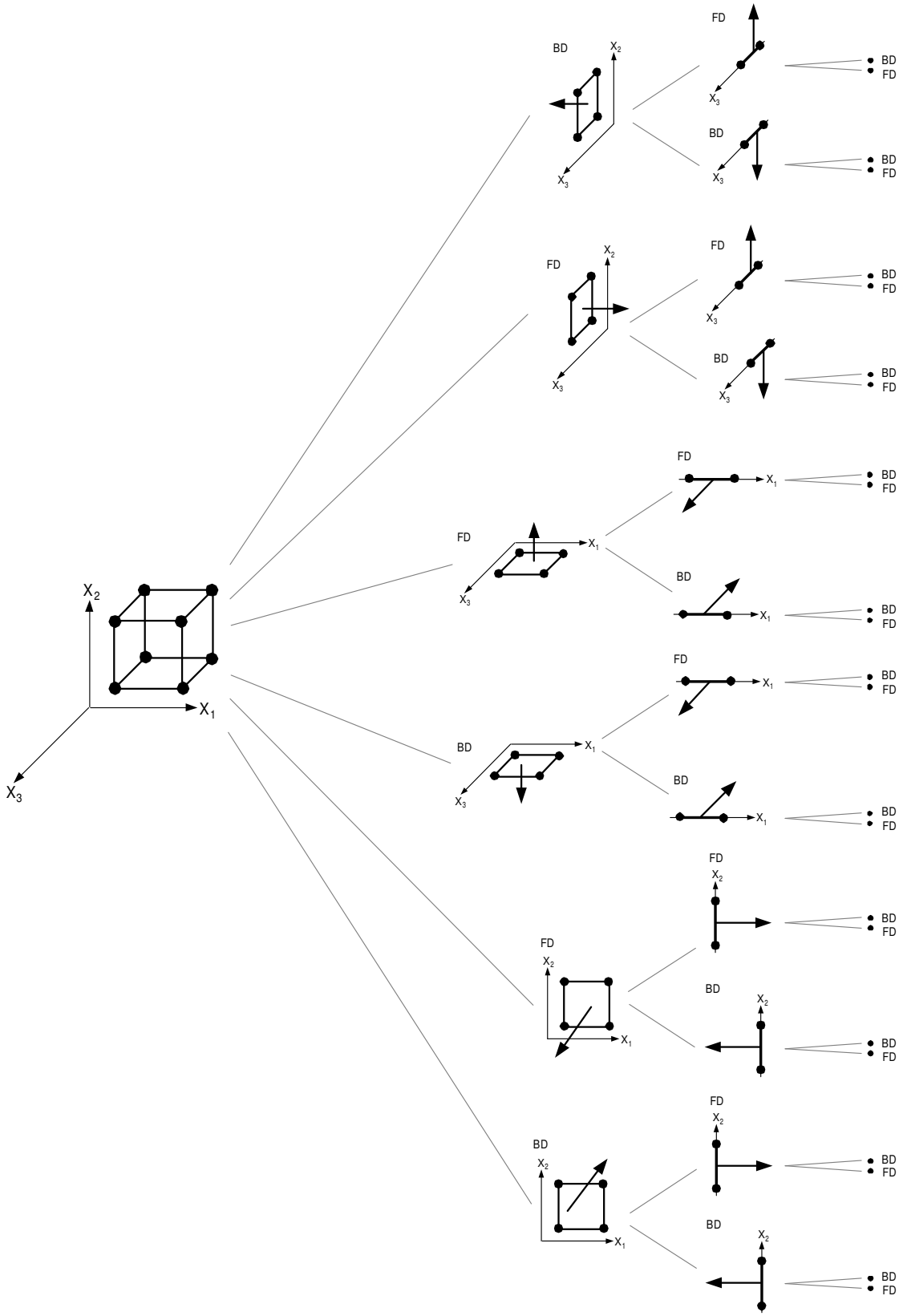


Figure 6.25. The Complete Differences Tree for a 3D cube (See text for details).

Let p be an nD-OPP. We assume the coordinates ordering in $EVM_n(p)$ is given by $X_1X_2\cdots X_{n-1}X_n$, hence, starting from that permutation we have the following n-1 permutations given by

$$X_2X_3\cdots X_nX_1$$
$$X_3X_4\cdots X_1X_2$$
$$\vdots$$
$$X_{n-1}X_n\cdots X_{n-3}X_{n-2}$$
$$X_nX_1\cdots X_{n-2}X_{n-1}$$

The **Algorithm 6.9** computes the Complete Differences Tree associated to p. The way it works is simple:

- A pointer *t* to a tree data structure is initialized and it, together with $EVM_n(p)$, is manipulated through a generic process (called *Process*) according to the needs of the application. Such pointer *t* is in fact the root of the Complete Differences Tree associated to p.
- A main loop is maintained while each one of the orderings, that is permutations, in the set $\{X_1X_2\cdots X_{n-1}X_n,\ X_2X_3\cdots X_nX_1,\ X_3X_4\cdots X_1X_2,\ \ldots,\ X_{n-1}X_n\cdots X_{n-3}X_{n-2},\ X_nX_1\cdots X_{n-2}X_{n-1}\}$ is used for sorting $EVM_n(p)$. Such sorting is performed by calling procedure *SortEVM*. Given a permutation $X_{\alpha_1}X_{\alpha_2}\cdots X_{\alpha_{n-1}}X_{\alpha_n}$ SortEVM sorts the extreme vertices of p first according to the coordinate X_{α_1}, after according to the coordinate X_{α_2}, and so on until p is sorted according to coordinate X_{α_n}. Following the calling to *SortEVM*, it is performed the calling to **Algorithm 6.8**. Through the procedure *GetDifferencesTree* we obtain the subtree that contains (n-1)D forward/backward differences perpendicular to X_{α_1}-axis, then the (n-2)D forward/backward differences perpendicular to X_{α_n}-axis, and so on until the level than contains their leaves is composed by vertices along X_{α_n}-axis. Such subtree is attached to the pointer *t* which performs the role of root node in the Complete Differences Tree of p (the linking takes place in **Algorithm 6.8**).

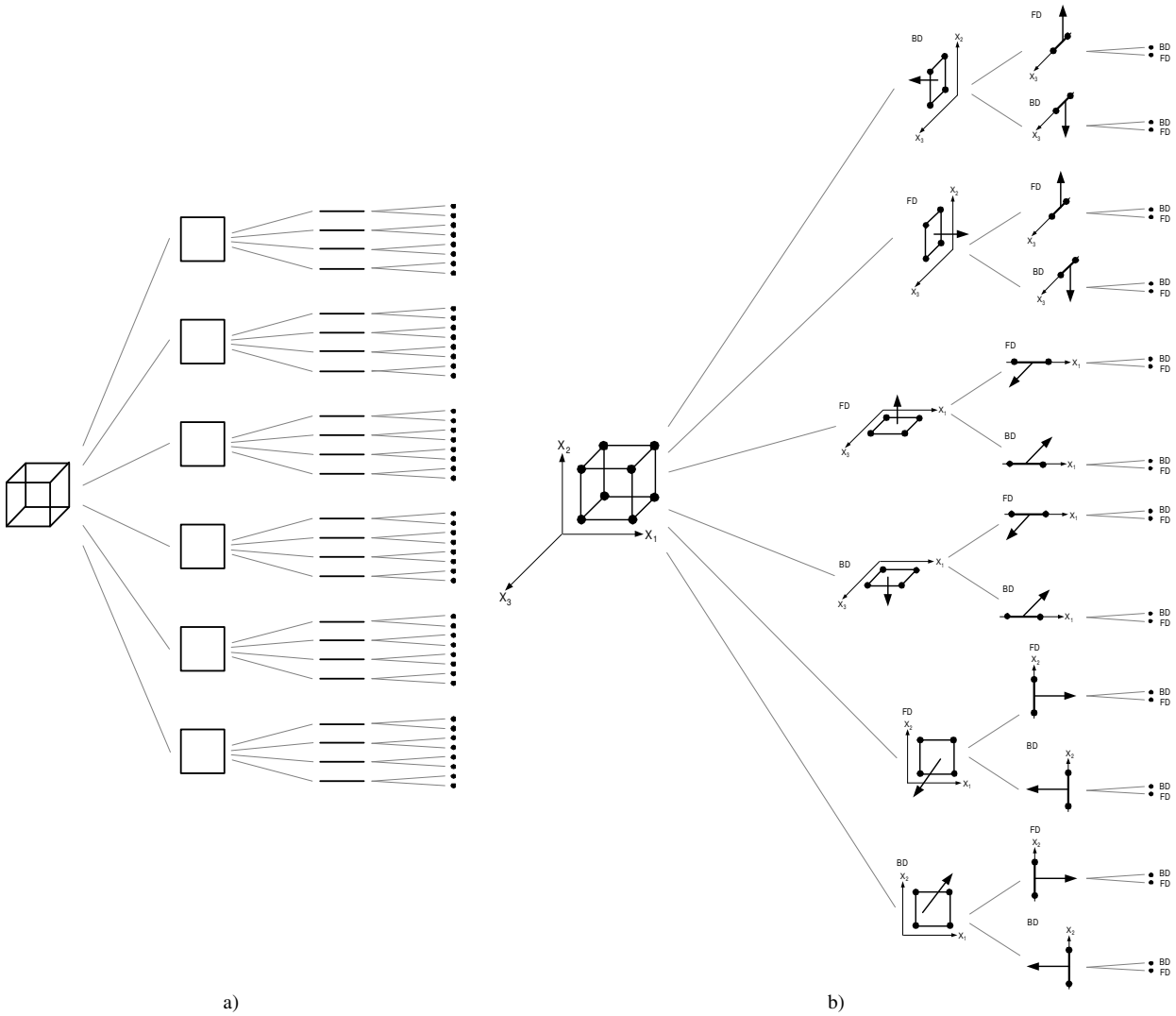
```
Input: An nD-EVM p.
       The number n of dimensions.
Output: A pointer to the Complete Differences Tree associated to p.
procedure GetCompleteDifferencesTree(EVM p, int n)
       Tree t         // The root of the Complete Differences Tree associated to p
       Initialize(t)
       Process(t, p)
       for sorting in {X1X2···Xn-1Xn, X2X3···XnX1, X3X4···X1X2, …, Xn-1Xn···Xn-3Xn-2, XnX1···Xn-2Xn-1} do
              SortEVM(p, n, sorting)
              /* We call Algorithm 6.8 and get the Differences Tree according to the current
                  sorting of p. */
              GetDifferencesTree(p, n, t)
       end-of-for
       return t
end-of-procedure
```

Algorithm 6.9. Computing the Complete Differences Tree of an nD-OPP expressed through the nD-EVM.

By comparing our Complete Differences Tree for the cube, which is shown in **Figure 6.26.b**, with its Boundary Tree, as defined in **Section 2.2.3**, as seen in **Figure 6.26.a**, it seems that the first one is incomplete respect to the second one. Although both trees coincide in the description of faces in the cube, the level corresponding to description of edges is bounded in the Complete Differences Tree because the way we have obtained Forward and Backward differences. However, each edge in the cube is present in our Complete Differences Tree because if one of them was not obtained through a given face under certain coordinates ordering, it was obtained by means of one of the 2 remaining permutations of coordinates. Speaking in a more general way, if an specific kD cell in an nD-OPP was not obtained through computing forward and backward differences of a (k+1)D cell under a coordinates ordering, it can be obtained by means of the n-1 remaining permutations of coordinates. Such kD cell in our Complete Differences Tree will be linked to the (k+1)D cell that generated it.

Some applications can find our Complete Differences Tree useful in the sense that it provides access to all the oriented boundary cells of an nD-OPP. However other applications can find that some information about the connectivity between boundary elements can be lost or hidden. For example, according to **Figure 6.26.b**, the Complete Differences Tree does not explicitly provide information about all the boundary edges in the face whose normal points towards the negative side of X_1-axis, because our tree presents explicitly two of the four edges. Such specific pair of edges was obtained starting from the coordinates ordering of the 2D-EVM associated to the face when its forward and backward differences were computed using **Algorithm 6.8**. A possible solution in order to

have access to those two "hidden" edges is to consider the two possible coordinates sortings of the 2D-EVM associated to the current face in the cube. The first ordering will provide the original pair of edges in the tree and the new sorting will provide the remaining two which will generate new subtrees which can be linked to the structure and processed in order to obtain its boundary elements. This process of sorting coordinates should be added to **Algorithm 6.8** with the objective to take in account the processing of all boundary elements on an OPP expressed through the EVM. Hence, the final obtained tree will correspond to a boundary tree as defined in **Section 2.2.3**.



a) b)

Figure 6.26. a) The boundary tree associated to a 3D cube as defined in **Section 1.2.3**.
b) The Complete Differences Tree associated to a 3D cube as computed through **Algorithm 6.9**.

6.6.3. Hyperspatial Occupancy Enumeration Models to the nD-EVM

This section deals with the process of converting other schemes for the modeling of nD-OPP's to the nD-EVM. We will consider particularly two conversions:

- nD Hypervoxelizations to the nD-EVM.
- 2^n-trees to the nD-EVM.

Both considered schemes correspond to the category of the Hyperspatial Occupancy Enumerations. A model in this category is a set of black and white cells or nodes where each cell is a convex orthogonal polytope. The set of black cells represents an nD-OPP p whose vertices coincide with some of the black cells' vertices. A hyperspatial occupancy enumeration model should provide means for generating a list of all 2^n vertices of each black cell.

Each of these vertices may be common to (surrounded by) up to 2^n black cells. So, according to **Theorem 5.1**, if a vertex is surrounded by an odd number of black cells (hyper-octants of a classical 2^n-tree) then it is an Extreme Vertex. Thus, a hyperspatial occupancy enumeration model to nD-EVM conversion algorithm would be as simple as collecting every vertex that belongs to and odd number of cells, and discarding the remaining vertices [Aguilera98].

Since black nodes in a hyperspatial occupancy enumeration model are quasi-disjoint convex orthogonal polytopes, then $p = \bigcup_{\lambda} BlackNode_{\lambda}$, thus, by the expression $EVM_n(p \cup {}^*q) = EVM_n(p) \otimes EVM_n(q)$ if $p \cap {}^*q = \varnothing$ (**Corollary 5.9**), we have [Aguilera98]:

$$EVM_n(p) = EVM_n\left(\bigcup_{\lambda} BlackNode_{\lambda}\right) = \bigotimes_{\lambda} EVM_n(BlackNode_{\lambda})$$

Since all 2^n vertices of a box are Extreme Vertices, then all we have to do is list all 2^n vertices of every black node and collect (because of the XOR) every vertex that appears in an odd number of times in such a list, and discarding the remaining ones.

This provides a method for converting hyperspatial occupancy enumeration models to the nD-EVM. However, as stated before, a hyperspatial occupancy enumeration model should provide means for generating a list of all 2^n vertices for each black cell. The following sections will provide some clues which are related to the last comment.

6.6.3.1. Listing Vertices' Coordinates for the nD Hypercube

[Coxeter63] establishes that the coordinates for an nD hypercube with edges of length 2 can be described in general as:

$$(\underbrace{\pm 1,...,\pm 1}_{n})$$

For example, using the above description, the coordinates for a square (n = 2) are:

$$(+1,+1)$$
$$(+1,-1)$$
$$(-1,+1)$$
$$(-1,-1)$$

If we apply the translation (1,…,1), and the scaling $\left(\frac{1}{2},...,\frac{1}{2}\right)$ we obtain the general set of coordinates for a unit n-Dimensional hypercube:

$$(\underbrace{0,0,...0,0}_{n}),..., (\underbrace{1}_{1},\underbrace{0,...,0,0}_{n-1}),..., (\underbrace{1,...,1}_{i},\underbrace{0,...,0}_{n-i}),..., (\underbrace{1,1,...,1}_{n-1},\underbrace{0}_{1}), (\underbrace{1,1,...,1,1}_{n}) =$$

$$(1^0,0^n),\ (1^1,0^{n-1}),\ ...,\ (1^i,0^{n-i}),\ ...,\ (1^{n-1},0^1),\ (1^n,0^0)$$

where the coordinates must be permuted according the following distribution:

$$\binom{n}{0},\ \binom{n}{1},\ ...,\ \binom{n}{i},\ ...,\ \binom{n}{n-1},\ \binom{n}{n}$$

where $\binom{n}{i} = \frac{n!}{i!(n-i)!}$ defines the number of those coordinates that have *i* ones and *n-i* zeros. Then we can evaluate and relate the previous distribution with the number of vertices in the n-Dimensional hypercube [Pérez-Aguila03d]:

$$1 + n + ... + \frac{n!}{i!(n-i)!} + ... + n + 1 = \sum_{i=0}^{n}\binom{n}{i} = 2^n$$

Table 6.22 shows the application of the procedure on the 4D hypercube.

Value of i	Number of Combinations	Coordinates
0	1	(0,0,0,0)
1	$\binom{4}{1}=4$	(1,0,0,0) (0,1,0,0) (0,0,1,0) (0,0,0,1)
2	$\binom{4}{2}=6$	(1,1,0,0) (1,0,1,0) (0,1,1,0) (1,0,0,1) (0,1,01) (0,0,1,1)
3	$\binom{4}{3}=4$	(1,1,1,0) (1,1,0,1) (1,0,1,1) (0,1,1,1)
4	1	(1,1,1,1)

Table 6.22. Defining the 4D hypercube's vertices coordinates.

6.6.3.2. Listing Hypervoxels Vertices

The procedure commented in previous section can be extended in a straightforward way in order to list the 2^n vertices of an nD hypervoxel. Consider an n-dimensional grid where $C_{\underbrace{0,\ldots,0}_{n}}$ is the origin and the dimensions of each hypervoxel are given by x_1Side, ..., x_nSide. By applying to the general set of coordinates corresponding to a unit n-Dimensional hypercube the translation (x_1, …, x_n) and the scaling (x_1Side, ..., x_nSide) we obtain the set of coordinates for an n-dimensional hypervoxel $C_{x_1,\ldots,x_n}$. For example, in **Table 6.23** is presented the listing of the 16 vertices from a *rexel* (a 4D hypervoxel) C_{x_1,x_2,x_3,x_4}.

Vertex	X_1 coordinate	X_2 coordinate	X_3 coordinate	X_4 coordinate
0	$x_1 \cdot x_1Side$	$x_2 \cdot x_2Side$	$x_3 \cdot x_3Side$	$x_4 \cdot x_4Side$
1	$x_1 \cdot x_1Side$	$x_2 \cdot x_2Side$	$x_3 \cdot x_3Side$	$(x_4+1) \cdot x_4Side$
2	$x_1 \cdot x_1Side$	$x_2 \cdot x_2Side$	$(x_3+1) \cdot x_3Side$	$x_4 \cdot x_4Side$
3	$x_1 \cdot x_1Side$	$x_2 \cdot x_2Side$	$(x_3+1) \cdot x_3Side$	$(x_4+1) \cdot x_4Side$
4	$x_1 \cdot x_1Side$	$(x_2+1) \cdot x_2Side$	$x_3 \cdot x_3Side$	$x_4 \cdot x_4Side$
5	$x_1 \cdot x_1Side$	$(x_2+1) \cdot x_2Side$	$x_3 \cdot x_3Side$	$(x_4+1) \cdot x_4Side$
6	$x_1 \cdot x_1Side$	$(x_2+1) \cdot x_2Side$	$(x_3+1) \cdot x_3Side$	$x_4 \cdot x_4Side$
7	$x_1 \cdot x_1Side$	$(x_2+1) \cdot x_2Side$	$(x_3+1) \cdot x_3Side$	$(x_4+1) \cdot x_4Side$
8	$(x_1+1) \cdot x_1Side$	$x_2 \cdot x_2Side$	$x_3 \cdot x_3Side$	$x_4 \cdot x_4Side$
9	$(x_1+1) \cdot x_1Side$	$x_2 \cdot x_2Side$	$x_3 \cdot x_3Side$	$(x_4+1) \cdot x_4Side$
10	$(x_1+1) \cdot x_1Side$	$x_2 \cdot x_2Side$	$(x_3+1) \cdot x_3Side$	$x_4 \cdot x_4Side$
11	$(x_1+1) \cdot x_1Side$	$x_2 \cdot x_2Side$	$(x_3+1) \cdot x_3Side$	$(x_4+1) \cdot x_4Side$
12	$(x_1+1) \cdot x_1Side$	$(x_2+1) \cdot x_2Side$	$x_3 \cdot x_3Side$	$x_4 \cdot x_4Side$
13	$(x_1+1) \cdot x_1Side$	$(x_2+1) \cdot x_2Side$	$x_3 \cdot x_3Side$	$(x_4+1) \cdot x_4Side$
14	$(x_1+1) \cdot x_1Side$	$(x_2+1) \cdot x_2Side$	$(x_3+1) \cdot x_3Side$	$x_4 \cdot x_4Side$
15	$(x_1+1) \cdot x_1Side$	$(x_2+1) \cdot x_2Side$	$(x_3+1) \cdot x_3Side$	$(x_4+1) \cdot x_4Side$

Table 6.23. Listing a rexel's sixteen vertices.

6.6.3.3. Listing Black Nodes' Vertices for 2^n-trees

The **Algorithm 6.10** is an extension of a procedure originally described in [Aguilera97b]. Its objective is to list all 2^n vertices for each black node in a 2^n-tree. Q is a reference (pointer) to the tree; *width* is the length of the node under consideration whose minimum coordinates are given by the point p = (x_1, …, x_n).

```
Input: A pointer Q to a 2^n-tree.
       The width of the current node in the tree.
       The point p = (x1, …, xn) whose coordinates are the minimum in the current node.
       The number n of dimensions.
Procedure ListHyperocttreeVertices(Tree Q, real width, Point p, int n)
       if(NodeType(Q) = Black) then
              for k = 0 until 2^n - 1 do
                     offset_p = OffsetVertex(p, k, width);
                     Write(offset_p); // We list the k-th point in the current black node.
              end-of-for
       else
              If(NodeType(Q) = Gray) then
                     for k = 0 until 2^n - 1 do
                            offset_p = OffsetVertex(p, k, width/2);
                            ListOctreeVertices(son(Q, k), width/2, offset_p); // Recursive call
                     end-of-for
              end-of-if
       end-of-else
end-of-procedure
```

Algorithm 6.10. Listing all 2^n vertices for each black node in a 2^n-tree.

The **Algorithm 6.11** shows, as an example, the offset of a vertex in nD space towards the k-th direction, where $k \in \{0, \ldots, 2^n\text{-}1\}$, or equivalently $k \in \{\underbrace{0\ldots0_2}_{n},\ldots,\underbrace{1\ldots1_2}_{n}\}$ where each bit determines whether or not the vertex will be displaced by distance *dist* along each one of the coordinate axes.

```
Input: The point p=(x1, …, xn) to be 'offseted'.
       An integer k which indicates the direction along which p will be 'offseted'.
       The distance dist which defines the amount of translation to be applied to point p.
Output: The point p1 which corresponds to the offset of input point p=(x1, …, xn)
Procedure OffsetVertex(Point p, int k, real dist)
       p1 = p;
       for i = 1 until 2^n do
              if Odd(k) then
                     p1.x_i = p.x_i + dist
              end-of-if
              k = Int(k/2) // We update k by performing integer division of k by 2.
       end-of-for
       return p1
end-of-procedure
```

Algorithm 611. Computing the offset of a vertex.

6.7. Conclusions

In this chapter we have experienced the development and performance of some algorithms designed under the context of the Extreme Vertices Model in the n-Dimensional space. Summarizing, we have shown the efficiency of our algorithms under the following tasks:

- Regularized Boolean Operations (**Algorithm 6.4**).
- nD-OPP's measures (**Algorithms 6.5** and **6.6**).
- Extraction of boundary elements of nD-OPP's (**Algorithms 6.7**, **6.8** and **6.9**).

As mentioned above, the efficiency of such algorithms was evaluated from a statistical point of view. In such statistical analyses we have proposed approximation surfaces that fit as good as possible to the measures we obtained from the execution times of these algorithms. Such surfaces depend on two parameters: the number of input extreme vertices and the number of dimensions. The quality of the approximations is given by the coefficient of determination R^2 which reveals how closely the estimated values for the approximation surfaces correspond to our time measures. **Table 6.24** summarizes the execution times of the algorithms that were analyzed in this chapter.

Algorithm	Operation	Approximation Surface	x's exponent	R^2
6.4	Regularized Intersection	$t = 4{,}271.11\ x^{1.1737} n^{1.0862}$	1.1737	0.9234
6.4	Regularized Union	$t = 16{,}698.63\ x^{1.0821} n^{1.0607}$	1.0821	0.9221
6.4	Regularized Xor	$t = 483.17\ x^{1.4161} n^{1.4161} + 108{,}263{,}080$	1.4161	0.9260
6.5	Computing Content	$t = 4{,}763.939\ x^{1.1894} n^{0.8390}$	1.1894	0.9803
6.6	Computing Boundary Content	$t = \frac{12921.02\ x + 1454430\ n - 7196150}{0.0212336\ n^3 - 0.247936\ n^2 + 0.92776\ n - 1}$	1.0000	0.9963
6.7	Extracting Forward and Backward Differences	$t = 1{,}849.27\ x^{1.3} n^{1.64855}$	1.3000	0.9797
6.8	Building Differences Tree	$t = 1{,}160.7\ x^{1.3189} n^{2.3720}$	1.3189	0.9871

Table 6.24. Summarizing execution times of algorithms under the nD-EVM (x: Number of input extreme vertices, n: number of dimensions).

In all the equations associated to our approximation surfaces we have that by fixing the number of dimensions our functions become dependent only of one variable: the number of input extreme vertices. By this way we can then identify, as shown in **Table 6.24**, that the exponents associated to the number of vertices varies between 1 and 1.5. This experimentally identified complexity for our algorithms provides us elements to determine the temporal efficiency when we perform some operations between nD-OPP's represented through the nD-EVM.

Respect to conversion from and to other schemes for representing polytopes, we have presented algorithms to convert Hyperspatial Occupancy Enumeration Models to the nD-EVM which are generalizations of algorithms originally presented in [Aguilera98]. On the other hand, our **Algorithms 6.7**, **6.8** and **6.9** provide elements to have access to boundary elements in an nD-OPP represented through the nD-EVM. Moreover, some clues have been proposed in order to modify **Algorithm 6.8**, if the application requires, for obtaining, in combination with **Algorithm 6.9** the boundary tree of an nD-OPP as defined in **Section 2.2.3** leading to a conversion process for nD-EVM to an n-Dimensional Boundary Representation.