Universidad de Oviedo
Universidá d'Uviéu
University of Oviedo

Escuela de Ingeniería Informática
Universidad de Oviedo

School of Computing Engineering

Bachelor of Software Engineering

FINAL DEGREE PROJECT

Development of aggregation methods of partially ordered sets

Presented by:

César García Cabeza

Academic course 2019-2020

Development of aggregation methods of partially ordered sets

César García Cabeza

César García Cabeza *Development of aggregation methods of partially ordered sets*.

Final Degree Project. Academic course 2019-2020.

Academic tutor

Susana Irene Díaz Rodríguez
Computer Science and Artificial Intelligence

Elías Fernández Combarro
Computer Science and Artificial Intelligence

Bachelor of Software Engineering

School of Computing Engineering

University of Oviedo

Contents

List of Figures

List of Tables

1 Introduction

The arrangement of sets has always been an intensively researched topic within the scientific community because of its well-known practical application: assigning a global evaluation to a certain object according to varying properties. The following examples illustrate situations where this global classification would be genuinely useful:

- On an e-commerce website, displaying recommended products based on specific properties such as price, launch date, number of reviews, ratings, etc.
- Students organizing their tasks according to the deadline, level of difficulty, completion time, etc.

As it can be observed, the examples above show a series of *objects* (products, tasks) and each object possesses a series of *properties* (price, launch date, deadline, level of difficulty, etc).

We need to go into more detail in the second illustration in order to have a deeper understanding of our project. The student has six tasks to perform (tasks t_0 to t_5) and wants to organize them to know which tasks to work in first. To do so, he would take into account three properties: p_0, deadline, being of higher priority the tasks whose deadline is closer; p_1, level of difficulty, being of higher priority the tasks with the highest level of difficulty; p_2, real time, being of higher priority the tasks whose time of accomplishment is longer.

In order to establish the preference, the student establishes priority orders (1st, 2nd, 3rd, 4th, 4th, 5th, 6th) for each property, where 1st means the first to be done, while 6th means the last to be done. After analyzing the tasks, the student could reflect the analysis in the table 1.1.

Just by having a look at table 1.1 we may assume that t_0 task seems to be the first one being done (priorities 1st, 1st, 2nd). Besides, task t_4 seem to be the last one to be carried out (priorities 6th, 5th, 5th). However, when tackling the rest of the tasks it might not be so easy to determine the order in which they will be performed, at least at first sight. Apart from this, we must take into account that it may be the case that a global condition infringes some partial comparison of the properties under consideration, we will call this phenomenon "*cost*".

Table 1.1: Tasks ordered according to their properties

Task/Property	p0	p1	p2
t_0	1^a	1^a	2^a
t_1	2^a	4^a	3^a
t_2	3^a	3^a	4^a
t_3	5^a	6^a	1^a
t_4	6^a	5^a	5^a
t_5	4^a	2^a	6^a

In theory there are 6!, or 720, possible orderings of the different tasks. At this point we begin to observe the complexity of the problem, O(n!). For this reason, solving this problem has a great computational cost. The following are the possible combinations of objects depending on the number of objects:

Table 1.2: Number of arrangements according to the number of objects

Number of objects	Possible combinations
3	6
4	24
5	120
6	720
7	5040
8	40320
9	362880
10	3628800
11	39916800
12	479001600

We can observe how as the number of objects increases, the possible combinations increase in a factorial way, finally having almost five hundred thousand million possible arrangement for just 12 objects.

In fact, it can be shown that this problem is NP-hard. As a consequence, it is very unlikely that an efficient algorithm for it exists [Bac+15]. Therefore, an approximate algorithm must be found.

For this reason, the focus of this project will be on the arrangement of partially ordered sets, i.e. series of objects ordered among themselves but that do not necessarily need to be compared with each other. That is, we can have an object A, smaller than another object B, and another one C, smaller than B, but objects A and C are not compared to among themselves.

More specifically, the aim of this dissertation will be to apply different algorithms for the aggregation of these partially ordered sets, *posets*, by studying their costs and execution times in order to finally figure out the best one according to some conditions or costs.

2 Basic Concepts

In this section we will provide a more formal description of *posets*. We will work with the mathematical definitions and the properties on which we have based this project.

2.1 *Posets* and their representations

Definition 2.1.1 A partially ordered set or *poset* is a set P with a binary relation $\leq$, also (P, $\leq$), such that for all $x, y, z \in P$ the following properties are met:

- Reflexivity: $x \leq x$
- Antisymmetry: $x \leq y, y \leq x \implies x = y$
- Transitivity: $x \leq y, y \leq z \implies x \leq z$

[Dep15, p. 2]

Definition 2.1.2 A pair of elements x, y are comparable if either $x \leq y$ or $y \leq x$.

Definition 2.1.3 A pair of elements x, y are incomparable if neither $x \leq y$ nor $y \leq x$.

Property 2.1.4 A possible way of graphically representing a *poset* is by means of a Hasse diagram.

Definition 2.1.5 A Hasse diagram [CDM13, p. 66] is a graph where $a \leq b$ if and only if there is a sequence of connected lines upwards from a to b .

Example 2.1.4.1 Graphical representation of a *poset* using a Hasse diagram.

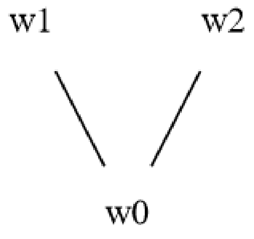


Figure 2.1: Hasse diagram of a *poset*

Example 2.1.2.1 In figure 2.1, object *wo* and *w1* are comparable and *wo* and *w2* are also comparable.

Example 2.1.3.1 In figure 2.1, objects *w1* and *w2* are incomparable.

Definition 2.1.6 Given a set P, a *linear extension* $\leq$ is a total order relationship, that is, an order such that all elements are comparable to each other.

Example 2.1.6.1 Representation of a *linear extension* using a Hasse diagram.

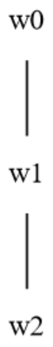


Figure 2.2: Hasse diagram of a *linear extension*

Property 2.1.7 Another possible way of representing a *poset* is by means of a matrix.

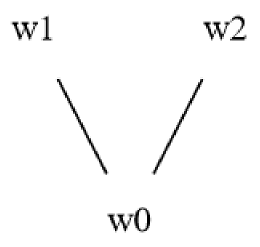
Definition 2.1.8 A boolean matrix M shows the relations of a *poset* (P, $\leq$) if it satisfies:

- $x \leq y \iff M[x,y] = 1$
- As a consequence, if x and y are incomparable $\implies M[x,y] = 0$

Note 2.1.8.1 A matrix M representing a *poset* (P,$\leq$) also meets its properties:

- Reflexivity: $\forall i \in P \; M[i,i] = 1$
- Antisymmetry: $\forall i,j \in P \; M[i,j] = 1 \implies M[j,i] = 0$
- Transitivity: $\forall i,j,k \in P \; M[i,j] = 1 \;\&\; M[j,k] = 1 \implies M[i,k] = 1$

Example 2.1.7.1 Representation of the *poset* in figure 2.3 by means of a matrix M.



(a) Hasse diagram

$$\equiv \begin{bmatrix} 1 & 1 & 1 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

(b) Matrix M

Figure 2.3: Equivalent representations of a *poset*

Remark 2.1.7.2 The typical representation of a *poset* is a matrix, as mentioned before. However, the data structure we will use throughout the investigation to represent *posets* is a list.
The equivalent i,j position of the matrix will be the position $i + j * n$ of the list, being n the total number of objects, as shown in figure 2.4.

2.2 Aggregation of *posets*

For our particular problem, an aggregation of partially ordered sets is a unique *linear extension* generated from many *posets* with the same elements.

$$\begin{bmatrix} M_{0,0} & M_{0,1} & M_{0,2} \\ M_{1,0} & M_{1,1} & M_{1,2} \\ M_{2,0} & M_{2,1} & M_{2,2} \end{bmatrix}$$

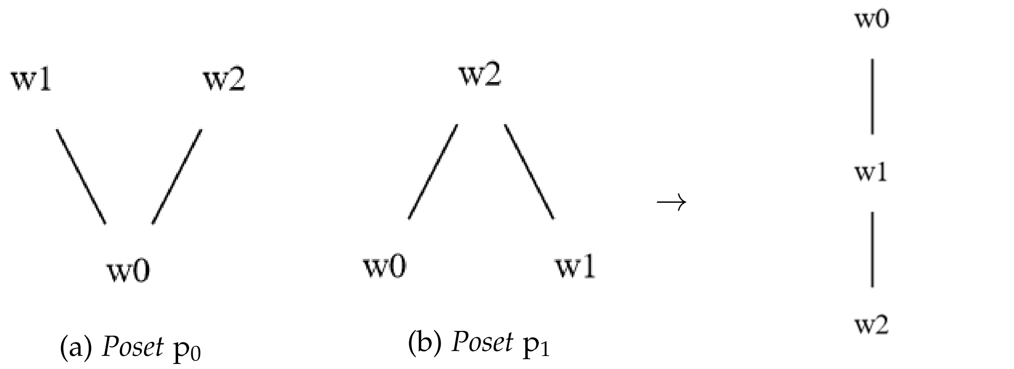
(a) Matrix

$$\begin{bmatrix} M_{0,0} & M_{1,0} & M_{2,0} & M_{0,1} & M_{1,1} & M_{2,1} & M_{0,2} & M_{1,2} & M_{2,2} \end{bmatrix}$$

(b) List

Figure 2.4: Equivalent data structures

Example 2.2.1 The example in figure 2.5 shows a possible aggregation of two *posets*.



(a) *Poset* p_0 (b) *Poset* p_1 (c) Linear extension

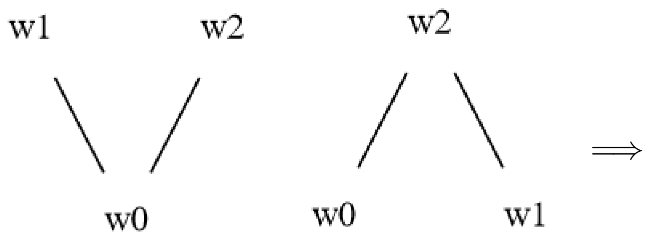
Figure 2.5: Possible aggregation of two *posets*

2.3 Aggregation Matrix

Definition 2.3.1 When aggregating *posets*, matrix A is the result of adding up all of them, so each position $A[i,j]$ will be the sum of each position $[i,j]$ of all the *posets*.

Property 2.3.2 Position $A[i,j]$ stores the total number of times that $i < j$.

Example 2.3.1.1 The computation of the matrix A can be observed in figure 2.6.



(a) *Poset* p_0 (b) *Poset* p_1

$$\begin{bmatrix} 2 & 1 & 2 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{bmatrix}$$

(c) Matrix A

Figure 2.6: Computation of matrix A

Pseudocode The pseudocode for obtaining the matrix A is shown in Algorithm 1.

Algorithm 1 SumPosets

```
Input: posets: list with all the posets
Input: n: number of objects
Output: A: A matrix
1: $A \leftarrow initMatrix(n, n)$
2: for poset in posets do
3:     for $i$; $i++$; $n$ do
4:         for $j \leftarrow i+1$; $j++$; $n$ do
5:             $A[i,j] \leftarrow A[i,j] + poset[i,j]$
6: return $A$
```

2.4 Cost of an aggregation

Once an aggregation is calculated, it may be the case that is not optimal, i.e. it has a cost. By not optimal it is understood that some partial comparison is violated.

Definition 2.4.1 The cost of an aggregation of *posets* is the number of partial restrictions violated by the linear extension.

Example 2.4.1.1 Let´s focus in the aggregation computed in figure 2.7. To compute the cost of the aggregation we need to check for each element if the upper elements were actually greater than it.

1. We need to check $w2$ and $w1$. To know whether $w1$ was lower than $w2$ we just simply have to check the value $A[1,2]$. So **cost=**1.

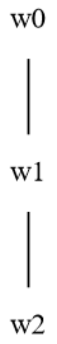
2. Next, we need to check $w2$ and $w0$. Therefore, **cost=**2.

3. Finally, we need to check $w1$ and $w0$. So **cost=**1.

So the total cost of the computed aggregation is **4**.

$$\begin{bmatrix} 2 & 1 & 2 \\ 0 & 2 & 1 \\ 0 & 0 & 2 \end{bmatrix}$$

(a) Matrix A



(b) Aggregation generated

Figure 2.7: Computation of the cost of an aggregation

Pseudocode To facilitate understanding of the algorithm, the pseudocode is shown in Algorithm 2.

Algorithm 2 Cost

```
Input: order: linear extension
Input: A: matrix containing partial orders added
Input: n: number of objects
Output: cost: cost of the linear extension computed
1: $cost \leftarrow 0$
2: for $i$; $i++$; $n$ do
3:     for $j \leftarrow i+1$; $j++$; $n$ do
4:         $elementX \leftarrow order[i]$
5:         $elementY \leftarrow order[j]$
6:         $cost \leftarrow cost + A[elementY, elementX]$
7: return $cost$
```

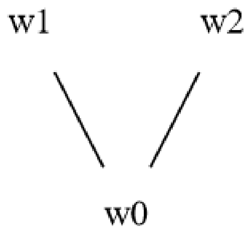
2.5 Random *poset* generation

Before aggregating *posets*, we have to take into account that we must be able to generate a large number of *posets* to ensure that they will be different enough for the results obtained in the project to be valid. Besides being generated in a random way, they must fulfill a series of properties apart from those mentioned in section 2.1.

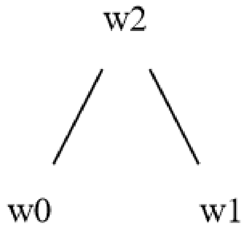
The quality of the results of our work will depend on the quality of the algorithm to randomly generate *posets*. That is why we will use an existing algorithm developed by Jaroslav Jezek and Vaclak Slavik [JS00, p. 129-133].

3 Methodologies and algorithms

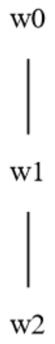
In this section, we will introduce all the algorithms used in this project. Moreover, in order to illustrate the algorithms mentioned below, a sample case will be used throughout their explanations. This example will be a basic one, with only 3 objects ($n = 3$) and 3 *posets* ($pp = 3$) and it is shown below.



(a) *Poset* p_0



(b) *Poset* p_1



(c) *Poset* p_2

$$\begin{bmatrix} 3 & 1 & 2 \\ 1 & 3 & 1 \\ 1 & 1 & 3 \end{bmatrix}$$

(d) Matrix A

Figure 3.1: Base case with $n = 3$ and $pp = 3$

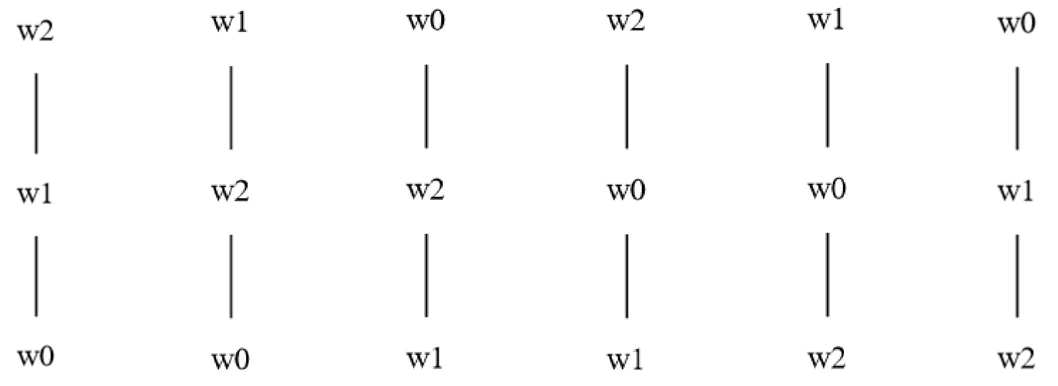
3.1 MinCost

The first and simple way to aggregate *posets* that comes to our mind is computing all the possible linear extensions and keeping the one with the lowest cost.

To do so, *MinCost* sequentially calculates the cost, with the algorithm mentioned in section 2.4, of all possible permutations. We need to bear in mind the problem of the permutations mentioned in section 1.2. The calculation of all possible permutations of n objects implies a very high computational cost translated in time due to its complexity: $O(n!)$.

The application of this algorithm in our base case (see figure 3.1) would consist of two parts:

1. Firstly, all the possible linear extensions would be calculated, as it is shown in figure 3.2.
2. Secondly, the costs of the linear extensions are computed. They are shown in table 3.1.
3. Finally, the aggregation with the best cost is chosen. In our case it would be s_0 or s_1 or s_3. Notice that the solution is not unique in general.



(a) s_0 (b) s_1 (c) s_2 (d) s_3 (e) s_4 (f) s_5

Figure 3.2: Possible solutions

Table 3.1: Possible solutions and their costs

Possible solution	Cost
s_0	3
s_1	3
s_2	4
s_3	3
s_4	4
s_5	4

So this algorithm guarantees to obtain an optimal solution, the aggregation with the lowest possible cost, in exchange of the considerable time that takes to calculate it when the number of object increases. This is the starting point of the development of the algorithms that we will see below, the search for an algorithm that obtains good costs or even the optimal one, but improving the complexity of the algorithm.

Pseudocode We can see the pseudocode in Algorithm 3.

Algorithm 3 MinCost

```
Input: $A$: matrix with the partial costs added
Input: $n$: number of objects
Output: $minPerm$: generated aggregations of the posets
1: $permInit \leftarrow initVector(n)$
2: $costInit \leftarrow \textbf{coste}(permInit, A, n)$
3: $minPerm \leftarrow permInit$
4: while $perm = next\_permutation(permInit)$ do
5:     $c \leftarrow \textbf{coste}(perm, A, n)$
6:     if $c < min$ then
7:         $min \leftarrow c$
8:         $minPerm \leftarrow perm$
9: return $minPerm$
```

3.2 Minimals

The initial algorithm studied in order to generate a linear extension is the algorithm *Minimals*, based on previous work by Combarro, Díaz and others [FHD19, p. 53].

The first concept we need to know in order to understand this algorithm is that of *minimal element*. An element $a \in P$ is a *minimal element* if there is no $b \in P$ such that $a > b$ [FHD19, p. 51].

To identify *minimal* elements, in the first part of the algorithm we will calculate for each element, the number of elements it has above and below. We must remember the meaning of each position *i,j* in the matrix A: the position $A[i, j]$ means the number of times that element i is lower than element j.

Vector up Taking this into account, in order to calculate the number of elements that each element i has above itself, we will need to go through the row i of matrix A, being the sum of the cells of row i the number of elements that element i has above itself. The vector *up* could be calculated in the following way:

$$up[i] = \sum_{j}^{n} A[i,j]$$

Vector down Similarly, to calculate the number of elements that each element has below itself in this case we must go through the cells of the same column. The vector *down* would be calculated in the following way:

$$down[i] = \sum_{j}^{n} A[j,i]$$

Bound constant Besides, we calculate a constant called *bound*, which we then use to initialize the value of the variable *min*.

$$bound = \sum_{i}^{n} up[i]$$

Used vector In addition, we initialize a *used* boolean vector to note which element has been used.

We can observe in table 3.2 the values of the previous variables for our base case.

Table 3.2: Values of the variables after the *Minimals* initialization

	w0	w1	w2
Up	6	5	5
Down	5	5	6
Used	False	False	False

Bound = 16

Search of the minimals Once this four initial parameters have been calculated, we proceed to the search of the *minimals*. To do so, we execute the following sequence of actions:

1. We compute the lowest number of elements below as long as the element in that position has not already been used.
   In our base case, the lowest number of elements below is 5, so **min=5**.

2. We select the possible *minimals*.

   $$down[i] == min \;\; \& \; !used[i] \implies i \textit{ is a minimal}$$

   According to the previous condition, both **w0** and **w1** are possible *minimals*.

3. We choose one element from the possible *minimals*, having more probability the one that has more elements above it. The probability of a possible *minimal* m_i of being chosen is:

   $$P(i) = \frac{up[i]}{\sum_{j \ minimal} up[j]}$$

   In our example, the probabilities of $w0$ and $w1$ of being chosen are $\frac{6}{11}$ and $\frac{5}{11}$ respectively. Let´s suppose that **w1** is chosen.

4. We mark the chosen element as used and we update the values of the up and $down$ vectors. To update the vector $down$ and up we will subtract from each i position of the vector the i position of the chosen *minimal* element row or column respectively.
   In our example, the values of the variables after the update are shown in table 3.3.

   Table 3.3: Values of the variables after *iteration* 0 of *Minimals*

	w0	w1	w2
Up	5	2	4
Down	4	2	5
Used	False	True	False

   Bound = 16

5. We finally add the chosen element to the vector that represents the aggregation.
   In our example, **order = [w1]**.

These 5 steps must be done n times and the aggregation would be computed.

Pseudocode The pseudocode of *Minimals* is that of Algorithm 4.

3.3 Minimals Random

Minimals Random, a variation done by Combarro, Díaz and others, is almost equal to the *Minimals* algorithm, differing from the first in a small way.

As it name indicates, the difference in the name is in the word *random*. In this algorithm, when choosing the *minimal* we do not take into account their relative position in the *poset* and we do not give more importance to the elements that have more elements above them. In this way, all the possible *minimals* have the same probability of being chosen, they are chosen *randomly*.

Because of this, when implementing the algorithm some variables are not needed. We need to remember that in *Minimals* algorithm, one of the necessary parameters that we calculated at the beginning was the vector *up*. This vector allowed us to know how many elements each element had above it, and it was the one we used to know the probability of each possible *minimal* of being chosen. However, we do not need this vector anymore since we do not assign different probabilities to each candidate.

As it has been explained, *Minimals Random* differs from the previous one in the way the *minimals* are chosen. So, the only difference would be in step 3: now the probability of a *minimal* candidate i of being chosen is

$$P(i) = \frac{1}{k}$$

being k the total number of possible *minimals*.
Therefore, in the *iteration* 0 of *Minimals Random* of our base case, both wo and $w1$ would have the same probability of being chosen, that is $\frac{1}{2}$.

Pseudocode The pseudocode of *Minimals Random* can be observed in Algorithm 5.

3.4 MinCost Multi-Thread

This algorithm is based on the original *MinCost* algorithm, described in section 3.1, with a small but very important modification.

Algorithm 4 Minimals

```
Input: $A$: matrix with all the partial costs added
Input: $n$: number of objects
Output: $order$: generated aggregation for the partially ordered sets
 1: $order \leftarrow initVector(n)$
 2: $used \leftarrow initVector(n)$
 3: $up \leftarrow initVector(n)$
 4: $down \leftarrow initVector(n)$
 5: $bound \leftarrow 0$
 6:
 7: for $i$; $i++$; $n$ do                          ▷ Here, we initialize the variables
 8:     for $j$; $j++$; $n$ do
 9:         $up[i] \leftarrow up[i] + A[i,j]$
10:         $down[i] \leftarrow down[i] + A[j,i]$
11:         $bound \leftarrow bound + A[i,j]$
12:
13: for $i$; $i++$; $n$ do               ▷ Here, we start the search of the *minimals*
14:     $min \leftarrow bound$
15:     for $j$; $j++$; $n$ do
16:         if $!used[j]$ & $down[j] < min$ then
17:             $min \leftarrow down[j]$
18:
19:     $posibilities \leftarrow initVector(n)$
20:     for $j$; $j++$; $n$ do
21:         if $!used[j]$ & $down[j] == min$ then
22:             for $k$; $k++$; $up[j]$ do
23:                 $add(posibilities, j)$
24:
25:     $chosen \leftarrow random(posibilities)$
26:     $used[chosen] \leftarrow true$
27:     for $j$; $j++$; $n$ do
28:         $up[j] \leftarrow up[j] - A[j, chosen]$
29:         $down[j] \leftarrow down[j] - A[chosen, j]$
30:
31:     $order.add(chosen)$
32:
33: return $order$
```

Algorithm 5 Minimals Random

```
Input: $A$: matrix with all the partial costs added
Input: $n$: number of objects
Output: $order$: generated aggregation of the $posets$
 1: $order \leftarrow initVector(n)$
 2: $used \leftarrow initVector(n)$
 3: $down \leftarrow initVector(n)$
 4: $bound \leftarrow 0$
 5: for $i;\ i++;\ n$ do                         ▷ Here, we initialize the variables
 6:     for $j;\ j++;\ n$ do
 7:         $down[i] \leftarrow down[i] + A[j,i]$
 8:         $bound \leftarrow bound + A[i,j]$
 9: for $i;\ i++;\ n$ do                  ▷ Here, we start the search of the minimals
10:     $min \leftarrow bound$
11:     for $j;\ j++;\ n$ do
12:         if $!used[j]\ \&\ down[j] < min$ then
13:             $min \leftarrow down[j]$
14:     $posibilities \leftarrow initVector(n)$
15:     for $j;\ j++;\ n$ do
16:         if $!used[j]\ \&\ down[j] == min$ then
17:             $add(posibilities, j)$
18:     $chosen \leftarrow random(posibilities)$
19:     $used[chosen] \leftarrow true$
20:     for $j;\ j++;\ n$ do
21:         $down[j] \leftarrow down[j] - A[chosen, j]$
22: return $order$
```

The *MinCost* algorithm is the first one that always manages to obtain the aggregation with the optimal cost. In exchange for this optimal solution, we will obtain an O(n!) complexity, the worst of all.

If we look at the *MinCost* algorithm again, we can see that what produces O(n!) complexity, and therefore where the algorithm takes the longest, is the calculation of all the possible permutations of *n* objects.

This calculation is carried out sequentially to check one by one which permutation is the one that has the lowest cost, i.e. what permutation is the optimal solution. However, we could do the same thing parallelly. We may calculate the permutations in different threads and, at the same time, calculate if the cost of the permutations is the lowest one.

Following this idea, we looked for different solutions to calculate the permutations in a parallel way and finally we selected the algorithms *PermutationMixOulletSaniSinghHuttunen* and *ExecuteForEachPermutationMT* [OS18]. These algorithms calculate the number of possible threads according to the processor and assign a number of permutations to each thread, thus being executed in a parallel way. They also receive as a parameter an action to be carried out in each permutation, which in our case will be the same as in the *MinCost* algorithm: to calculate the cost and if it is lower that the current one, update the current cost value.

Therefore, the application of this algorithm to our base case would be exactly the same as the one of *MinCost* but done in parallel.

Pseudocode The pseudocode of this algorithm is shown in the next page.

3.5 Sorting Algorithms

A *posets aggregation* is still a total sort of *n* objects. That is why a possible solution to our problem is to use sorting algorithms, after all what they do is to sort objects based on *x* criteria.

Algorithm 6 MinCost MT

```
Input: $A$: matrix with the partial costs added
Input: $n$: number of objects
Output: $minPerm$: generated aggregation of the posets
 1: $permInit \leftarrow initVector(n)$
 2: $costInit \leftarrow$ coste$(permInit, A, n)$
 3: $minPerm \leftarrow permInit$
 4: for all $PermutationMixOulletSaniSinghHuttunen(permInit)$ do in parallel
 5:     $c \leftarrow$ coste$(perm, A, n)$
 6:     if $c < min$ then
 7:         $min \leftarrow c$
 8:         $minPerm \leftarrow perm$
 9:
10: return $minPerm$
```

We are going to try *Bubble*, *Insertion*, *Selection*, *Quicksort* and *MergeSort* algorithms. The implementations of these five sorting algorithms are not going to be explained in this dissertation because they are already well-known.

The difficulty is, therefore, defining when one object is lower than another, that is, it lies in defining a method of comparison for the objects. In section 2.3, it was explained what the A matrix was and what it meant. Each position i, j of the matrix represented the number of times object i was lower than object j, therefore, position i, j represents the number of times object i is lower than object j. At this point, we need to remember a very important property of the *posets*, and that is that two objects i, j may no be compared to each other. This last property is the one that makes us define two possible ways of comparing objects.

When it comes to deal with experiments, we will execute the two comparisons, presented in the following explanations, with the different sorting algorithms mentioned before.

Comparison A In this comparison we will only take into account the times that an i element is lower or larger than a j element.

To calculate the number of times that $i < j$, we will calculate it directly from the i, j position of the matrix A.
Similarly, to calculate the number of times that $i > j$, we will obtain it directly from the position i, j of the matrix A.

As it can be observed, we cannot state with certainty that an object i is lower than j. To decide this comparison we will refer to a probability

$$P(i < j) = \begin{cases} \frac{lower}{lower+greater} & if \quad lower + greater \neq 0 \\ 0.5 & else \end{cases}$$

being *lower* the amount of times that i is lower than j, and *greater* the amount of times that i is greater than j.

Let´s compute these probabilities for our base case. We just need to look at the values of the matrix A and compute the probabilities one by one. We can observe

Table 3.4: *ComparisonA*: How to compute $P(i < j)$

	w0	**w1**	**w2**
w0	1	$\frac{A[0,1]}{A[0,1]+A[1,0]}$	$\frac{A[0,2]}{A[0,2]+A[2,0]}$
w1	$\frac{A[1,0]}{A[1,0]+A[0,1]}$	1	$\frac{A[1,2]}{A[1,2]+A[2,1]}$
w2	$\frac{A[2,0]}{A[2,0]+A[0,2]}$	$\frac{A[2,1]}{A[2,1]+A[1,2]}$	1

in table 3.4 the values of the matrix A that we need to compute each probability. Finally, in table 3.5 we can see the actual values of these probabilities for our base case.

Table 3.5: *ComparisonA*: Actual values of $P(i < j)$

	w0	**w1**	**w2**
w0	1	$\frac{1}{2}$	$\frac{2}{3}$
w1	$\frac{1}{2}$	1	$\frac{1}{2}$
w2	$\frac{1}{3}$	$\frac{1}{2}$	1

Comparison B In this case we will take into account both the times that an object i is smaller or larger than another object j, and the times that the two objects are not compared to each other.

The times that i is lower or larger than j would be calculated in the same way as in the previous comparison method.

In order to calculate the number of times these objects are not compared, we first need to know the total number of previous comparisons, i.e. how many *posets* form the set. To do this, we must remember that the *posets* are *reflexive*, so in the elements of the diagonal of A we will have this value. In our base case, **total = 3**.

Once we know this value, knowing how many times two objects are not comparable is as simple as computing

$$notCompared = total - (lower + greater)$$

We can see in table 3.6 the number of times each element is not comparable to other for our base case.

Table 3.6: *ComparisonB*: Number of times that objects are not comparable to each other

	w0	**w1**	**w2**
w0	0	1	0
w1	1	0	1
w2	0	1	0

In this case, the probability that determines if an object is lower than another one is:

$$P(i < j) = \frac{lower + 0.5 \times notCompared}{total}$$

by weighting the number of times they are not compared to each other by 0.5.

How to compute these probabilities is shown in table 3.7 and we can observe there that now we are taking into account the number of times two objects are not compared. Finallly, we can see in table 3.8 the actual values of these probabilities.

Table 3.7: *ComparisonB*: How to compute $P(i < j)$

	w0	**w1**	**w2**
w0	1	$\frac{A[0,1]+0.5\times1}{3}$	$\frac{A[0,2]+0.5\times0}{3}$
w1	$\frac{A[1,0]+0.5\times0}{3}$	1	$\frac{A[1,2]+0.5\times1}{3}$
w2	$\frac{A[2,0]+0.5\times0}{3}$	$\frac{A[2,1]+0.5\times1}{3}$	1

Table 3.8: *ComparisonB*: Actual values of $P(i < j)$

	w0	**w1**	**w2**
w0	1	$\frac{1}{2}$	$\frac{2}{3}$
w1	$\frac{1}{2}$	1	$\frac{1}{2}$
w2	$\frac{1}{3}$	$\frac{1}{2}$	1

For our base case, we can observe in tables 3.5 and 3.8 that both probabilities of *ComparisonA* and *ComparisonB* are the same. However, we can perfectly notice in tables 3.4 and 3.7 the difference in the way of computing those probabilities. In a more general case, the probabilities will be different.

3.6 Simulated Annealing

Simulated annealing is an algorithm based on the analogy between the *annealing* of solids and the problem of solving combinatory optimization problems. We need to clarify that this algorithm does not calculate a base *linear extension*, instead, an initial *linear extension* is passed to its input and, as it is being iterated, the aggregation is finally optimized.

Annealing is the process of heating a solid body and then cooling it very slowly until it crystallizes. Atoms have more energy at higher temperatures. As the temperature decreases, the energy of the atoms is gradually reduced [PK00, p. 11].

The probability of distribution of the body´s energy, *P(E)* for a given temperature *T* is determined by the Boltzmann probability:

$$P(E) = e^{-E/(kT)}$$

where *E* is the energy of the system and *k* is Boltzmann´s constant [Dré+06, p.25].

But, how can this physical process be applied to an optimization problem? To do so, we must establish an analogy between the main elements of *annealing* and an optimization problem:

- In our algorithm, possible solutions would resemble solid body states.
- The values of the costs would be the energies of the body states.
- Our optimal solution would be the state with the least energy.

- And the new solution would be accepted in accordance to this probability

$$P(accepted) = \begin{cases} 1 & if \quad newCost \leq currentCost \\ e^{(currentCost - newCost)/T} & if \quad newCost > currentCost \end{cases}$$

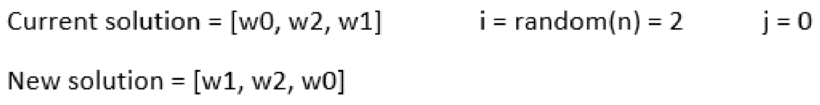
  In this way, better solutions are always accepted and worse solutions are accepted depending on this probability.

The algorithm would then receive an initial ordering and would be an iterative process of the following actions:

1. **Calculate the new solution**
2. **Calculate the cost of the solution**
3. **Decide whether to accept and update the current solution**
4. **Update the best solution**
5. **Lower the temperature**

Calculate the new solution In order to calculate a new solution, we will randomly calculate a number i between $0 \ and \ n-1$, where n is the number of objects in the aggregation. That number i represents the position of the aggregation that will be exchanged with its position on the right j. In case of changing the last position, it will be changed with the first one.

Let´s imagine that for our base case an **initial solution** $[w0, w2, w1]$ is passed as input and its cost = 3. Then, a random index is computed and we suppose that this number is 2, so we would change positions 2 and 0. So the **new solution** would be $[w1, w2, w0]$. This process is shown in figure 3.4.


Current solution = [w0, w2, w1]   i = random(n) = 2   j = 0

New solution = [w1, w2, w0]

Figure 3.4: *Simulated Annealing*: computing new solution

Calculate the cost of the solution To calculate the cost of the solution we use the algorithm described in section 2.4. We can check in table 3.1 the value of the cost of our new solution, so **newCost = 4**.

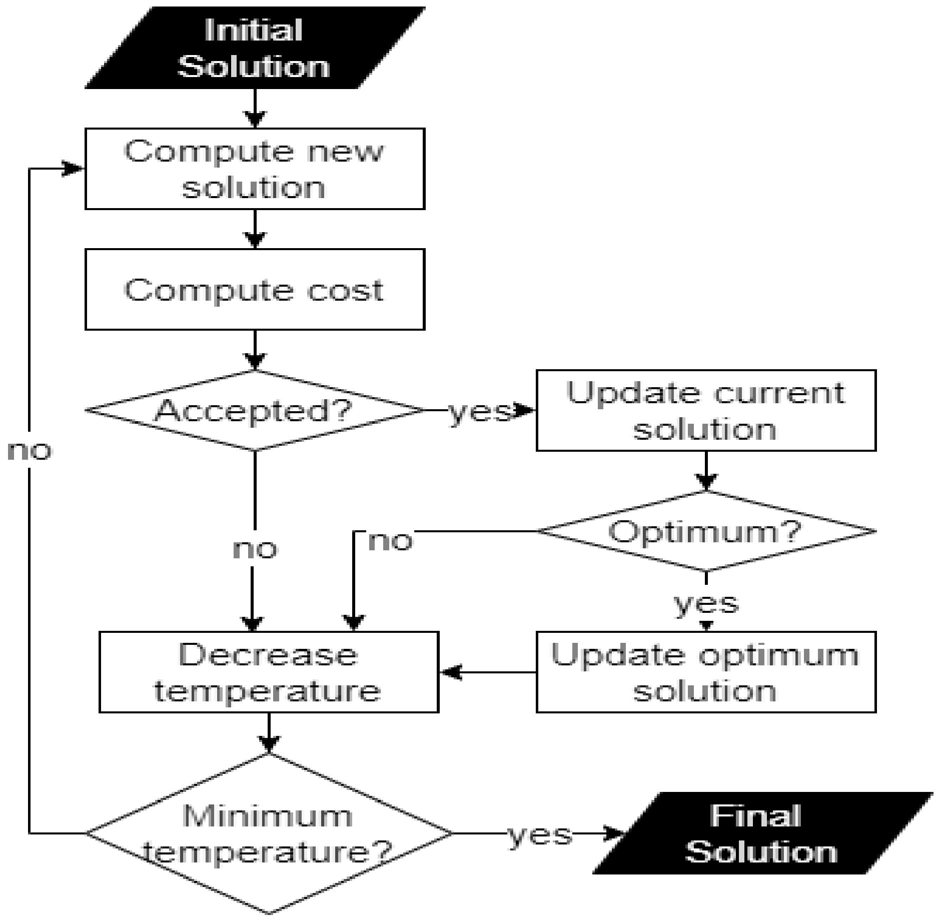


Figure 3.3: Flowchart of the Simulated Annealing algorithm

Decide whether to accept and update the current solution At this point, we know the cost of the new solution (newCost = 4) and the cost of the current solution (cost = 3).

- If the new cost is lower than the previous one, we update the value of the current cost and the aggregation calculated becomes the current solution.
- If the new cost is higher or equal to the previous one, the update is more complex. At this point the Boltzmann probability formula already mentioned comes into play, since the new solution will be accepted based on that probability. At high temperatures, it is easier to accept new solutions with a worse cost; but as the temperature drops, that probability becomes lower and it is more difficult to accept a worse solution.

Therefore, we must take into account that using this algorithm there is a strong likelihood that, at some point of the iteration, a worse cost aggregation will be chosen, being this fact one of the advantages of this algorithm.

We would use the Boltzmann´s probability formula since the new cost is worse than the current cost. The probability of being accepted would be

$$P(accepted) = e^{\frac{currentCost-newCost}{T}} = e^{\frac{3-4}{T}} = e^{\frac{-1}{T}}$$

As we can see, the probability of being accepted depends only on the temperature, the greater the temperature is the greater the probability is for the same cost variation.

Update the best solution Once the cost and the current solution have been updated, we have to keep the best solution iteration by iteration, so that once the iterations are over, be able to return it. In the same way we did before, we check if the cost of the current solution is better than the best solution we have saved and if it is better we update both the cost and the best solution value.

Cooling system One of the important parts of this algorithm is the initial temperature, its cooling system and the limit temperature to stop iterating. The higher the initial temperature, the more iterations will be performed and, therefore, the higher probability of obtaining the optimal solution. The same applies to the cooling system. A very common option and the one we will use in this dissertation is a geometric cooling rule:

$$T_{i+1} = \beta\, T_i$$

where β is a constant temperature factor less than 1 but very close to 1.

Pseudocode The pseudocode of *Simulated Annealing* is shown in Algorithm 7, that together with the flowchart of figure 3.3 facilitates its comprehension.

Algorithm 7 Simulated Annealing

```
Input: initialSolution: aggregation passed as the initial solution
Input: A: matrix A
Input: n: number of objects
Input: T: initial temperature
Input: β: cooling constant
Output: bestSolution: aggregation computed as the final solution
Output: bestCost: best cost computed
 1: $currentSolution \leftarrow initialSolution$
 2: $currentCost \leftarrow coste(currentSolution)$
 3: $newSolution \leftarrow currentSolution$
 4: $bestSolution \leftarrow initialSolution$
 5: $bestCost \leftarrow currentCost$
 6: while $T > 1$ do
 7:     $newCost \leftarrow coste(newSolution)$
 8:     if booltzman($currentCost, newCost, T$) then
 9:         $currentSolution \leftarrow newSolution$
10:         $currentCost \leftarrow newCost$
11:     if $currentCost < bestCost$ then
12:         $bestSolution \leftarrow currentSolution$
13:         $bestCost \leftarrow currentCost$
14:     $T \leftarrow \beta \times T$
15:     $positionA \leftarrow random(n)$
16:     $swap(newSolution, positionA, (positionA + 1)\%n)$
17: return $bestSolution, bestCost$
```

3.6.1 Initializations

As it has already been explained, an initial solution is needed to be passed as input to the *simulated annealing* algorithm. Therefore, the following algorithms will be tested as initializations in section 4: *Minimals*, *Minimals Random*, *Random*, *Bubble*, *Selection*, *Insertion*, *QuickSort*, *MergeSort*.

3.7 Linear Programming

Linear Programming is a mathematical optimization technique used to maximize or minimize a linear function. It is widely used in many fields such as Economics or Business.

3.7.1 General form of Linear Programming

Every linear programming problem consists of four concepts [Vid20] that must be known to understand this algorithm:

1. **Decision variables**: are the variables that will decide the output. In a general context, they are represented by the vector $\mathbf{x}$.
2. **Domain**: it is the set of values that decisions variables can take. They must always take non-negative values. So this last limitation can be expressed as $\mathbf{x} \geq 0$, being $\mathbf{x}$ the vector of variables.
3. **Constraints**: are the restrictions or limitations on the decision variables. They limit the possible values of the decision variables. They are represented as $\mathbf{Ax} \leq \mathbf{b}$, where $\mathbf{A}$ stands for the matrix of coefficients, $\mathbf{x}$ is the vector of variables and $\mathbf{b}$ is a vector of coefficients.
4. **Objective function**: is the linear function to be optimised, either to maximise or minimise. It can be represented as $\mathbf{c}^\mathbf{T}\mathbf{x}$, being $\mathbf{x}$ the vector of variables, $\mathbf{c}$ a vector of coefficients and $^\mathbf{T}$ stands for transpose.

Therefore, the standard form [Cam18, p. 8-12] of writing a linear programming problem is

$$max\{\mathbf{c}^\mathbf{T} | \mathbf{Ax} \leq \mathbf{b} \wedge \mathbf{x} \geq 0\} \qquad or \qquad min\{\mathbf{c}^\mathbf{T} | \mathbf{Ax} \leq \mathbf{b} \wedge \mathbf{x} \geq 0\}$$

Moreover, other definitions [cited in Ste04, p. 3] should be known:

1. A **solution** to a linear program is a setting of the variables.
2. A **feasible solution** to a linear program is a solution that satisfies all constraints.
3. The **feasible region** in a linear program is the set of all possible feasible solutions.
4. An **optimal solution** to a linear program is the feasible solution with the greatest or lowest objective function value.

3.7.2 Linear Programming application

The adaptation of Linear Programming to our particular problem would be the following:

Variables Suppose that in our problem we have a number of n objects. This would result in a V matrix of $n \times n$ size, so we would have $n \times n$ variables, each of them representing a position of the V matrix.

$$V = \{V_{0,0}, V_{0,1}, ..., V_{n-1,n-1}\}$$

Domain These variables are binary, taking only the values 0 or 1. More precisely, we are working with zero-one linear programming since our variables are binary.

$$D = \{0,1\}$$

Constraints The restrictions in our case are imposed by the properties of the *posets* mentioned in section 2.1, on page 6.

- **Diagonal**: The first type of constraint is a result of every element being less than or equal itself. Therefore, the elements in the diagonal of the matrix will have to be equal to 1.
$$V[i,i] = 1\, \forall i$$
- **No cycles**: The second type of constraint is a result of the fact that in a linear extension if an x element is smaller than another y element, the y element is larger than the x element.
$$V[i,j] = 1 \implies V[j,i] = 0\, \forall i \neq j$$
- **Transitivity**: The last constraint results from the property that if an element x is smaller that an element y, and y is smaller than another element z, x is smaller than z.
$$V[i,j] = 1 \;\&\; V[j,k] = 1 \implies V[i,k] = 1\, \forall i \neq j \neq k$$

In the table 3.9, we can see the detailed constraints for our base case.

Table 3.9: *Linear Programming*: Constraints for our base case

Constraint	**Type**
V[0, 0] = 1	Diagonal
V[1, 1] = 1	Diagonal
V[2, 2] = 1	Diagonal
V[0, 1] + V[1, 0] = 1	No cycles
V[0, 2] + V[2, 0] = 1	No cycles
V[1, 2] + V[2, 1] = 1	No cycles
V[0, 1] + V[1, 2] - V[0, 2] $\leq$ 1	Transitivity
V[0, 2] + V[2, 1] - V[0, 1] $\leq$ 1	Transitivity
V[1, 0] + V[0, 2] - V[1, 2] $\leq$ 1	Transitivity
V[1, 2] + V[2, 0] - V[1, 0] $\leq$ 1	Transitivity
V[2, 0] + V[0, 1] - V[2, 1] $\leq$ 1	Transitivity
V[2, 1] + V[1, 0] - V[2, 0] $\leq$ 1	Transitivity

Objective Function Our function to optimize will be the cost of the aggregation of the *posets*, with the aim of having this function with the minimum value. This function will be the sum of the multiplication of each variable by its respective partial cost.
We need to remember that in the A matrix each position i, j represents the number of time that object i is smaller than object j. Therefore, we can obtain the cost by multiplying the value of each variable $V_{i,j}$ by the value of its transposed position of matrix A.

$$objectiveFunction(V, A) = \sum_{i,j}^{n} V[i, j] \times A[j, i] \qquad \forall i \neq j$$

Therefore, the objective function would look like this for our base case:

$$\begin{aligned} objectiveFunction(V, A) &= V[0,1] \times A[1,0] + V[0,2] \times A[2,0] + ... + \\ &\quad V[2,1] \times A[1,2] \\ &= V[0,1] \times 1 + V[0,2] \times 1 + ... + V[2,1] \times 1 \end{aligned}$$

Libray used For the implementation of this algorithm we consider two libraries: *Microsoft Solver Foundation* [Kri17] and *lpsolve* [EN18].

Finally, we have opted for the second one, since it is a more powerful library and the model created has no size limitations as happens in the first library. However, a disadvantage is that it is more difficult to implement and to understand the code once it is written.

Pseudocode The pseudocode of this algorithm is shown below.

Algorithm 8 LinearProgramming

```
Input: $A$: matrix with all the partial costs added
Input: $n$: number of objects
Output: $solution$: linear extension generated
Output: $cost$: cost of the aggregation
 1: $model \leftarrow createModel()$
 2: for $i;\ i++;\ n$ do
 3:     for $j;\ j++;\ n$ do
 4:         $addVariable(model, V[i,j])$
 5:         $setVariableDomain(V[i,j], "binary")$
 6: for $i;\ i++;\ n$ do
 7:     $addDiagonalConstraint(model, V[i,i])$
 8: for $i;\ i++;\ n$ do
 9:     for $j;\ j++;\ n$ do
10:         if $i \neq j$ then
11:             $addAntiSymmetryConstraint(model, V[i,j], V[j,i])$
12: for $i;\ i++;\ n$ do
13:     for $j;\ j++;\ n$ do
14:         if $i \neq j$ then
15:             for $k;\ k++;\ n$ do
16:                 if $i \neq j \neq k$ then
17:                     $addTransitivityConstraint(model, V[i,j], V[j,k], V[i,k])$
18: for $i;\ i++;\ n$ do
19:     for $j;\ j++;\ n$ do
20:         if $i \neq j$ then
21:             $addTermObjectiveFunction(model, V[i,j], A[j,i])$
22: $setObjectiveFunction(model, "minimize")$
23: $solution \leftarrow solve(mode)$
24: return $solution, cost(solution)$
```

4 Experiments

This is the part where we will use the algorithm mentioned in section 2.5 to randomly generate *posets* in order to apply all the algorithms mentioned in section 3.

The structure that we will follow in this section will be that of analyzing the algorithms by families to make a collective analysis of the best algorithm of each family. Finally, all the best family algorithms will be analyzed altogether.

We will execute each algorithm, except two of them, 50 times for each number of objects n such that $3 \leq n \leq 12$, and for each number of objects we will try with sets formed by *pp posets* such that $2 \leq pp \leq 50$. Therefore, the cost and time values will be the mean of the 50 executions.

The algorithms mentioned in section 3 will be grouped in the following families:

- MinCost Single-Thread vs MinCost Multi-Thread
- Minimals vs MinimalsRandom
- Sorting algorithm
- Simulated Annealing
- Linear Programming

The different methods will be evaluated in terms of the costs and execution times of the algorithms. For the costs, we will represent the percentage error with respect to the minimum cost, that is

$$((cost - minCost)/minCost) \times 100$$

One example for our base case would be computing the cost error of $s2$ (cost=4) when the optimal cost is 3. The cost error of $s2$ would be

$$\begin{aligned} costError_{s2} &= \frac{4-3}{3} \times 100 \\ &= 33.33\% \end{aligned}$$

From now onwards, when we refer to lower costs, we will understand that it is so because the percentage errors are lower.

The values of the costs will represent the average cost for each *n*, that is, it will be the average cost of all *pp* for the same *n*. As for the times, the values will represent the total time in *milliseconds* for each number of objects *n*, that is, the sum of the times of all *pp* for the same *n*.

As indicated above, some algorithms will not be executed 50 times. These algorithms are *MinCost ST* and *MinCost MT* due to their very long execution times and *Linear Programming* since it always computes the best solution.

4.1 Technology used

The implementation of *Minimals*, *Minimals Random* and *MinCost* was originally done in $C++$. However, I tried to improve the performance of the last one since the beginning of this project.

That made me change the programming language of the implementations to *C#* because of the facilities it has to parallelize code.

4.2 Hardware

All the experiments have been run in a computer with an Intel Core i5 6600K overclocked at 4.2GHz, excluding the one in section 4.3, that has been run in an Azure virtual machine with an Intel Xeon Platinium 8168.

4.3 MinCost ST vs MinCost MT

This family of algorithms is made up of the two algorithms that we know ensure the optimal aggregation, that is the one with the lowest cost. So in this section we are interested in analyzing the difference in time, in order to determine with algorithm is better.

In this particular case, the table of costs does not reflect the errors, since it is known that they always obtain the best solution, but represents the real costs obtained in the executions.

You can see how the costs (table 4.1) of both algorithms are the same and increase as the number of object *n* increases. In addition, you can clearly see in graph 7.1 this positive trend.

In table 4.2, the execution times of these two algorithms can be seen. The *single-thread* version takes a total of 469484772,59850 milliseconds (a little more than 130 hours) for the 6046229,96280 milliseconds (a little more than one hour and a half) of the *multi-thread* version.

Considering the execution times, it is very clear that the best algorithm of this family is ***MinCost MT***.

Table 4.1: Costs: *MinCost ST* vs *MinCost MT*

N	MinCost ST	MinCost MT
3	7.143	7.143
4	14.286	14.286
5	23.061	23.061
6	32.898	32.898
7	46.796	46.796
8	61.551	61.551
9	79.102	79.102
10	87.592	87.592
11	107.000	107.000
12	133.245	133.245
Average cost	59.267	59.267

4.4 Minimals vs MinimalsRandom

As we can see in table 7.1 and graph 4.1, the cost errors of the *Minimals* algorithm are lower than those of *MinimalsRandom* for all *n*, therefore, the average total cost is lower: 10.55% vs 11.80%.

In addition, we see in table 4.3 that both algorithms are very fast, being *Minimals* the one that takes more time with *2.49 ms* compared to *0.81 ms* that its brother *MinimalsRandom* takes. In graph 7.2, we can also see that the trend of the times is better in *MinimalsRandom*, increasing much more slowly than in *Minimals*.

Table 4.2: Times: *MinCost ST* vs *MinCost MT*

N	MinCost ST (ms)	MinCost MT (ms)
3	15.62600	0.00000
4	0.00000	0.00000
5	15.62580	15.86260
6	124.76500	0.00000
7	1248.70730	16.92470
8	12241.75300	382.08370
9	148657.78950	3532.31900
10	1894837.47120	33778.26790
11	27693420.91140	384026.80100
12	439734209.94930	5624477.70390
Total time (ms)	469484772.59850	6046229.96280

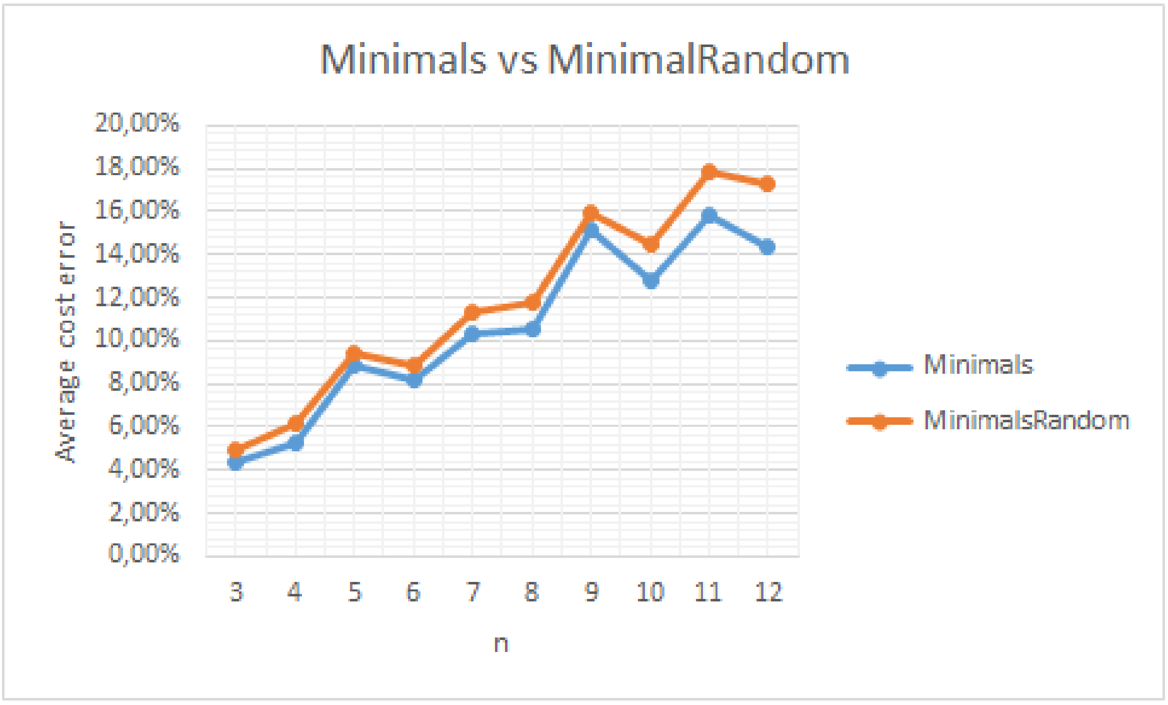


Figure 4.1: Cost errors: *Minimals* vs *MinimalsRandom*

However, since both execution times are so low, we do not consider the difference in time to be important, we therefore prioritize the cost. Therefore, for this research, ***Minimals*** is the best of this family, despite the fact that the cost/time ratio of *MinimalsRandom* is better.

Table 4.3: Times: *Minimals* vs *MinimalsRandom*

N	**Minimals (ms)**	**MinimalsRandom (ms)**
3	0.21945	0.03991
4	0.09973	0.05983
5	0.11969	0.07976
6	0.19945	0.05984
7	0.15957	0.03990
8	0.29928	0.07985
9	0.31924	0.11970
10	0.31931	0.11971
11	0.39789	0.09975
12	0.35854	0.11931
Total time (ms)	2.49215	0.81756

4.5 Sorting Algorithms

In this section what we will be doing is the following:

1. Check algorithm by algorithm which comparison method (section 4.5) is better.
2. Check which sorting algorithm is the best.

4.5.1 Bubble

For the *Bubble* algorithm we can see in table 7.2 and graph 4.2 that the comparison method **A** generates aggregations with a lower cost.

In addition, we see that the execution times are very similar for both comparison methods (table 4.4).

Because of this, we consider ***BubbleA*** to be the best of this family.

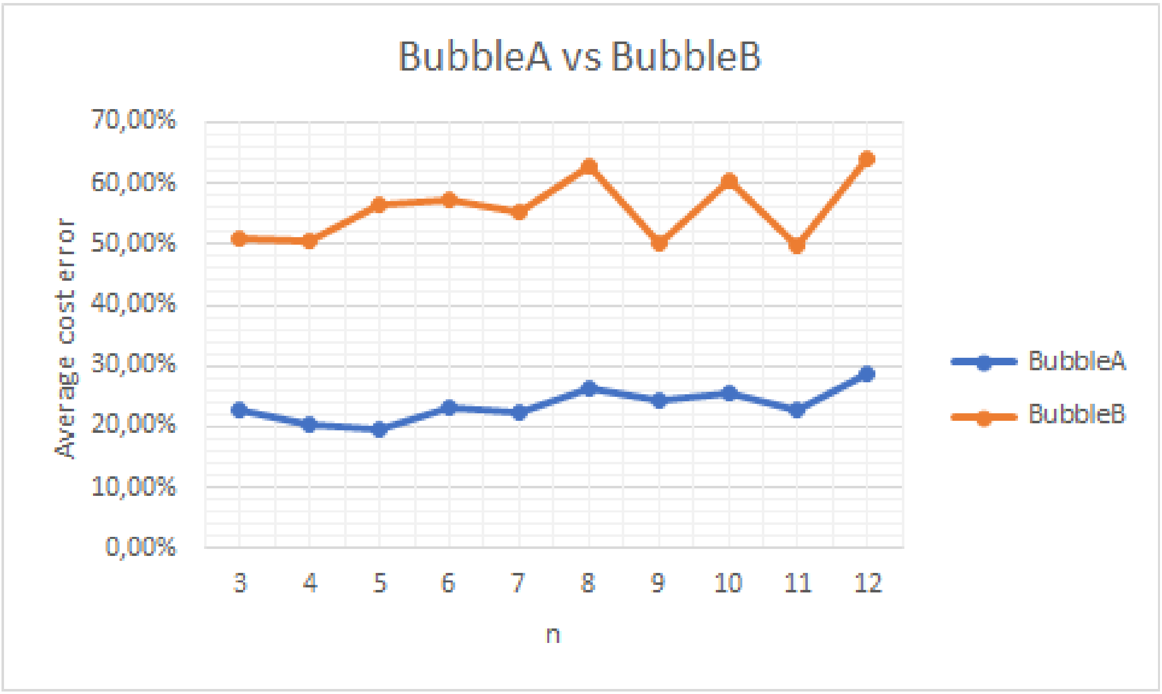


Figure 4.2: Cost errors: *BubbleA* vs *BubbleB*

Table 4.4: Times: *BubbleA* vs *BubbleB*

N	**BubbleA (ms)**	**BubbleB (ms)**
3	0.09973	0.01995
4	0.00000	0.01997
5	0.03989	0.00000
6	0.03991	0.07977
7	0.00000	0.01995
8	0.01994	0.09951
9	0.03989	0.00000
10	0.15964	0.07985
11	0.11869	0.17753
12	0.07919	0.09814
Total time (ms)	0.59689	0.59466

4.5.2 Selection

Again, we see in table 7.3 and graph 4.3 that the ***A*** comparison method generates aggregations with lower costs than the B method.

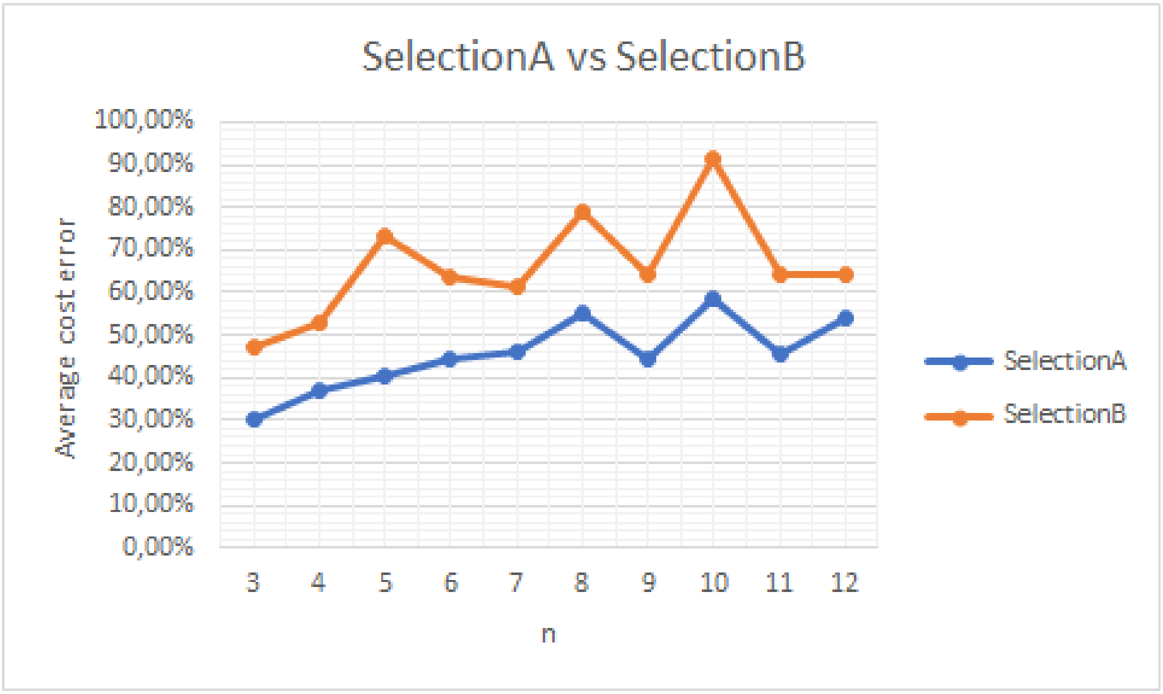


Figure 4.3: Cost errors: *SelectionA* vs *SelectionB*

We also see that the execution time is 2 times less than that of method B (table 4.5).

Because of this, we can conclude that the best one is ***SelectionA***.

4.5.3 Insertion

Looking at table 7.4 and graph 4.4, we see how the comparison method ***A*** generates better costs than B.

In this case, the times of the comparison method ***B*** are better than those of comparison method A (see table 4.6). However, as mentioned before, in these cases where the execution times are so small we prioritize the costs.

For this reason, the best algorithm is ***InsertionA***.

Table 4.5: Times: *SelectionA* vs *SelectionB*

N	**SelectionA (ms)**	**SelectionB (ms)**
3	0.01995	0.01995
4	0.00000	0.07979
5	0.00000	0.03987
6	0.00000	0.07979
7	0.01994	0.07976
8	0.07981	0.05987
9	0.01996	0.03991
10	0.01996	0.09929
11	0.03990	0.11859
12	0.13809	0.17900
Total time (ms)	0.33762	0.79582

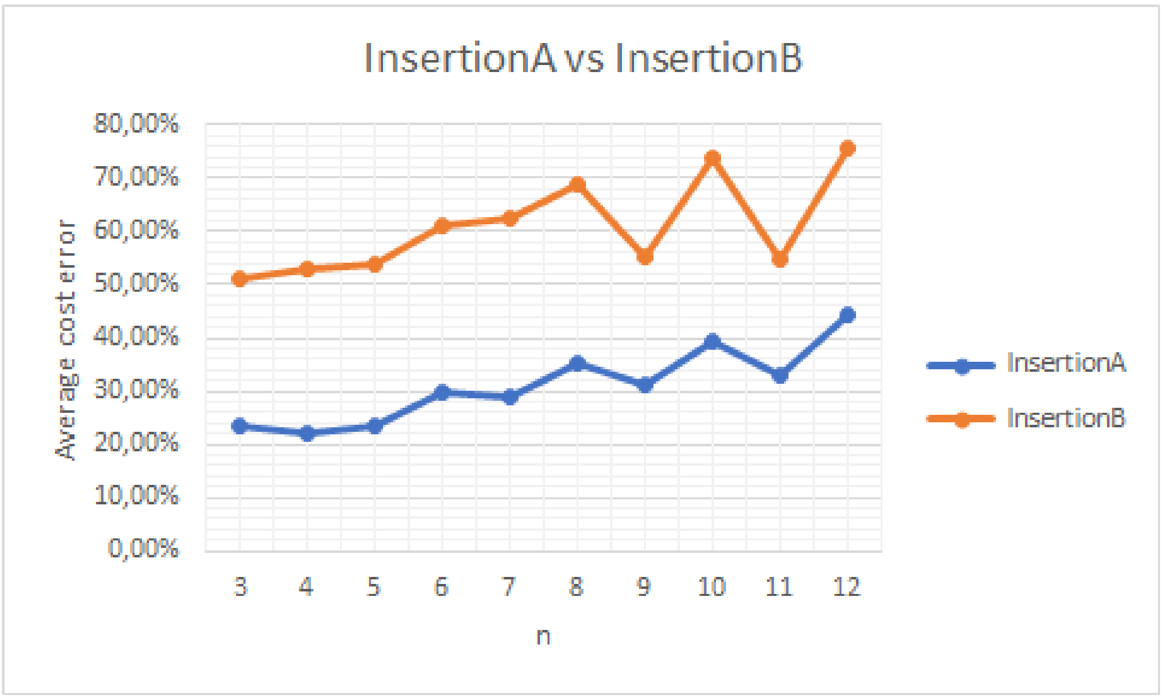


Figure 4.4: Cost errors: *InsertionA* vs *InsertionB*

Table 4.6: Times: *InsertionA* vs *InsertionB*

N	InsertionA (ms)	InsertionB (ms)
3	0.09974	0.00000
4	0.01995	0.00000
5	0.09972	0.03988
6	0.11968	0.03989
7	0.19944	0.05984
8	0.17953	0.00000
9	0.15961	0.05985
10	0.07982	0.11972
11	0.13872	0.05981
12	0.13906	0.11972
Total time (ms)	1.23528	0.49870

4.5.4 QuickSort

In the *QuickSort* algorithm we can see in table 7.5 and graph 4.5 again a big difference, in terms of cost, of the comparison method ***A*** over B.

However, in terms of time, as we have seen in other cases, the comparison method ***B*** is faster than A (table 4.7). But speaking of such small execution times, for this research we are more interested in the cost obtained.

Table 4.7: Times: *QuickSortA* vs *QuickSortB*

N	QuickSortA (ms)	QuickSortB (ms)
3	0.03991	0.00000
4	0.01995	0.00000
5	0.01996	0.03990
6	0.01993	0.00000
7	0.00000	0.00000
8	0.05986	0.01995
9	0.09972	0.01996
10	0.07981	0.03991
11	0.07983	0.06038
12	0.06058	0.00000
Total time (ms)	0.47955	0.18009

Therefore, again the algorithm with the comparison method A, ***QuickSortA***, is the best.

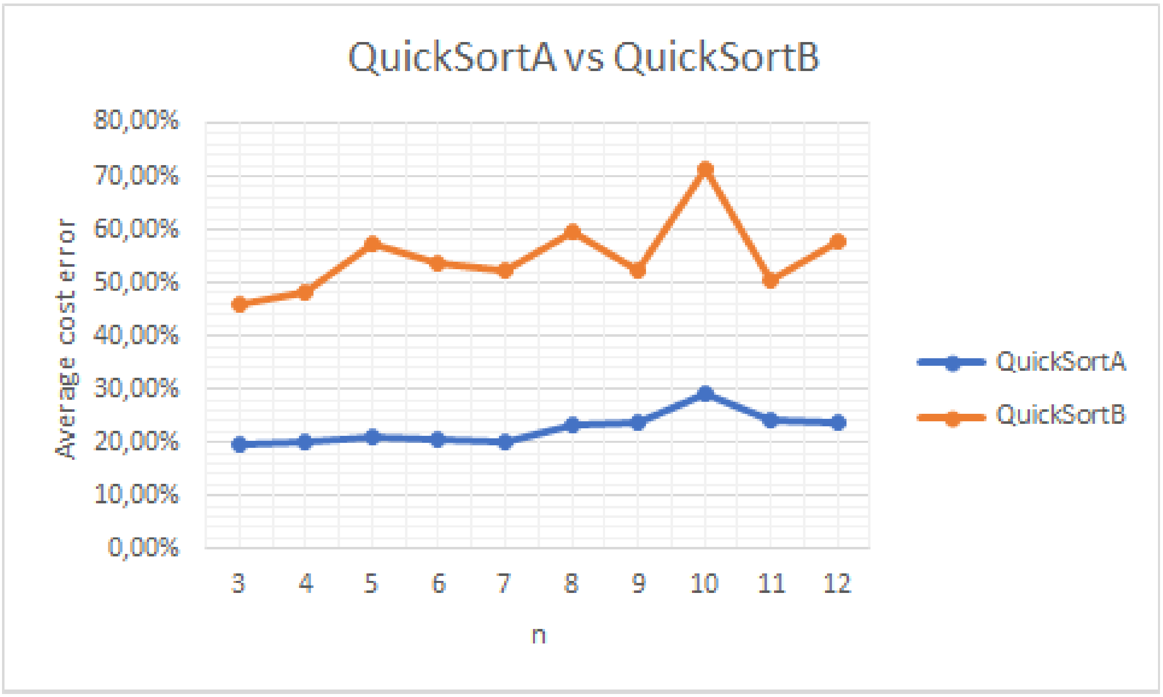


Figure 4.5: Cost errors: *QuickSortA* vs *QuickSortB*

4.5.5 Mergesort

Again we see in table 7.6 and graph 4.6 that the costs of the aggregations generated by the A-comparison method are better than those of the B-method.

We can also see in table 4.8 that the execution time is less than that of comparison method B.

So the best algorithm is ***MergeSortA***.

4.5.6 General

After analyzing each sorting algorithm individually, we can see that that comparison method ***A*** is the one that generates the best costs. Once we know which comparison method is the best, we have to know which sorting algorithm is the best.

Looking at table 7.7 and graph 4.7 we see that two sorting algorithms that the two algorithms that generate aggregations with lower costs are *QuickSortA* and *BubbleA*.

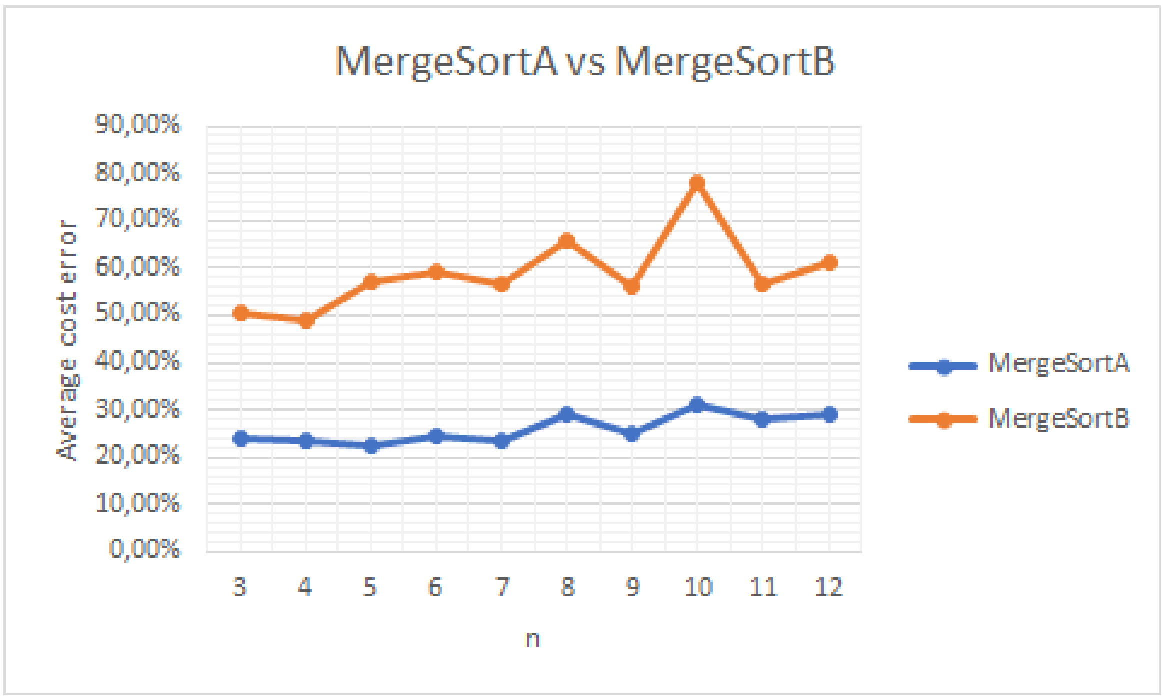


Figure 4.6: Cost errors: *MergeSortA* vs *MergeSortB*

Table 4.8: Times: *MergeSortA* vs *MergeSortB*

N	**MergeSortA (ms)**	**MergeSortB (ms)**
3	0.00000	0.01994
4	0.03991	0.03987
5	0.01996	0.05986
6	0.01995	0.11969
7	0.03989	0.15961
8	0.07983	0.07981
9	0.07981	0.13966
10	0.15957	0.12018
11	0.04050	0.25975
12	0.12090	0.22100
Total time (ms)	0.60032	1.21937

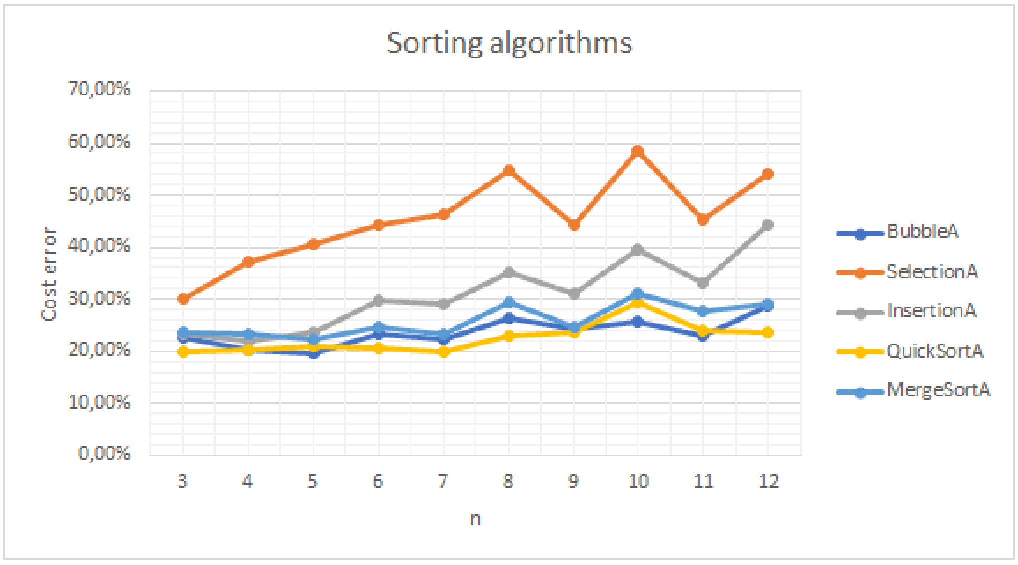


Figure 4.7: Cost errors: *Sorting algorithms*

The execution times of both algorithms are very low, with only tenths of a difference between them. We can observe in table 4.9 that the algorithm that has the shortest time of both is *QuickSortA*. In addition, in figure 4.8, we can see not only the times, but also the trends in execution times. In it, we can see that although *QuickSortA* is not the fastest of the five algorithms, its trends seems to indicate that if we increase the number of objects, the execution time would be the lowest of the five.

Table 4.9: Times: *Sorting algorithms*

N	BubbleA (ms)	SelectionA (ms)	InsertionA (ms)	QuickSortA (ms)	MergeSortA (ms)
3	0.09973	0.01995	0.09974	0.03991	0.00000
4	0.00000	0.00000	0.01995	0.01995	0.03991
5	0.03989	0.00000	0.09972	0.01996	0.01996
6	0.03991	0.00000	0.11968	0.01993	0.01995
7	0.00000	0.01994	0.19944	0.00000	0.03989
8	0.01994	0.07981	0.17953	0.05986	0.07983
9	0.03989	0.01996	0.15961	0.09972	0.07981
10	0.15964	0.01996	0.07982	0.07981	0.15957
11	0.11869	0.03990	0.13872	0.07983	0.04050
12	0.07919	0.13809	0.13906	0.06058	0.12090
Total time (ms)	0.59689	0.33762	1.23528	0.47955	0.60032

Therefore, the best algorithm of the sorting family is ***QuickSortA***.

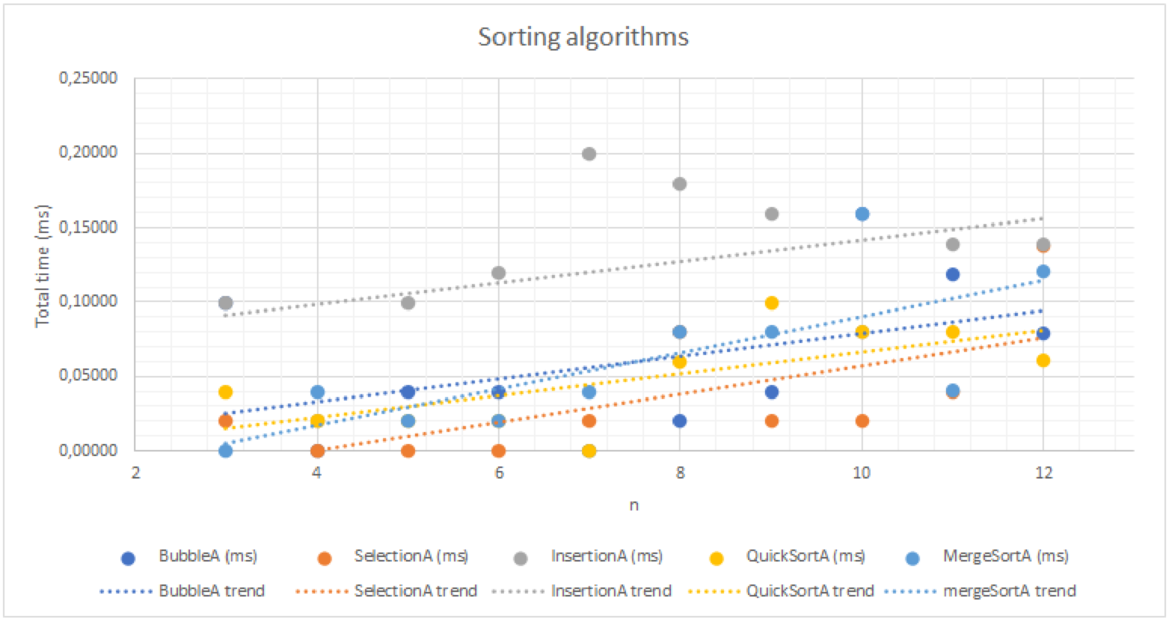


Figure 4.8: Times: *Sorting algorithms*

4.6 Simulated Annealing: Initial algorithm

We have combined *Simulated Annealing* with *Minimals Random*, *Bubble*, *Insertion*, *Selection*, *MergeSort*, *Random*, *Minimals* and *QuickSort*. However, only the results of the last three combinations are shown since they are the most significant.

Firstly, we need to bear in mind that the focus of this section is to demonstrate that the quality of the initial solution, and therefore the initial algorithm, influences the quality of the final solution obtained by this optimization process.

What we will do is to test this algorithm in combination with the best algorithm of the 4.4 and 4.5 families. Therefore, the simulated annealing will be tested by passing as initial solution a *linear extension* randomly computed, an aggregation computed with *Minimals* and another one computed with *QuickSort*, with a low initial temperature and a slow cooling constant:

$$T = 4 \qquad \beta = 0.97$$

Looking at table 7.8 and figure 4.9, the difference in the costs incurred depending on the initial algorithm can be observed. It is clearly seen that our hypothesis is confirmed, since both executions with *Minimals* and *QuickSort* have lower costs than the one with an initial random solution. When comparing passing *Minimals* or *QuickSort* as initial solution, it is also clear that the first one obtains better costs.

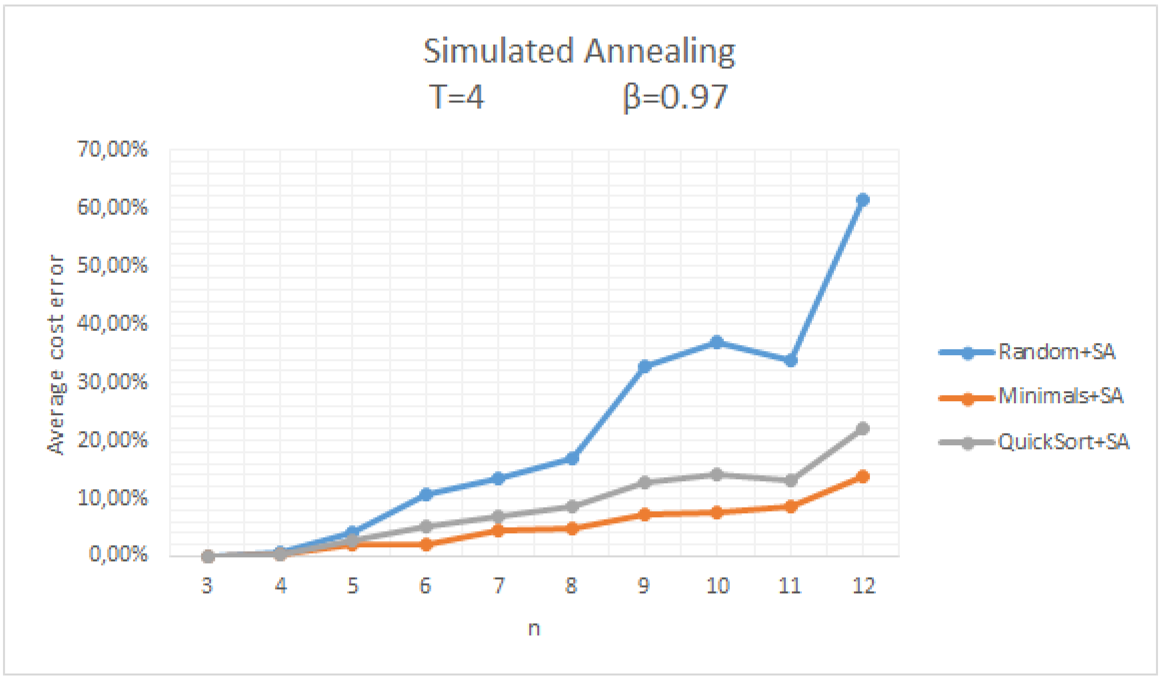


Figure 4.9: Cost errors: *Simulated Annealing (T=4,β=0.97)*

It can also be seen in table 4.10 and figure 7.3 that the difference in the times of the two executions with *Minimals* and *QuickSort* are slightly greater than the one with a random initial solution. Moreover, the difference between the execution times of *Minimals+SA* is only 2*ms* slower than the one with *QuickSort*, so it is not significant for this project.

Therefore, we can confirm that the initial algorithm clearly affects the quality of the final solution, and therefore ***Minimals+SA*** is the best of this family.

4.7 Linear Programming

As we can see in table 4.11, this algorithm always computes the optimum aggregation, so the costs obtained are always the minimum.

The execution times are greater than those of the previous families, as we can observe in table 7.9. But the truly interesting fact is the tendency of the execution times, shown in graph 4.10, maybe being in this case exponential or polynomial, so in case of continuing increasing the number of objects, these times would increase considerably.

Table 4.10: Times: *Simulated Annealing (T=4,β=0.97)*

N	Random+SA (ms)	Minimals+SA (ms)	QuickSort+SA (ms)
3	0.53877	1.01377	0.45889
4	0.59933	0.61558	0.44168
5	0.65964	0.81349	0.60191
6	0.71753	0.89735	0.81871
7	0.91793	1.03734	0.87776
8	0.97734	1.03757	0.99731
9	1.15690	1.33654	1.15665
10	1.25664	1.51616	1.27633
11	1.31703	1.61399	1.43779
12	1.47622	1.81819	1.55686
Total time (ms)	9.61733	11.69999	9.62389

Table 4.11: Costs: *Linear Programming*

N	LinearProgramming	MinCost MT
3	6.857	6.857
4	13.592	13.592
5	23.184	23.184
6	33.449	33.449
7	47.796	47.796
8	60.082	60.082
9	75.490	75.490
10	91.265	91.265
11	110.898	110.898
12	131.184	131.184
Average cost	59.380	59.380

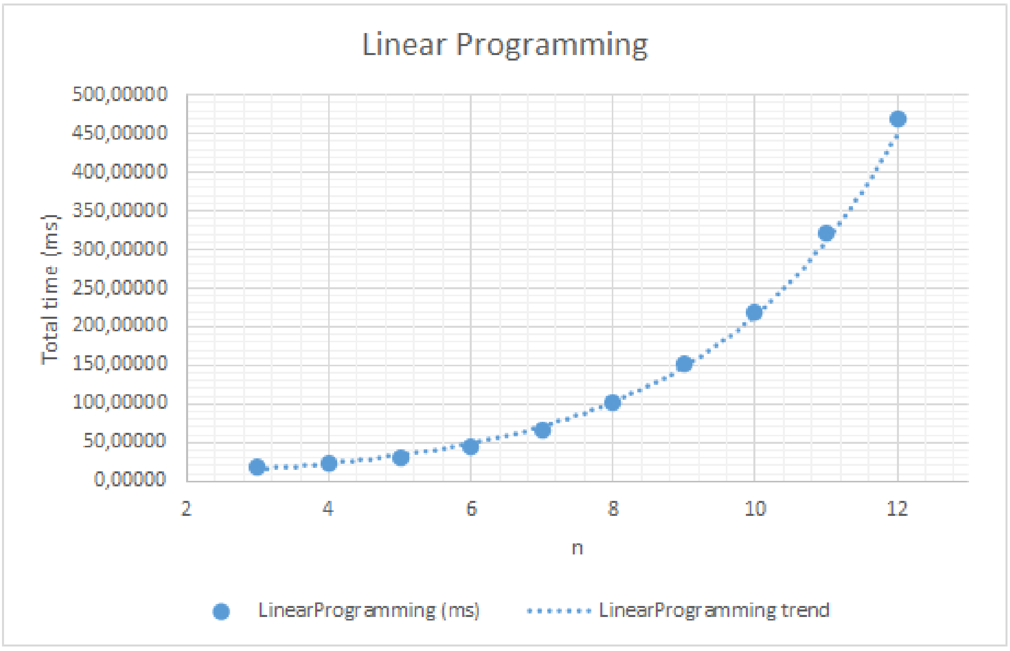


Figure 4.10: Times: *Linear Programming*

4.8 Best aggregation method

Once we have analyzed all the families of algorithms and decided which is the best algorithm of each one, we must analyze these as a whole to have a global vision of all.

The algorithms that are going to be analyzed in this section are the following: *Minimals*, *QuickSortA*, *Minimals+Simulated Annealing* and *Linear Programming*. In the case of *Simulated Annealing*, we will execute it with

$$T = 4 \qquad \beta = 0.97$$

and it will be called *Minimals+SA LT*, and another execution with

$$T = 100 \qquad \beta = 0.999$$

that will be called *Minimals+SA HT* in order to prove that, increasing both the temperature and cooling constant, apart from increasing the number of iterations, generates aggregations with better costs.

The first thing we want to confirm is that, the higher the temperature and the cooling constant, the better the cost of the aggregations generated by the *Simulated*

Annealing algorithm. This fact can be confirmed in table 4.12 and figure 4.11, showing that the costs obtained by *Minimals+SA HT* are lower than the ones obtained by *Minimals+SA LT*.

It can also be observed how, when the parameters are increased and consequently, the iterations, the execution times are greater (see table 4.13).

Table 4.12: Cost errors: Best *aggregation* method

N	Minimals	QuickSort	Minimals+SA LT	Minimals+SA HT	Linear Programming
3	4.19%	23.47%	0.00%	0.00%	0.00%
4	5.22%	21.63%	0.27%	0.00%	0.00%
5	5.90%	19.73%	1.11%	0.00%	0.00%
6	9.36%	20.81%	2.87%	0.00%	0.00%
7	11.82%	22.74%	5.61%	0.01%	0.00%
8	15.61%	27.60%	8.59%	0.16%	0.00%
9	13.90%	24.69%	7.40%	0.27%	0.00%
10	14.15%	22.17%	8.67%	0.46%	0.00%
11	12.72%	23.03%	8.49%	1.03%	0.00%
12	15.87%	26.12%	11.32%	1.22%	0.00%
Average cost error	10.87%	23.20%	5.43%	0.31%	0.00%

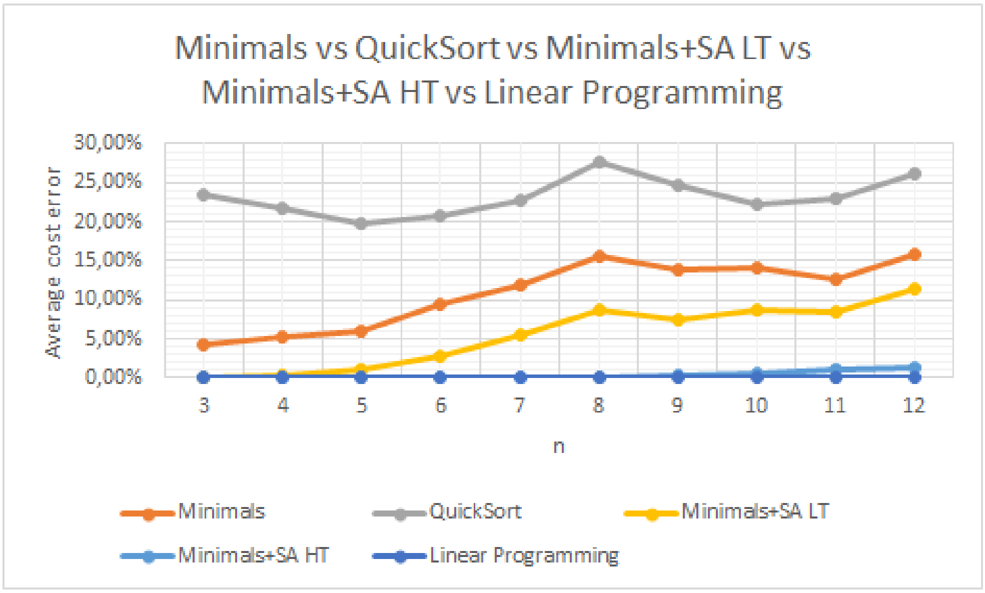


Figure 4.11: Cost errors: Best *aggregation* method

Once our initial assumption is verified. we can see how the *Minimals* algorithm obtains better costs than the *QuickSortA* algorithm. Moreover, when combining *Minimals* with the *Simulated Annealing* optimization algorithm, the cost errors are lower than using only the first one. Therefore, we can ensure the effectiveness of using the optimization algorithm (see table 4.12 and figure 4.11).

Table 4.13: Times: Best *aggregation method*

N	Minimals (ms)	QuickSort(ms)	Minimals+SA LT (ms)	Minimals+SA HT (ms)	Linear Programming (ms)	MinCost MT (ms)
3	0.19894	0.03990	0.63184	55.24363	6963.14810	32.82010
4	0.09810	0.01943	0.52893	61.15640	6234.57310	0.97050
5	0.19574	0.01995	0.60648	79.48764	6438.74210	1.99220
6	0.19478	0.05738	0.69495	89.79747	7002.72700	12.14280
7	0.25991	0.04095	0.96700	97.62566	7868.88970	41.58960
8	0.19948	0.05933	1.00327	106.92099	9126.45090	264.87810
9	0.27718	0.05987	1.24245	134.00389	11065.30240	2569.49330
10	0.35919	0.05987	1.39723	148.50157	13921.74060	27600.38580
11	0.35117	0.09916	1.60339	158.93650	18212.21800	275991.06630
12	0.47300	0.06042	1.74745	172.52270	23393.36860	3147048.23830
Total time (ms)	2.60750	0.51625	10.42301	1104.19646	110227.16050	3453563.57700

Furthermore, if you look at table 4.13, you can see that the runtimes of combining *Simulated Anneling* with another algorithm are greater, but much more lower than the times of *MinCost*.

In general terms, when we are to compare any algorithm with *Linear Programming* we can see that all, except *Minimals+SA HT*, are far from obtaining similar costs. When executing *Minimals+SA* with

$$T = 100 \qquad \beta = 0.999$$

we can noticed that the costs obtained are practically the same as with *Linear Programming* (see table 4.12 and figure 4.11).

On the other hand, if we look at table 4.13 we can see that the difference in the execution times of these two algorithms is quite vast, *Linear programming* takes 100 times longer than *Minimals+SA HT* at the cost of ensuring that the minimum cost is always obtained.

Moreover, if we look closely at the trend of the execution times, we can see that the trend of *Minimals+SA HT* (see figure 4.12) is more or less linear while the trend of *Linear Programming* (see figure 4.10) may seem exponential or polynomial.

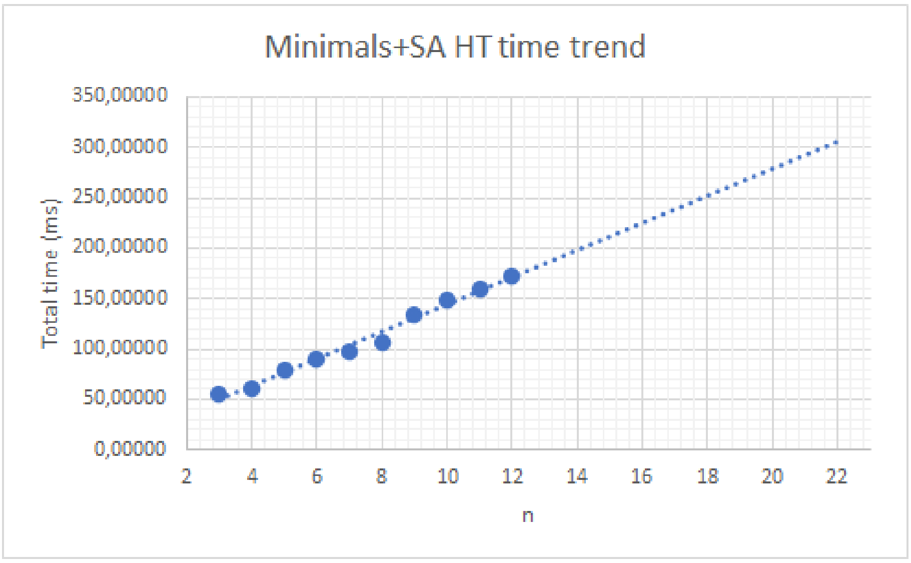


Figure 4.12: Time trend: *Minimals + SA HT*

5 Conclusions

After having analyzed the best algorithm of each family in section 4, it can be observed how difficult it is to determine which algorithm, based on costs and times, is better. Not only do we need to look at the data obtained, but also at how the algorithm is supposed to behave if the number of objects is increased. In regards of the latter, it has been verified that observing the tendency is of the foremost importance.

Besides, it is difficult to determine which of the following is more relevant: the costs or the times. Therefore, it is the user who would need to prioritise on one of these two parameters. In our case, we are interested in obtaining aggregations at the best possible cost, as long as it is within a reasonable time.

Therefore, when analysing the different families, we have always chosen the algorithms that generate aggregations with the lowest costs or that take less time for the same cost: *Minimals*, *QuickSort*, *Minimals+SA*, *Linear Programming* and *MinCost MT*.

After conducting the experiments, the algorithms could be classified into two main categories:

1. **Optimal algorithms**: those that ensure that the best aggregation is always obtained, i.e. aggregation with the minimum cost. This category would include *MinCost MT* and *Linear Programming*.
2. **Non-optimal algorithms**: those that do not always ensure an optimal solution. The rest of the algorithms we have studied would fall under this category.

Having said that, we may draw the following conclusions from this project:

- The *MinCost MT* algorithm is a trivial algorithm, that is, it is known that it will always obtain the aggregation with the lowest cost by calculating the cost of all the possible aggregations and keeping the aggregations with the best cost. The gain in execution time with respect to its single-threaded brother is particularly striking. Besides, we could still test this algorithm for $n = 13$ and the execution times would still be reasonable

$$T_{13} \approx T_{12} \times 13$$

since the result would be about 21 hours.

- Despite the above-mentioned, with *Linear Programming* algorithm we would always get the aggregation with the optimal cost but with much lower execution times than with the previous algorithm. Therefore, should we want to further increase the number of objects, *Linear Programming* would be a more suitable option than *MinCost MT*.

- The quality of the initial solution passed as input in the *Simulated Annealing* algorithm influences the quality of the final solution. The better the initial solution, the better the final solution.

- The execution times of *Minimals*, *QuickSort* and *Minimals+SA HT* are very low and would allow to continue aggregating larger *posets*. Considering these three agorithms, the one that aggregates *posets* with the lower costs is *Minimals+SA HT*, being these costs remarkably close to the optimal ones. Consequently, *Minimals+SA HT* is the best algorithm of the "non-optimal" ones.

- It has also been proven that increasing the temperature and cooling constant in the *Simulated Annealing* algorithm means generating more cost-effective aggregations.

  At this point the following question may arise: *If we continue increasing the temperature and the cooling constant, i.e. the iterations of the algorithm, would it be possible to generate aggregations with the minimum cost also with this algorithm?* Everything seems to indicate that this is likely the case, but it has not been demonstrated in this dissertation.

Finally, should we have to choose just one algorithm to aggregate *posets*, we would opt for *Simulated Annealing* with high temperature and cooling constant, passing an aggregation generated with *Minimals* as the initial solution. Despite the fact that the costs obtained are not the optimum ones, they are very close to them and the executions times and the trend of the times are better than the ones of the optimal algorithms.

6 Research limitations

The experiments of this project had started to be executed in a Mac server of the University of Oviedo, but we needed to search for an alternative due to a compatibility problem with the library used for the linear programming part [EN18].

Because of this, the experiments had to be executed on the personal computer of a colleague. The experiments were executed at night, trying to make the processor carry exclusively the load of the experiments. Nevertheless, it is possible that the fact of using a personal computer has impacted the investigation because it is likely that at some times the processor was affected by other separate processes, which could add some alterations in the times obtained.

As it was a personal computer, it was not feasible to run the experiment of section 4.3 due to its execution times. Therefore, this experiment was the only one executed in a virtual machine of Azure. Since we simply counted with the available credits granted by the University of Oviedo, we were only able to configure a slower machine than the personal computer.

The fact of having executed the experiments in two different machines does not affect the conclusions, since the time and cost comparisons have always been carried out between algorithms executed in the same machine.

Finally, it would also have been interesting to have been able to run all the experiments on the University server, this way testing up to $n = 13$.

7 Annexes

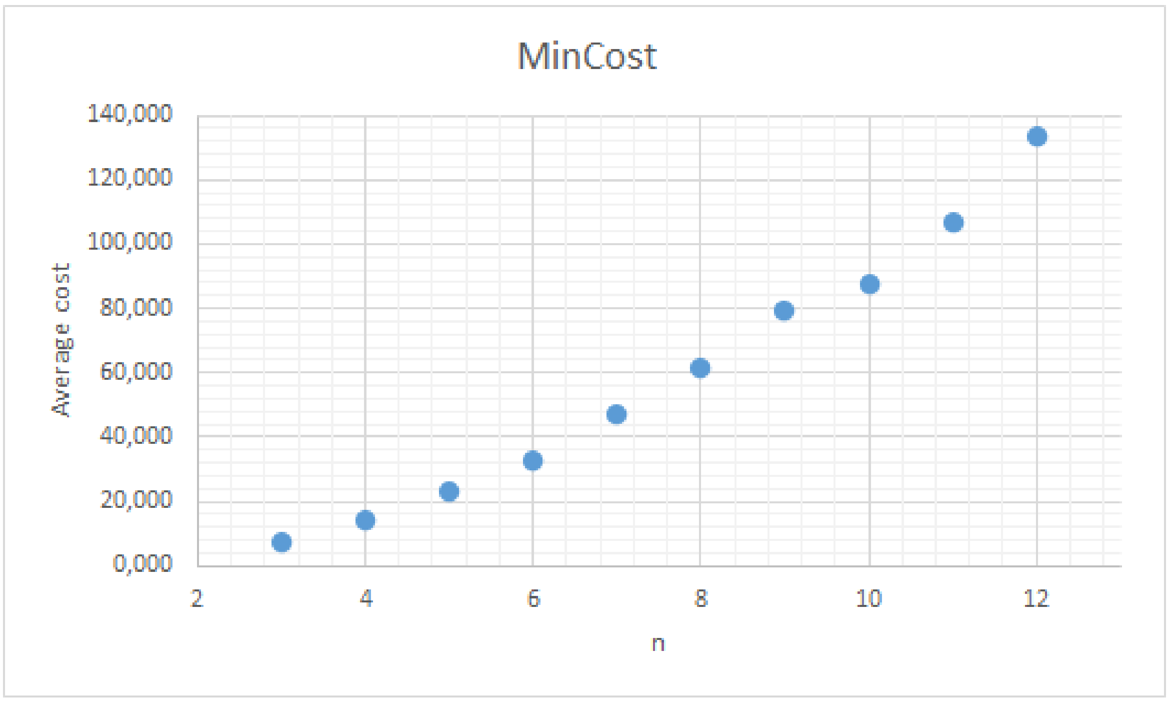


Figure 7.1: Costs: *MinCost*

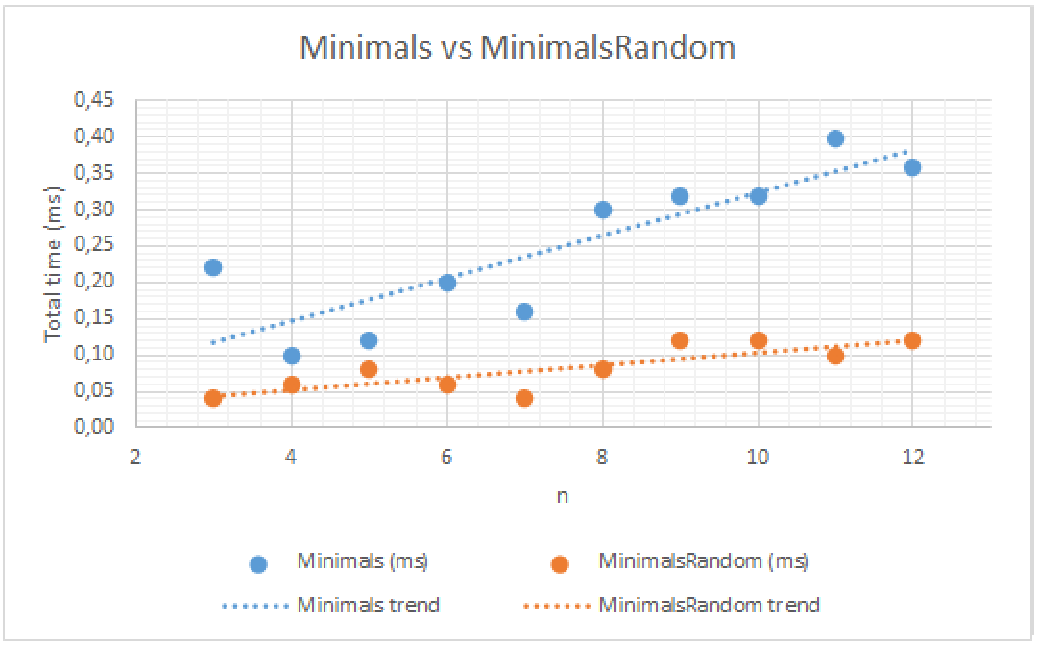


Figure 7.2: Times: *Minimals* vs *MinimalsRandom*

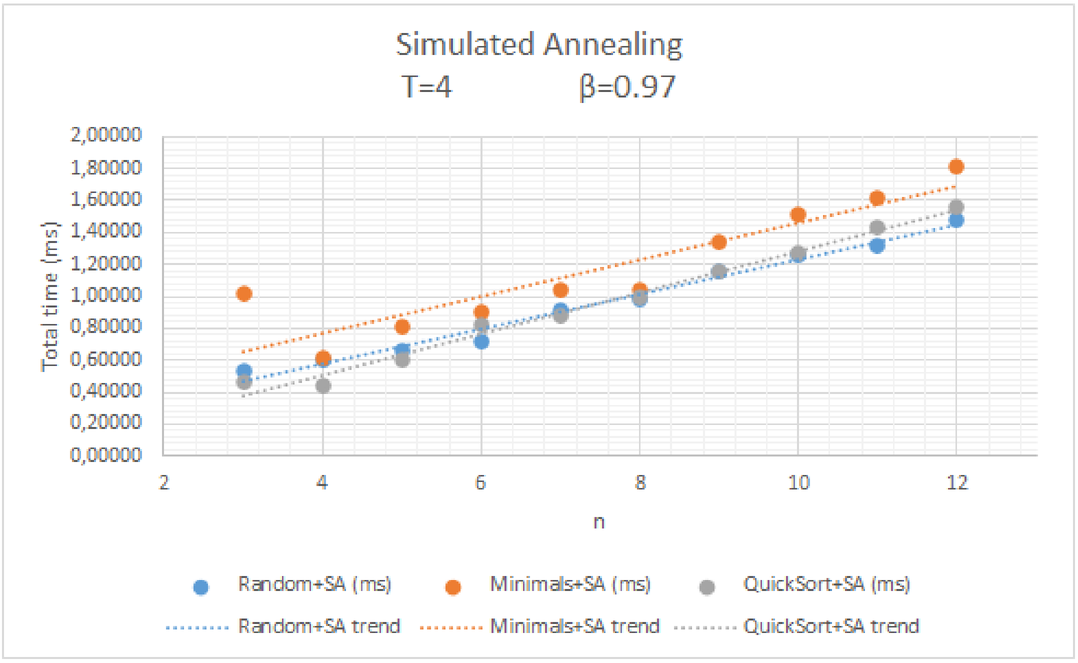


Figure 7.3: Times: *Simulated Annealing (T=4,β=0.97)*

Table 7.1: Costs: *Minimals* vs *MinimalsRandom*

N	Minimals	MinimalsRandom
3	4.32%	4.96%
4	5.24%	6.19%
5	8.87%	9.37%
6	8.18%	8.89%
7	10.28%	11.37%
8	10.54%	11.77%
9	15.15%	15.90%
10	12.75%	14.43%
11	15.80%	17.88%
12	14.33%	17.27%
Average cost error	10.55%	11.80%

Table 7.2: Cost errors: *BubbleA* vs *BubbleB*

N	BubbleA	BubbleB
3	22.77%	50.91%
4	20.30%	50.46%
5	19.51%	56.29%
6	23.28%	57.18%
7	22.45%	55.46%
8	26.48%	62.93%
9	24.22%	49.97%
10	25.54%	60.49%
11	22.88%	49.85%
12	28.63%	64.21%
Average cost error	23.61%	55.77%

Table 7.3: Cost errors: *SelectionA* vs *SelectionB*

N	**SelectionA**	**SelectionB**
3	30.09%	47.41%
4	37.17%	53.09%
5	40.51%	73.17%
6	44.37%	63.93%
7	46.14%	61.44%
8	54.92%	78.90%
9	44.38%	64.06%
10	58.46%	91.38%
11	45.42%	64.12%
12	54.24%	64.39%
Average cost error	45.57%	66.19%

Table 7.4: Cost errors: *InsertionA* vs *InsertionB*

N	**InsertionA**	**InsertionB**
3	23.40%	51.34%
4	22.12%	52.78%
5	23.61%	53.69%
6	29.78%	61.11%
7	28.90%	62.28%
8	35.03%	68.60%
9	31.18%	55.01%
10	39.47%	74.02%
11	33.02%	54.54%
12	44.22%	75.80%
Average cost error	31.07%	60.92%

Table 7.5: Cost errors: *QuickSortA* vs *QuickSortB*

N	**QuickSortA**	**QuickSortB**
3	19.79%	45.83%
4	20.13%	48.03%
5	20.87%	57.40%
6	20.66%	53.66%
7	19.90%	52.18%
8	23.13%	59.45%
9	23.81%	52.28%
10	29.27%	71.43%
11	24.04%	50.55%
12	23.66%	57.74%
Average cost error	22.52%	54.85%

Table 7.6: Cost errors: *MergeSortA* vs *MergeSortB*

N	**MergeSortA**	**MergeSortB**
3	23.74%	50.68%
4	23.28%	49.06%
5	22.31%	57.27%
6	24.49%	58.95%
7	23.34%	56.64%
8	29.27%	65.87%
9	24.77%	56.17%
10	31.20%	77.91%
11	27.83%	56.75%
12	28.91%	61.32%
Average cost error	25.91%	59.06%

Table 7.7: Cost errors: *Sorting algorithms*

N	BubbleA	SelectionA	InsertionA	QuickSortA	MergeSortA
3	22.77%	30.09%	23.40%	19.79%	23.74%
4	20.30%	37.17%	22.12%	20.13%	23.28%
5	19.51%	40.51%	23.61%	20.87%	22.31%
6	23.28%	44.37%	29.78%	20.66%	24.49%
7	22.45%	46.14%	28.90%	19.90%	23.34%
8	26.48%	54.92%	35.03%	23.13%	29.27%
9	24.22%	44.38%	31.18%	23.81%	24.77%
10	25.54%	58.46%	39.47%	29.27%	31.20%
11	22.88%	45.42%	33.02%	24.04%	27.83%
12	28.63%	54.24%	44.22%	23.66%	28.91%
Average cost error	23.61%	45.57%	31.07%	22.52%	25.91%

Table 7.8: Cost errors: *Simulated Annealing (T=4,β=0.97)*

N	Random+SA	Minimals+SA	QuickSort+SA
3	0.02%	0.00%	0.00%
4	0.79%	0.21%	0.31%
5	3.97%	1.94%	2.77%
6	10.58%	1.92%	5.26%
7	13.28%	4.30%	6.79%
8	16.74%	4.73%	8.75%
9	32.86%	7.17%	12.65%
10	37.01%	7.74%	14.16%
11	33.81%	8.77%	13.14%
12	61.61%	13.79%	22.24%
Average cost error	21.07%	5.06%	8.61%

Table 7.9: Times: *Linear Programming*

N	**LinearProgramming (ms)**
3	19.67886
4	23.43813
5	29.37552
6	45.00179
7	66.26160
8	101.08387
9	151.47112
10	218.42127
11	321.49887
12	469.38988
Total time (ms)	1445.62090

References

[Bac+15] Christian Bachmaier et al. "On the hardness of maximum rank aggregation problems". In: *Journal of Discrete Algorithms* 31 (2015). 24th International Workshop on Combinatorial Algorithms (IWOCA 2013), pp. 2–13. ISSN: 1570-8667. DOI: https://doi.org/10.1016/j.jda.2014.10.002. URL: http://www.sciencedirect.com/science/article/pii/S1570866714000707.

[Cam18] Omar Antolín Camarena. *Linear Programming*. 2018. URL: https://www.matem.unam.mx/~omar/math340/math340notes.pdf.

[CDM13] E.F. Combarro, I. Díaz, and P. Miranda. "On random generation of fuzzy measures". In: *Fuzzy Sets and Systems* 228 (2013), pp. 64–77.

[Dep15] Carnegie Mellon University Department of Mathematical Sciences. *Partially Ordered Sets*. Oct. 2015. URL: https://www.math.cmu.edu/~af1p/Teaching/Combinatorics/Slides/Posets.pdf.

[Dré+06] Johann Dréo et al. *Metaheuristics for Hard Optimization: Simulated Annealing, Tabu Search, Evolutionary and Genetic Algorithms, Ant Colonies,... Methods and Case Studies*. 1st ed. Chapter 1. 2006. ISBN: 978-3-540-23022-9,978-3-540-30966-6.

[EN18] Kjell Eikland and Peter Notebaert. *lpsolve*. May 2018. URL: https://sourceforge.net/projects/lpsolve/.

[FHD19] Elías Fernández Combarro, Julen Hurtado de Saracho, and Irene Díaz Rodríguez. "Minimals Plus: An improved algorithm for the random generation of linear extensions of partially ordered sets". In: *Information Sciences* 501 (2019), pp. 50–67.

[Gre18] David Green. *Constraint Satisfaction Programming With Microsoft Solver Foundation*. Apr. 2018. URL: https://www.appliedis.com/constraint-satisfaction-programming-with-microsoft-solver-foundation/.

[Jac13] Lee Jacobson. *Simulated Annealing for beginners*. 2013. URL: http://www.theprojectspot.com/tutorial-post/simulated-annealing-algorithm-for-beginners/6.

[JS00] Jaroslav Jezek and Vaclak Slavik. "Random Posets, Lattices and Lattices Terms". In: *Mathematica Bohemica* 125.2 (2000).

[Kri17] Juri Krivoruchko. *Solving optimization problems with Microsoft Solver Foundation*. Apr. 2017. URL: https://www.codeproject.com/Articles/1183168/Solving-optimization-problems-with-Microsoft-Solve.

[OS18] Eric Ouellet and Omar Saad. *Permutations: Fast implementations and a new indexing algorithm allowing multithreading*. July 2018. URL: https://www.codeproject.com/Articles/1250925/Permutations-Fast-implementations-and-a-new-indexi.

[PK00] D. T. Pham and D. Karaboga. *Intelligent Optimisation Techniques: Genetic Algorithms, Tabu Search, Simulated Annealing and Neural Networks*. 1st ed. Chapter 4. 2000. ISBN: 978-1-4471-1186-3,978-1-4471-0721-7.

[Ste04] Cliff Stein. *Linear Programming Definition*. 2004. URL: http://www.columbia.edu/~cs2035/courses/ieor3608.F04/lpdef.pdf.

[Vid20] Analytics Vidhya. *Introductory guide on Linear Programming*. Apr. 2020. URL: https://www.analyticsvidhya.com/blog/2017/02/lintroductory-guide-on-linear-programming-explained-in-simple-english/.