UNIVERSITÀ DEGLI STUDI DI PADOVA

Faculty of Engineering

Degree Thesis

# Optimizing Rectangular Partitions of Histograms

Author

**Andrea Zoncapè**

Supervisor: Ch.mo Prof. **Geppino Pucci**

anno accademico 2009-2010

# Abstract

The problem of partitioning isothetic polygons into rectangles, where the goal is to minimise the total length of the partition, has some applications in VLSI-circuit design and construction. If these polygons have holes inside, the problem becomes NP-hard, instead, in the case of hole-free polygons, an optimal partition can be computed in $O(n^4)$ exploiting *dynamic programming*.

In this thesis, I focus my attention on rectangular partitions of histograms, which are special cases of hole-free isothetic polygons, using the fact it's possible to partition them into histograms in linear time. I've used dynamic programming in order to produce optimal partitions and a parameterised approximation algorithm, a variant of the so called "thickest first" algorithm. Then, I've compared experimentally these two algorithms to check theoretical bounds and to evaluate the goodness of the approximation algorithm, especially on some interesting histogram shapes like staircase and staircase united with its mirror image.

In parallel, an interactive tool has been implemented, with a graphical representation, which can help users to visualise the differences between these methods. The implementation of the algorithms, the auxiliary structures and the GUI of the tool have been implemented with *C++* and *Qt graphics libraries*.

# Acknowledgements

I would like to thank the University of Lund and the LTH department which gave me the possibilities to study and prepare my final master thesis. All people in LTH, especially my supervisor, prof. Christos Levcopoulos, for his dedication and for helping and suggesting me during my work, in all the five months at Lund.

I want to thank my parents for giving me the opportunity to study and develop myself in complete serenity, due to their proximity and their love. Then I would like to thank my sisters for their love and for all the beautiful moments we lived together.

I thank Rossella, there are no words to explain what she has done and she's doing for me. Her support and presence is essential in my life.

Finally I would like to thank all the people who love me and who have accompanied me in these years.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Nowadays optimisation problems represent a huge research field (always moving). Probably the *Traveling Salesman Problem* is the most famous example. It is known that it can be very difficult, if not impossible, to find optimal solution which minimise or maximise certain object functions for larger inputs. It may be prohibitive searching among all possible combinations in order to find the best solution. Maybe it is possible to obtain it, but with a cost too high to sustain (i.e., the computation time).

In VLSI circuit design we want to minimise the length of rectilinear lines that link pairs of points. The problem of partitioning isothetic polygons into rectangles, is helpful to reach this aim. In this problem, we want to divide the original polygon into empty rectangles with the goal to minimise the sum of the lengths of the segments that we are inserting. These segments are not allowed to overlap except on their endpoints. In this work I've concentrated my attention on a particular type of isothetic polygons, the histograms. A histogram has all edges parallel to the x- or the y-axis, but it also has a long edge whose length is equal to the total length of all its parallel edges.

Particularly, we analysed two algorithms to partition histograms into rectangles. The first algorithms we will present is the optimal algorithm, but as we said above, it has a high cost to afford. The second one is an approximate method that manifests a good trade-off between the quality of the solution, and the time used to obtain it. It is a variant of the *thickest* method.

In order to understand the differences between the produced solutions of these methods, a tool with a graphical interface has been developed and used.

## 1.1 Thesis objectives

The thesis has the purpose to test the goodness of some theoretical bounds that haven't been checked in practice. In particular, we used a dynamic programming approach to produce optimal partitions. Then we compared these solutions with those produced by the approximation algorithm, the *q-thickest first* algorithm (a modification of the original thickest-first algorithm). The terms we used to evaluate the validity of q-thickest algorithm are the ratio between partition lengths and the computation times.

Another goal is to realise a practical implementation of both techniques, creating a source code as clear as possible, by using simple data structures, without losses in time performance, in relation with their expected time analysis.

To allow a complete verification, the use of a graphical representation, through the interactive tool, helped us to visualise what happens. It was necessary when we were working with polygons defined by tens of vertices, since a manually scan would have been prohibitive.

## 1.2 Organisation

Now, I will present a brief description of what each chapter contains.

**Chapter 2** In this section, I've introduced the problem of partitioning polygons (histograms) into rectangles; then, I explained both algorithms. The first algorithm is the *optimal method*. I've started defining its dynamic programming scheme: the definition of minimal sub-problems and its way to build the entire solution. After that, I have illustrated the implementation of this algorithm and the data structures I used. As a final step I proved the time and space complexity associated with my implementation. Afterwards, I illustrated the approximation algorithm, the *q-thickest first* method, in the same manner that has been done for the optimal one. From the complexity analysis, the results are:

$$Optimal \begin{cases} \in O(n^3) & \text{time} \\ \in O(n^2) & \text{space} \end{cases} \qquad q-thickest \begin{cases} \in O(n \log n) & \text{expected time} \\ \in O(n) & \text{space} \end{cases}$$

**Chapter 3** For our purpose to test the goodness of the solutions of the approximation algorithm, we have performed several simulations. We have tested the behaviour of the $q$-thickest algorithm with histograms which have been generated with a randomisation technique. We defined three families of histograms: general histograms, staircase histograms and symmetric histograms. We could imagine

this last type as a staircase histogram united with its mirror image. They represents, at the moment, the "worst case" for the approximation method. As an initial step, we have studied the best values of the $q$ parameter on the different kinds of histograms. Then, we used these values to compare the two partitioning algorithms. The terms of comparison are those reported in section 1.1. The data produced in the simulation tests have been collected, and the final results we have computed is an average value on them.

The main results are two. The first is that the trend of the lengths of the approximate solutions follows the trend of the lengths of the optimal ones. The second result is that the solutions were never close to the theoretical bound of the $q$-thickest algorithm. In [9], Levcopoulos proved that every $\sqrt{2}$-thickest partition has length $< (1 + \sqrt{2}) \times M(P)$, where $M(P)$ denote the length of the optimal partition set. The average results for considered cases, are in practice better: we obtained an upper bound of about $1.1 \times M(P)$. In this way, we affirm that the $q$-thickest algorithm is preferable, by setting $q$ to a value which indicated by our test results.

At the end, we also wanted to examine the behaviour of the optimal algorithm in terms of computation time and number of really solved sub-prolems.

**Chapter 4** Inside this chapter, we explain how the GUI tool appears and how users can make use of it. We show its main features like the possibility to create or to load histograms and to visualise the information about the solutions produced by the algorithms.

**Chapter 5** In this final chapter, I outlined the results of this work and I discussed some possible improvements or changes to be done, especially on the study of the $q$ parameter. I have also introduced the question to extend the results to other possible related problems.

# Chapter 2

# Rectangular partitions of histograms

## 2.1 Introduction to the problem

The terms "rectangular partition of a polygon" indicates the process which finds a set of segments that divides the polygon into empty rectangles. Finding an optimal partition could be very difficult especially when the polygon has holes inside; in this case the problem is NP-hard. A partition is optimal when the sum of the segments' length is minimal, and it's denoted by *M(P)*.

Histograms are a special case of isothetic polygons: they have all edges parallel to $x$- or $y$-axis, and additionally they have a distinct edge, the *base*, whose length is equal to the total length of all other edges parallel to it. In this thesis, the focus is on histograms, exploiting the fact that it is possible to partition an isothetic polygon (without holes) into histograms in linear time ($O(n)$).

In the following sections, an optimal ($O(n^3)$) and an approximation algorithm ($O(n \log n)$ in the average case) to partition histograms in rectangles are presented, with the details of the implementation and the associated analysis of the time and space complexity.

At the end of the chapter the linear time algorithm to divide isothetic polygons into histograms is described.

## 2.2 Optimal method

In this section, we describe the method which finds the optimal rectangular partition of a histogram. The approach that best describes how this method works, is based on *dynamic programming*.

There are two different ways in which dynamic programming could be implemented:

1. **Top-down approach**. In this variant, there is a global structure where all subproblems already solved and their solutions are memorized. To solve a new problem, we split it into all possible subproblems and check if a solution of these subproblems exists or not. In the first case, we obtain the solution by accessing the global structure; in the other case, we have to solve the subproblems and store their solutions. After that, we have to combine the partial results in order to obtain the solution of the original problem. This technique is named *memoization* (and not memorization [4]).

2. **Bottom-up approach**. In this case, we define a subproblems graph and start to solve it from the leaves (minimal subproblems) to the root (original problem), in a backward order, using the subproblems solution to build the solution of a bigger one. An intelligent way in the construction of this topological order may lead to a simple processing scheme.

This last approach has the advantage over the memoization scheme that there's no unsuccessful access to the global structure. This is due to the fact that, when we're processing a problem, all the possible subproblems are already solved. On the other hand, finding a topological order may be too complicated; in these cases the first variant is preferable because it's easier to implement. In this work, the second variant has been used to implement the optimal partitioning method. The reasons will be clearer in the section below.

### 2.2.1 Description

The starting point is represented by the definition of how a subproblem is made. Given $n$, the number of the original vertices that completely describe a histogram, a *minimal subproblem* is defined by five consecutive vertices, which describe a couple of consecutive horizontal edges (e.g., {0,1,2,3,4} and {1,2,3,4,5} define the leftmost minimal subproblems). When defining a sub-histogram,

We have two possibilities to represent two consecutive horizontal edges: we could start and finish with even vertices or odd ones. Sometimes the solutions of these subproblems are the same, sometimes not, so we have to distinguish them like two separate subproblems (see Fig. 2.2). As explained above, we construct the original problem solution using the derived subproblems solutions. To compute subproblems with three consecutive horizontal edges, we recall the previous results: we split them into minimal subproblems putting a horizontal/vertical edge and combining the partial results (see Fig. 2.3). Then we continue until we reach the root of our "topological tree". That explains why we have chosen the second

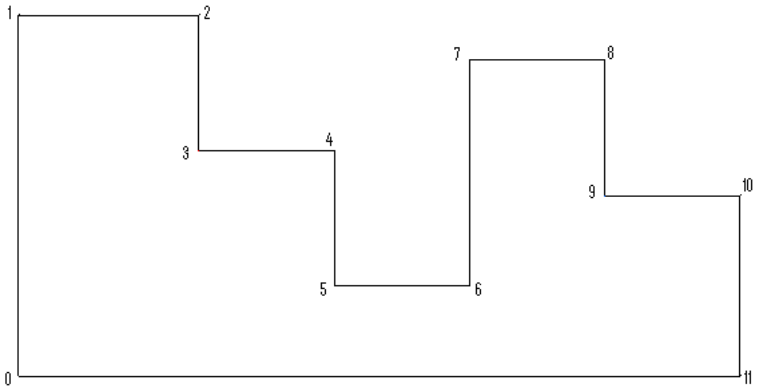
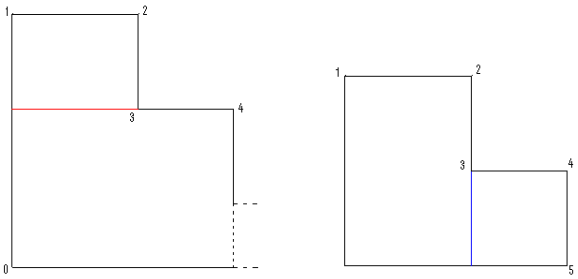



Figure 2.1: Histogram example




(a) *even subproblem* (b) *odd subproblem*

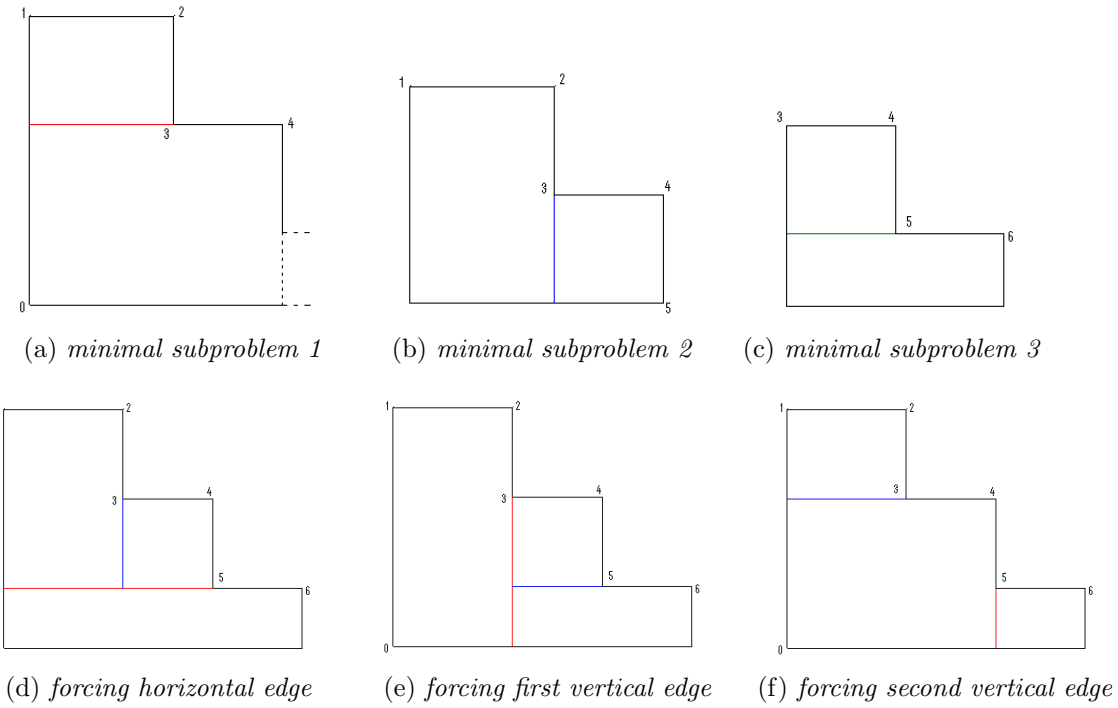
Figure 2.2: Example of distinct subproblems and their solutions associated at the same two horizontal edges

approach of the dynamic programming: to determine the solutions of subproblems of level $i$, we use all the results obtained by sub-subproblems of level $i+1$; in this way there is no access to the global structure without retrieving the solution from a smaller subproblem (see 2.2.1 details of implementation).

When we solve a subproblem, it suffices to consider two cases:

1. we don't insert any additional vertical edge in the partition; in this case we put the highest possible horizontal edge which cuts off the base of the current subproblem;

2. we insert a vertical edge connecting the base to a vertex. In that case we put either the leftmost or the rightmost such vertical edge which is in the optimal partition. In this way we ensure that in one of the resulting subproblems, if the corresponding base endpoint is not an original vertex of the input



(a) *minimal subproblem 1* (b) *minimal subproblem 2* (c) *minimal subproblem 3*

(d) *forcing horizontal edge* (e) *forcing first vertical edge* (f) *forcing second vertical edge*

Figure 2.3: Example of computation of 3-edges subhistogram using previous result

histogram, then there is no other vertical edge there in the optimal partition reaching the base, and hence the corresponding part which includes the base can be cut off as in case (1) above, by the highest possible horizontal edge.

In this way we can also ensure that in the other subproblem, the remaining part of the base is still "anchored" at the lowest original vertex of the original histogram which belongs to the subsequence.

It suffices to consider a quadratic number of subproblems, because we don't have to consider, for each subproblem, all possible places where the base could be. We must store solutions for the case when the base of the subproblem is at a lowest original vertex (i.e., of the original input histogram) at one end of the associated subsequence of the perimeter.

When defining a sub-histogram with a couple of even or odd vertices, we could find a histogram whose base is composed by a couple of non-original vertices. In this case we have a degenerated histogram and we solve it as in the first case (see Fig. 2.3.c).

In the case both cases find a solution with equal costs, we gave major priority to the insertion of a vertical segment.

Listing 2.1: Optimal Method

```
for i=1 to numEdges-1 {
    for j=0 to numEdges-i-1 {
        //even case
        start = 2*j
        stop = 2*(j+i+1)
        horiz_sol = put_horizontal_segment(start,stop)
        vert_sol = put_vertical_segment(start,stop)
        memorize_best(horiz_sol,vert_sol)
        //odd case
        start = 2*j+1;
        stop = 2*(j+i+1)+1;
        horiz_sol = put_horizontal_segment(start,stop)
        vert_sol = put_vertical_segment(start,stop)
        memorize_best(horiz_sol,vert_sol)
    }
}
```

### Details of the implementation

As usual, when dynamic programming is used, the main idea has been to store solutions on a matrix in which the $(i, j)$-element contains the solution of the *sub-histogram* induced by vertices in the interval $[i, j]$, in terms of segments inserted, total partition length and indexes of the associated subproblems. Thereby we use only the upper triangular part of the matrix. Additionally, since we compute only sub-histograms induced by even(odd) vertices, a huge amount of space remains unused (about 3/4 of the matrix). For this reason I've used a linear array to simulate the matrix cells, adding a second array to handle the different offsets between two neighboring cells in the matrix but distant in the array (see 2.2.2 space complexity).

During the tests to check the correctness and to evaluate the performance of this implementation, I realized that in the symmetric case (section 3.3), the algorithm didn't work as we expected: it took too much time to solve a histogram like that. The cause was as simple as hidden in the implementation itself: when we solve a sub-histogram, there are at least two ways to partition it (i.e. see Fig. 2.2 different solution for *minimal subproblems*), but only the best one is memorized in the global structure. Therefore, the following situation may arise: assume (wlog) we have a problem with the lowest original vertex on the left(right) and the best choice is to put vertical edges for the subproblems associated with the current

one. If we are putting a vertical edge to partition it, then we have to search if the solution of the right(left) induced subproblem has a horizontal edge in the global structure. When we look up the structure and we don't find a horizontal but a vertical edge, we have to rebuild the solution for that subproblem with the constraint of putting a horizontal edge on the right(left) of the currently introduced vertical edge. An example is shown in fig 2.4: both subproblems presented one vertical edge as solution (Fig. 2.4.b and .c); when in the bigger problem (Fig. 2.4.a) we try to put the leftmost vertical edge, we don't find a horizontal edge in the associated subproblems so we have to recompute an already solved case forcing the insertion of such an edge (Fig. 2.4.d).

The time-computation increase is an obvious consequence; another time the trade-off between time and space comes out, we could save computation time by adding a structure, similar to the global one and with the same dimensions, that stores only "horizontal solutions", those which force the inclusion of a first horizontal segment (which however are calculated in the search). A further example in which this trade-off appears is when we want to put a new segment to the partition:

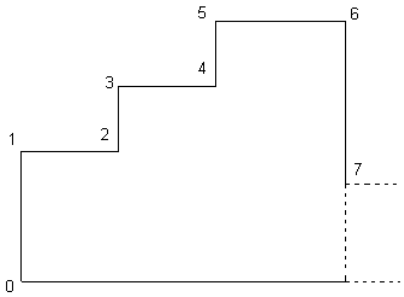
- *horizontal segment*. We have to search whether the highest horizontal edge which cuts off the base crosses or not more than one original horizontal edges.

- *vertical segment*. We have to search whether in the subhistogram induced by the pair of original vertices $(start, stop)$, there exists one or more original vertices smaller or equal than the *start* (or *stop*) vertex in case *start* is even (or odd). In that case we are constrained to put the highest possible horizontal segment.

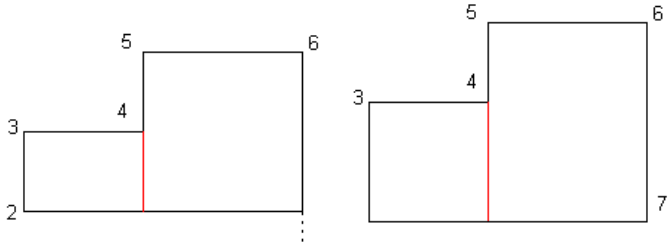
In the first implementation, both of these cases required a linear-time scan over the vertices in the range $[start, stop]$ every time we were solving a (sub)histogram; in the second one I've introduced two arrays similar to the global structure to store this information before the start of computation. In this way, when I need, I can retrieve them with an easy access to the related structure. To compute and store this useful information, I've used the second dynamic programming approach, as for the partitioning.

With the aim to have a simpler managing and clearer source code, I've divided the global structure in three separated sub-structures:

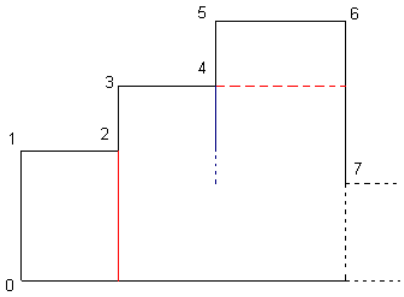
- *lengths* . This structure stores only the total partition length of the associated subproblem. So, when we're trying to solve the current problem, we add the length of the current vertical/horizontal segment to the values related to the lengths of the induced subproblems;




(a) *current subhistogram*




(b) *even subproblem associated* (c) *even subproblem associated*




(d) *real solution vs previous result*

Figure 2.4: Example of re-computation of already solved sub-problem

- *segments*. It memorizes only new segments inserted in the solution of the associated (sub)histogram, since the segments of the associated subproblems are stored in their related cells. If the solution is to insert a vertical edge, it stores only one segment, else it saves one or more edges (depending on how many original horizontal edges it passes through);

- *previous*. It represents the link between current (sub)histogram and its related subhistograms, it memorizes the indices of associated subproblems in such a way that, when we want to build the current solution, we use this

Listing 2.2: recursive procedure to build solution

```
function getSegments (segments, previous, solution) {
for i=0 to segments.size()
   solution.push(segments[i]);
for i=0 to previous.size() {
   if (previous[i] != -1)
      getSegments(segments[previous[i]],
                  previous[previous[i]],solution);
}
```

structure to create a *recursive* path.

The last two structures are a bit different from the first one, because the cells of these are not single cells but variable size arrays. The size of a cell depends on the number of new introduced segments (*segments* structure) and the number of related subproblems (*previous* structure).

Assuming original vertices are numbered in clockwise order starting from lower left vertex to lower right vertex (the base vertices are the endpoints), the final best solution is stored in the cell associated to $(0, n-2)$-problem or $(1, n-1)$-problem. These cells store the same solution because the define the entire original histogram, to decrease time computation we can decide to compute only one. In order to build the segments set which partitions the histogram into rectangles, I've used a recursive procedure similar to the procedure 2.2.

A possible recurrence equation could be described as:

$$S(i,j) = \begin{cases} \sum_{k=1}^{e'} s_k + \sum_{w=1}^{m'} S(i_w, j_w) & \text{if } j \geq i+6, \\ \\ s & \text{if } j = i+4 \end{cases} \tag{2.1}$$

in which the terms are:

- $S(i,j)$ is the set of all segments which divide the histogram (induced by the couple of vertices $(i,j)$) into rectangles;
- $s$ is the vertical/horizontal segment inserted in a minimal subproblem (defined by two consecutive horizontal edges of the histogram;
- $e'$ is the number of new introduced segments in the (sub)histogram induced by $(i,j)$;

- $m'$ is the number of resulting subproblems from the current one and whose indices are stored in the $(i, j)$-element in *previous* structure.

Before writing the recurrence relation for *length* structure, we need other definitions. When we are trying to put a horizontal edge:

$$H(i,j) = \sum_{k=1}^{m'} l_k + L(i_k, j_k) \tag{2.2}$$

where $l_k$ is the length of the $k$-th horizontal edge we introduce (if there are more than one horizontal edges) and $L(i_k, j_k)$ is the associated subhistogram partition length. Instead when we want to find the best vertical edge:

$$V(i,j) = min\{[L(k,k) + L(i,k) + L(k,j)], \forall k \in \{i+2, i+4, \ldots, j-2\}\} \tag{2.3}$$

where $L(k,k)$ is the length of the vertical segment (linking the vertex $k$ to the base) we want to put, $L(i,k)$ indicates partition length of the left subhistogram and $L(k,j)$ the partition length of the right one. Now we are able to define the relation for *length* structure as:

$$L(i,j) = min(H(i,j), V(i,j)) \quad j \geq i+4 \tag{2.4}$$

### 2.2.2 Complexity

#### Time

As reported above, the time complexity of the optimal algorithm is $O(n^3)$. In this section I'll show the reason because we expect a cubic complexity.

With a simple and fast analysis, we could note that the sum of the total number of subproblems we solve is quadratic, and, for each of them, we spend linear time (up to a multiplicative constant due to accesses to the structure and comparisons between partial solutions).

With a deeper study we can compute a more precise upper bound. Given $n$, the number of the original vertices, with denote with $e$ the number of horizontal edges excluding the base of the histogram.

$$e = \frac{n-2}{2} \tag{2.5}$$

For each (sub-)histogram with $i$ horizontal edges we have to evaluate $i$ associated situations:

$$\text{problem of size i} = \begin{cases} 1 & \text{we insert a horizontal edge} \\ i-1 & \text{we insert a vertical edge} \end{cases} \tag{2.6}$$

and the number of sub-histograms with $i$ consecutive horizontal edges are:

$$\#\text{ subhistograms of size i} = (e - i + 1) \tag{2.7}$$

so the total amount of time can be evaluated as:

$$2\sum_{i=2}^{e}(e - 1 + 1)i = ... = \frac{e^3 + 3e^2 - 4e}{3} \quad \in O(e^3) = O(n^3) \tag{2.8}$$

This summation is multiplied by two because we have to keep separate the cases in which we have a sub-histogram defined by a couple of even or odd vertices.

### Space

As described above, using the simplest idea, we can imagine to store all information we need in a $n \times n$ matrix, using solely a half of the upper triangular part (about 1/4 of the entire matrix with $n \to \infty$ see Fig. 2.5). This is the reason why

| i \ j | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|---|
| 0 | ■ | | ■ | | ■ | | ■ | |
| 1 | | ■ | | ■ | | ■ | | ■ |
| 2 | | | ■ | | ■ | | ■ | |
| 3 | | | | ■ | | ■ | | ■ |
| 4 | | | | | ■ | | ■ | |
| 5 | | | | | | ■ | | ■ |
| 6 | | | | | | | ■ | |
| 7 | | | | | | | | ■ |

Figure 2.5: Example of really used space (black) in a 8-vertices histogram.

I've decided to compact this table in a linear array with the addition of a second array to manage the different offset between two neighboring cells in the matrix. We can count the number of effective cells we need with the equation below:

$$\#\text{ cells} = 2\sum_{i=0}^{\frac{n}{2}}(\frac{n}{2} - i) = \frac{n^2 + 2n}{4} \tag{2.9}$$

The offset array stores the number of cells we have to "jump" to position to the first sub-problem with a fixed start-point.

For example if we want to retrieve information about the sub-histogram induced by the couple $[start, stop]$ we have to access the element with index defined by:

$$\text{offset}[start] + \frac{stop - start}{2} \tag{2.10}$$

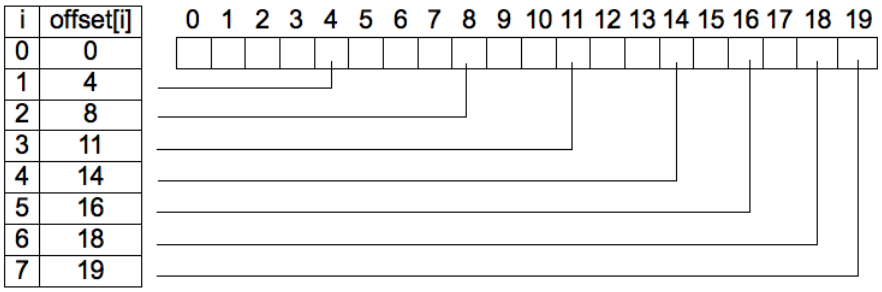



Figure 2.6: Example of the use of the "offset array".

In the second version, the auxiliary structures to memorize the "horizontal" solution of a subhistogram and the information to avoid linear scanning during the computation of horizontal and vertical edges, use the same amount of space as the global structure. We continue to have a space complexity $\in O(n^2)$ but with a large saving of computation time.

## 2.3 Thickest first method

As mentioned above, the problem of optimally partitioning polygons with holes into rectangles is NP-hard [10]. For this class of problems it is difficult to find a reasonable polynomial-time algorithm that obtains an exact solution, but it's possible to find a *near-optimal* solution in acceptable time. This type of algorithms is called *approximation algorithms* and the associated solutions are considered good enough. Generally, this paradigm exploits a trade-off between the solution quality (optimality) and runtime (tractability).

Although I treat a special case of hole-free polygons and an optimal method exists, I've implemented a variant of the "thickest first" algorithm, the *"locally q-thickest first"* algorithm, studying its behavior in histograms, while changing a parameter value $q$.

### 2.3.1 Description

In order to present this technique, we need some definitions. We say that a rectangle $R$ is *thicker* than a rectangle $R'$ iff the shortest side of $R$ is longer than the shortest side of $R'$. A rectangle within $P$ is called *maximal* iff there is no other rectangle in $P$ that properly includes it. Then we can say it's the *thickest* rectangle in $P$ iff there isn't any thicker rectangle within $P$. $R$ is a *locally-thickest*

Listing 2.3: Thickest-first Method

```
if P is a rectangle
   finished
else {
   find the thickest maximal rectangle R
   draw R
   continue with the sub-histograms induced by R
}
```

rectangle iff it's a maximal rectangle in $P$ and there is no thicker rectangle in P which overlaps with it.

Given $S$, a set of segments, we affirm that $S$ is a *thickest-first* rectangular partition of $P$ iff is composed by disjoint segments, lying inside $P$ and not overlapping with the boundary of $P$, except their endpoints and the rectangles that partition $P$ can be determinated by the procedure 2.3.

In this work, I've studied a variant of this algorithm in which $q$-thickest rectangles are considered: a rectangle is called $q$-rectangle if the ratio between the length of its horizontal side and the length of its vertical side is equal to $q$. The only difference between $q$-thickest algorithm and the original one is that we search $q$-thickest rectangles instead of thickest. For pratical purpose, there's a linear pre and post-processing of $y$-coordinates of original vertices and segments obtained from partitioning, in which they are multiplied by $q$. In [9] it is shown that any $\sqrt{2}$-*thickest* rectangular partition is of length $< (1 + \sqrt{2}) \times M(P)$, where *M(P)* indicates the optimal partition length.

### Details of the implementation

Although a linear time procedure to find a thickest-first partition exists, I've developed another algorithm which uses simple structures at the expense of performance degradation: from $O(n)$ to $O(n \log n)$. The main reasons is that we want to study the goodness of the solution of these method in relation with the optimal algorithm.

In order to compute this technique, I need some auxiliary structure in which I can store useful information about the original histogram: for example, how long a horizontal edge could be extended to the left and to the right. The reason is that for every horizontal edge I have to find the shortest side of the associated rectangle, and compare it with the shortest sides of all rectangles associated to

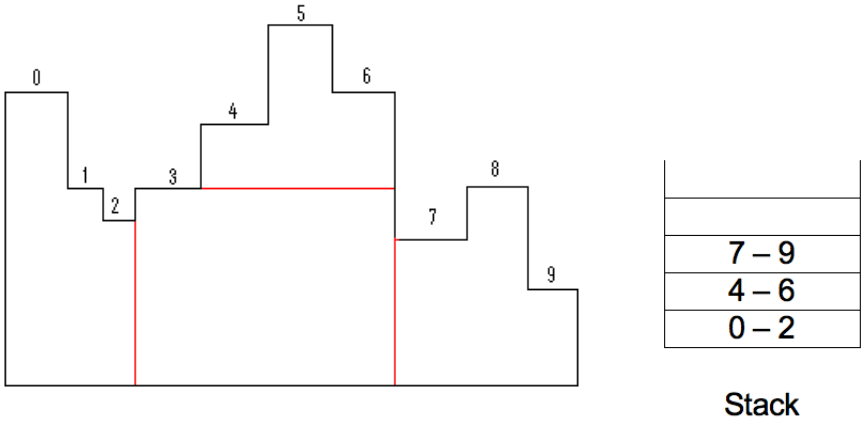
Listing 2.4: Find edge extension procedure

```
for i=0 to numEdges
   left_ext[i] = right_ext[i] = i
push (first_edge)
for i=1 to numEdges {
   if (edge[i].height > top_edge.height) //taller
      push(i)
   else {
      while (edge[i].height  <= top_edge.height) {
         left_ext[i] = left_ext[top()]
         if (edge[i].height < top_edge.height) {
            right_ext[top()] = i-1
            pop()
         }
        else
           right_ext[top()] = i
      }
   }
}
for each element in stack
   update left, right extension
for each edge
   compute length extension
```

other edges. With the help of a stack, I get these extensions and their related lengths with a linear scanning (procedure 2.4). After that, I can afford to start the locally-thickest method which is similar to the procedure 2.3 with some small changes: when the locally thickest rectangle is identified I have to modify the information about edge-extensions (2.5).

Listing 2.5: Locally q-thickest procedure

```
push ([0,numEdges-1])  //total histogram
while (stack != empty) {
   [start,stop] = top()
   pop()
   find q-thickest rectangle in the histogram
      defined by [start,stop]
   store partitioning segments (with original y-value)
   update edge-extensions
   for each induced sub-histogram
      push([start,stop])
      //start and stop are the original numbers of the
      //first and last edge in the subhistogram
}
```




(a) *Thickest rectangle*

(b) *subhistograms to compute*

Figure 2.7: Example of first steps in thickest partitioning

As it's possible to see in the procedure 2.5 and Fig. 2.7, there are some interesting points to specify:

- we number the original horizontal edges (excluding the base) from left (number 0) to right (number $(numEdges - 1)$);
- we define a (sub-)histogram with the leftmost horizontal edge (*start*) and the rightmost one (*stop*);
- after we've found the *q-thickest* rectangle we insert related segments with original y-coordinates. So we avoid the final post-processing phase, in which we should divide by $q$ all the y-coordinates of the segments obtained during the computation.

In the following section we'll analyze the time and space complexity.

### 2.3.2 Complexity

#### Time

Contrary to the optimal case, with this technique we have to asymptotically separate the average case from the worst case: in the average case we obtain a $O(n \log n)$ complexity, in the worst case it becomes $O(n^2)$, and it fluctuates depending on the shape of the histogram to partition.

Excluding the case where there are several horizontal edges with the same $y$-coordinate, one of the best cases happens when the thickest rectangle partitions

the histogram into pieces such that the largest piece has about one third of the vertices. Therefore, we consider the following recurrence relation:

$$T(e) = \begin{cases} 3T(e/3) + \Theta(e) & \text{if } e > 3, \\ \\ \Theta(1) & \text{if } e \leq 3 \end{cases} \tag{2.11}$$

where $e$ is defined as the number of horizontal edges in the original histogram (see eq. 2.5). With a simple analysis:

1. at level $i$ we have $3^i$ nodes each of them with work $\Theta(e/3^i)$;
2. the leaves of the tree are at level $\log_3 e - 1$.

So if we ignore constant factors, then the total time-complexity would be:

$$\sum_{i=0}^{\log_3 e - 1} 3^i \left(\frac{e}{3^i}\right) = e \log_3 e = \Theta(n \log n) \tag{2.12}$$

In order to calculate more precisely the time complexity, we would have to take into account that in many cases one or two of the resulting subproblems may often have more than one third of the horizontal edges. However, to obtain an $O(n \log n)$ time upper bound, it would also suffice if the expected number of vertices of the largest subpolygon, in the partition induced by the thickest rectangle, is smaller by some constant factor, since this would suffice to show that the expected depth of the recursion tree becomes $O(\log n)$.

Instead, in the worst case, this method deteriorates, because it doesn't divide the original histogram into two-three subhistograms of the same size (approximately) like in the average case. At each step, it is able to "eliminate" the leftmost edges, so the resulting subhistogram has only one or two horizontal edges less than the current one, and, to find the locally thickest rectangle, we consume a linear time scanning. This is the reason the worst case become $\in O(n^2)$. A possible recurrence equation that could describe this situation is:

$$T(e) = \begin{cases} T(e-2) + \Theta(e) & \text{if } e > 2, \\ \\ \Theta(1) & \text{if } e \leq 2 \end{cases} \tag{2.13}$$

### Space

Regarding the implementation space complexity, there's no difference between average and worst case. I've used a linear array to store the y-coordinates of the original vertices multiplied by $q$ ($\Theta(n)$) and two linear arrays to memorize the edge

extensions and their lengths ($\Theta(n)$). During the computation these last arrays are updated according to the locally thickest rectangle identified. In addition to this structure there is a stack which stores the endpoints (start and stop edges) of the next subhistograms to process. In the average case, this stack stores at most $O(2\log_3 e - 1)$ elements simultaneously.

## 2.4 Extending the result to hole-free polygons

In this section it's shown how to extend the previous techniques to partition hole-free polygons. Taken $P$ and $e$, respectively the hole-free polygon and an edge of it, we defined *HIST(P,e)* the set of segments partitioning P we can compute *HIST(P,e)* with a recursive approach as in procedure 2.6.

Listing 2.6: HIST(P,e) procedure

$H$ =maximal histogram in P with base e
**if** $H = P$
    $HIST(P, e) = \emptyset$
**else** {
    $P_1 \dots P_k$ resulting subpolygons with $P_1 = H$
    $s_i$ segments where $P_i$ touches $P_1$ with $2 \leq i \leq k$
    HIST(P, e) = $\bigcup_{2 \leq i \leq k} \left( \{s_i\} \cup HIST(P_i, s_i) \right)$
}

We introduce some notation:

- a vertex is called *concave* when it's the corner of an interior angle of 270°;
- a vertex is called *flat* when it's the corner of an interior angle of 180°;
- a vertex is called *convex* when it's the corner of an interior angle of 90°;
- $P$ is in *BAS* normal form if, $\forall z$ on the boundary of $P$, non positioned at a corner, there's a flat vertex at $z$ iff an isothetic segment within P links $z$ to some concave vertex of $P$;
- if $P$ is in *BAS*-form, and $v_1$ and $v_2$ are two distinct vertices of $P$, then we call $v_1$ the *left neighbor* of $v_2$ iff they can be linked by a horizontal segment lying entirely in $P$ without passing through any other vertices of $P$.

When a polygon is in its *BAS* normal form, we can use a simple structure, called *BAS(P)*, to traverse it in an easy manner. We use a double-linked structure so that every vertex of $P$ (including flat vertices) has a pointer to all its neighbors. There is a method to compute this structure in linear time ([1] and [3]). Now we are able to compute *HIST(P,e)*, using this *BAS(P)* structure, in the same recursive way it's defined: given two neighbouring vertices $v_i$ and $v_j$ we can compute the maximal histogram in any direction with a simple linear scanning of the original vertices which lie on the histogram boundary. These vertices are traversed in a clockwise order beginning on the second vertex of $[v_i, v_j]$.

When determining the next vertices to traverse with the help of the *BAS(P)* structure, it should happen that the segments which link two neighbouring vertices $v_k$ and $v_l$ is a part of $H$ but disjoint from $P$. In this case the segment is inserted into the set of segments to be output and pushed on the stack to later compute the maximal histogram with base $[v_k, v_l]$ with the appropriate direction.

It's clear that at most two histograms can share the same vertex, so the total time is $O(n)$.

# Chapter 3

# Experimental results

In this section, we present the results of applying both algorithms presented above, on different kinds of histograms, which can be grouped in three types:

1. general histograms;
2. staircase histograms;
3. symmetric histograms.

The algorithms and the functions that generate random instances of the three histogram families are been implemented in *C++*. Test results, which we obtained with our implementation on histogram of different sizes (number of vertices), were carried on a Intel Pentium Core2Quad 2.6/1.3GHz with 2GB of main memory running Mandriva Linux and GNU gcc 4.3.2.

All the results like execution times, number of subproblems or partition lengths, are average of many runs on randomly generated histograms. For all three types of histograms, we started analyzing the $q$ parameter within the $q$-thickest first method. When we were doing tests on histograms with a fixed number of vertices, we have counted how many times a q-value was first, second or third, sorting all possible values in ascending order on their partition length. The range of possible values of $q$ was in $[0.5, 3]$ with $step = 0.1$, while the number of histogram vertices start from 200 and stop at 6000 with a $step = 200$. We found three best values depending on the shape of the histogram, these are $1, 1.2$ and $1.5(1.6)$.

After that, we have compared the optimal method with the $q$-thickest one, with only the best values written above. As terms of comparison, we have used partition lengths and time of computation (this last one was useful only with the optimal procedure since for thickest method the time was almost negligible).

Then, we've also confronted the behavior of the optimal method, in terms of subproblems solved and time computation on the different types of histograms.

| q | 0,5 | 0,6 | 0,7 | 0,8 | 0,9 | 1 | 1,1 | 1,2 | 1,3 | 1,4 | 1,5 | 1,6 | 1,7 | 1,8 | 1,9 | 2 | 2,1 | 2,2 | 2,3 | 2,4 | 2,5 | 2,6 | 2,7 | 2,8 | 2,9 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0 | 0 | 0 | 1 | 0 | 1 | 3 | 3 | 17 | 24 | 50 | 55 | 21 | 49 | 52 | 11 | 2 | 7 | 4 | 4 | 0 | 1 | 0 | 0 | 0 | 0 |
| **2** | 0 | 0 | 0 | 0 | 1 | 0 | 3 | 5 | 2 | 16 | 26 | 43 | 45 | 33 | 59 | 41 | 12 | 7 | 6 | 6 | 4 | 1 | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 2 | 2 | 5 | 15 | 23 | 34 | 33 | 60 | 50 | 35 | 20 | 8 | 12 | 4 | 2 | 2 | 1 | 1 | 0 |

Table 3.1: q results general histograms - type 1.

## 3.1 Random shape histograms

These are common histograms that people imagine when thinking about a histogram, they have no recurring patterns as the types below, they are made by frequent "latch" of various lengths. We have created three different types of these histograms, based on possible features:

1. same fixed maximum length of vertical and horizontal edges;
2. fixed maximum length to horizontal edges and maximum length equal to base length to vertical edges;
3. "uniform" choice in the range $(0, n]$ of the $x$- and $y$- coordinates of vertices.

In order to have maximum uniformity in the choice of the lengths of the edges, we used the *rand()* method, provided by *C/C++* standard library, initialized with a seed linked with time and date of the day, in a manner that we don't have the same generation on two distinct tests.

### 3.1.1 Fixed edge lengths

The procedure to build these histograms can be represent as we have done in procedure 3.1.

#### *q*-study

For histograms produced in this manner, the best values of $q$ were 1.6 and 1.9 as we can see in Fig. 3.1 and in table 3.1.

The reason because we have those values is simple. Histograms with this shape present a feature visible to the naked eye: they have a huge difference between the long base and maximal height. So, when imposing a value of $q > 1$, we stretch the histogram vertically, changing (probably) the first thickest rectangle and, as a consequence, all the associated subproblems.

Looking at the figure and the table, we don't see a perfect symmetry centered on best values. It depends exclusively on the randomly generated histograms and on the score rules for the best $q$ values for a simulation. For example, $q = 1.7$ was

Listing 3.1: procedure.

```
for (int i=1; i<numvertices-1; i++) {
    //put an horizontal edge
    if (horiz) {
        x += (rand() % max) + 1;
    }
    //put a vertical edge
    else {
        currY = (rand() % max) + 1;
        //Deciding to increase / decrease the height
        if ((rand()%2) == 0)
            currY = -currY;
        while (y+currY<=0) {
            currY = (rand() % max) + 1;
            if ((rand()%2) == 0)
                currY = -currY;
        }
        y += currY;
    }
    //save vertex(x,y);
    x_coor[i] = x;
    y_coor[i] = y;
}
```

probably among the best values, but not often among the best three, so its score has become too low.

### Comparison between optimal and q-thickest

We've compared optimal method and thickest one using, for the second, the value {1,1.2,1.6}. We use 1.6 since it's the value with more "first positions".

In picture 3.2.a, we can see the behaviors of optimal and thickest partition lengths in relation to the length of variable size histograms: when optimal method worsens (the ration between partition and histogram length increases), even the thickest one worsens, with all possible usable values. Then, in figure 3.2.b, it's shown that length differences between optimal and thickest methods are not defined by a constant factor but they are "floating", due to algorithm's locally decisions and specific histograms shape. Interesting is the case of histograms with 3600 vertices, for those the average partition length, produced by thickest algo-

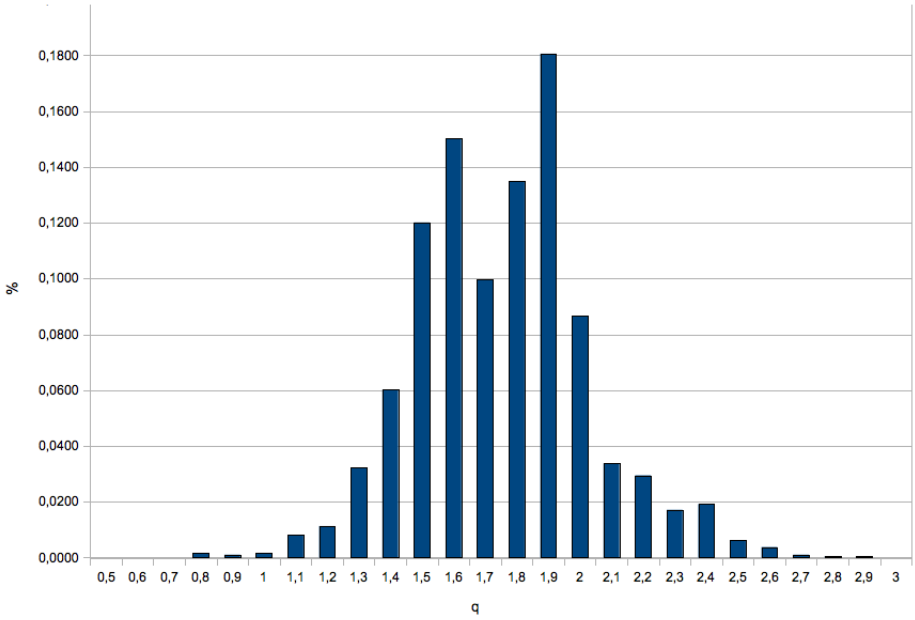
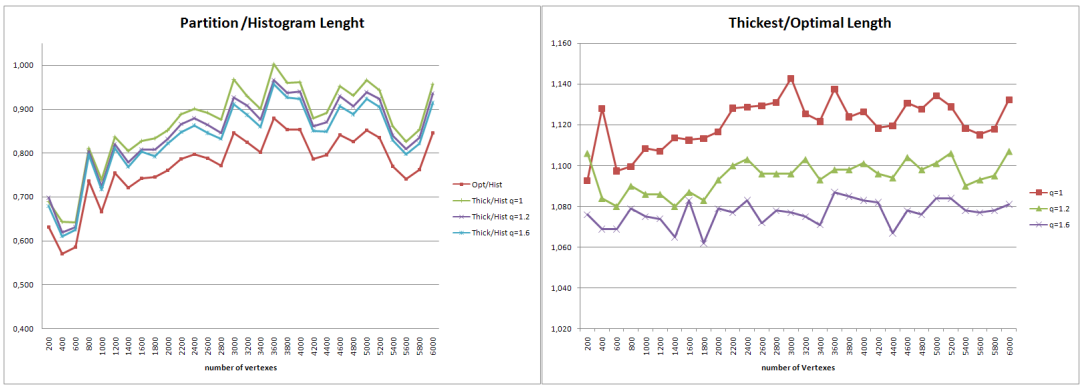



Figure 3.1: q results general histograms - type 1.




(a) *Ratio between obtained partitions and histograms lengths*

(b) *Ratio between q-thickest and optimal partition lengths*

Figure 3.2: comparison optimal-thickest method on general histograms - type 1.

rithm with $q = 1$, is nearly the perimeter of histograms.

### 3.1.2 Semi-fixed lengths

With the aim to balancing the height of a histogram with its base, we considered histograms defined as in the following procedure:

Listing 3.2: procedure.

```
//add lowest left vertex
x_coor[0] = 0.0;
y_coor[0] = 0.0;
//fixing x-coordinates
for (int i=1; i<=numEdges; i++) {
    old_x = x;
    x += rand() % maxOriz + 1;
    x_coor[2*i] = x;
    x_coor[2*i-1] = old_x;
}
//fixing y-coordinates
maxVert  = x;
for (int i=1; i<=numEdges; i++) {
    currY = rand() % maxVert + 1;
    if (rand()%2 == 0)
        currY = -currY;
    while (y+currY<=0 || y+currY>maxVert) {
        currY = rand() % maxVert + 1;
        if (rand()%2 == 0)
          currY = -currY;
    }
    y += currY;
    y_coor[2*i] = y_coor[2*i-1] = y;
}
//add lowest right vertex
x_coor[numvertices-1] = x;
y_coor[numvertices-1] = 0.0;
```

An important feature of histograms produced as described above, is that they could present a lot of 'peaks" which influence the decisions rules and results of both algorithms. Indeed, as we will see in following sections, both techniques are forced to put horizontal edges and cut off those peaks.

| q | 0,5 | 0,6 | 0,7 | 0,8 | 0,9 | 1 | 1,1 | 1,2 | 1,3 | 1,4 | 1,5 | 1,6 | 1,7 | 1,8 | 1,9 | 2 | 2,1 | 2,2 | 2,3 | 2,4 | 2,5 | 2,6 | 2,7 | 2,8 | 2,9 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0 | 0 | 1 | 3 | 15 | 51 | 62 | 60 | 44 | 31 | 23 | 10 | 4 | 1 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **2** | 0 | 0 | 0 | 0 | 8 | 14 | 49 | 60 | 64 | 38 | 34 | 20 | 20 | 6 | 1 | 1 | 2 | 1 | 1 | 1 | 1 | 1 | 1 | 1 | 0 | 0 |
| **3** | 0 | 0 | 0 | 0 | 0 | 10 | 15 | 35 | 44 | 64 | 41 | 43 | 32 | 13 | 6 | 3 | 3 | 3 | 3 | 1 | 0 | 1 | 0 | 0 | 1 | 1 |

Table 3.2: q results general histograms - type 2.

### *q*-study

From Fig. 3.3 and table 3.2, we can affirm the best value of $q$, for these histograms, it's 1.2.

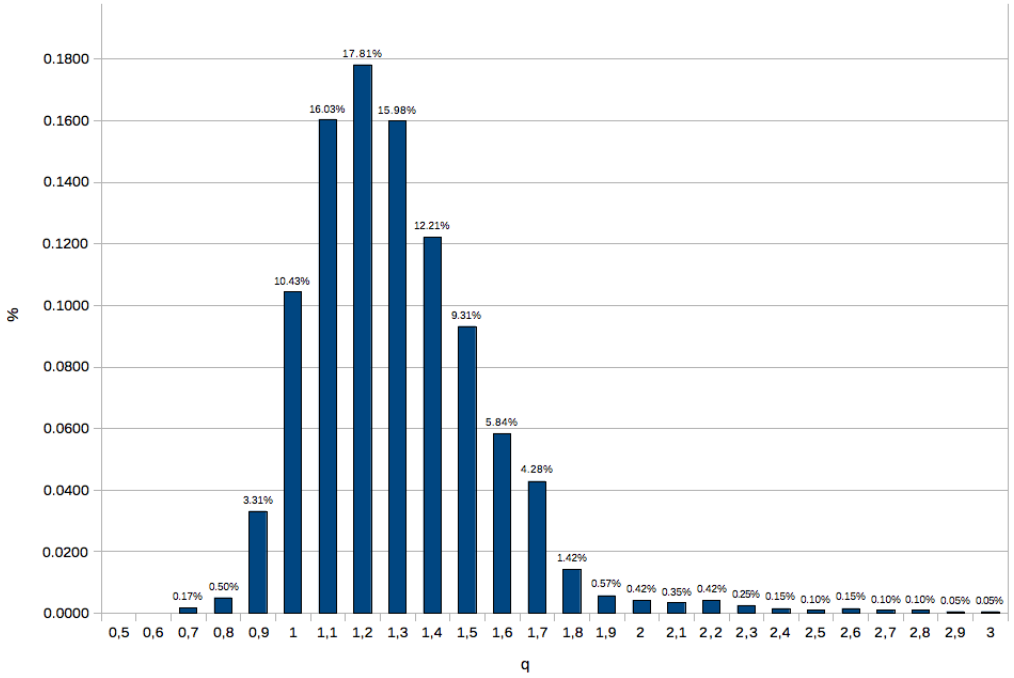
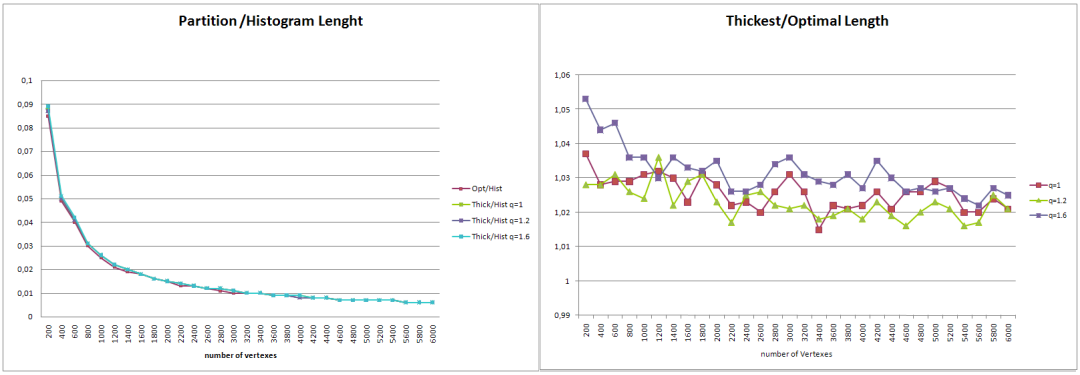



Figure 3.3: q results general histograms - type 2.

In this case we can note a semi-perfect simmetry centered on best value of $q$.

### Comparison between optimal and q-thickest

To compare optimal and thickest algorithms, we've used the values {1,1.2,1.6}. Differently from previous case, from results illustrated in fig 3.4.a, we see the decrease of the value of ratio between partition and histograms length (both techniques) with the increase of the number of vertices. This is due to the presence of the "peaks" which force horizontals segments insertion that cut off them. Obviously, the lengths of these segments are much shorter than the length of the




(a) *Ratio between obtained partitions and histograms lengths*

(b) *Ratio between q-thickest and optimal partition lengths*

Figure 3.4: comparison optimal-thickest method on general histograms - type 2.

perimeter of the associated peak. From both figure (3.4.a and .b), we can say there are no difference between optimal and thickest (with different values of $q$) algorithms: the length of obtained partitions are almost the same, we have a multiplication factor of about 1.03 from length of optimal solution to length of thickest one.

### 3.1.3 "Uniform" lengths

In the above sections, we've built general histograms fixing some maximal lengths: in type 1 we've imposed the same maximal length as for vertical as for horizontal edges; instead, in type 2 we've enforced a fixed maximal length for vertical edges and a variable length, in relation with total length of the base, for horizontal edges. Now, we are trying to not forcing a specific length for both kind of edges.

With the term "uniform" we mean that we fix only possible maximal length for the base and the height of histograms (otherwise random generator hasn't a real limit for extractions), then we'll extract the values of $x$- and $y$-coordinates in the range $[0, maxValue]$, with the attention to avoid possible copies of same $x$ value. We also don't want the presence of peaks, or we want to limit them, so we've used a "probability" function: $y$ values are defined as $numvertices/rand()$, where *numvertices* is used as maximal value for base and height of the histogram.

Listing 3.3: procedure.

```
//add left lowest vertices
x_coor[0] = 0.0;
y_coor[0] = 0.0;
x_coor[1] = 0.0;
y_coor[numvertices-1] = 0.0;
//fixing x-coordinates
vector coorVector;
while (coorVector.size()<numEdges) {
    for i=coorVector.size() to numEdges {
        coorVector.push_back(rand());
    }
    sort(coorVector);
    eliminate_copies(coorVector);
}
//memorize x-coordinates
for i=1 to numEdges
    x_coor[2*i] = x_coor[2*i+1] = coorVector.at(i-1);
coorVector.clear();
//fixing y-coordinates
while (coorVector.size()<numEdges) {
    for i=coorVector.size() to numEdges
        coorVector.push_back(numvertices/rand());
}
//memorize y-coordinates
for i=1 to numEdges
    y_coor[2*i] = y_coor[2*i-1] = coorVector.at(i-1);
```

### *q*-study

For histograms with these features, most suitable values of $q$ are 1 and 1.1 (see table 3.3 and Fig. 3.5).

| **q** | 0,5 | 0,6 | 0,7 | 0,8 | 0,9 | 1 | 1,1 | 1,2 | 1,3 | 1,4 | 1,5 | 1,6 | 1,7 | 1,8 | 1,9 | 2 | 2,1 | 2,2 | 2,3 | 2,4 | 2,5 | 2,6 | 2,7 | 2,8 | 2,9 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0 | 0 | 0 | 2 | 51 | 162 | 73 | 11 | 0 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **2** | 0 | 0 | 0 | 0 | 9 | 45 | 163 | 71 | 11 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 0 | 0 | 1 | 40 | 165 | 77 | 14 | 1 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 3.3: q results general histograms - type 3.

The reason since 1 and 1.1 are best values for $q$ is simple: histograms built in this manner are more "balanced", so there isn't the need to modify the ratio

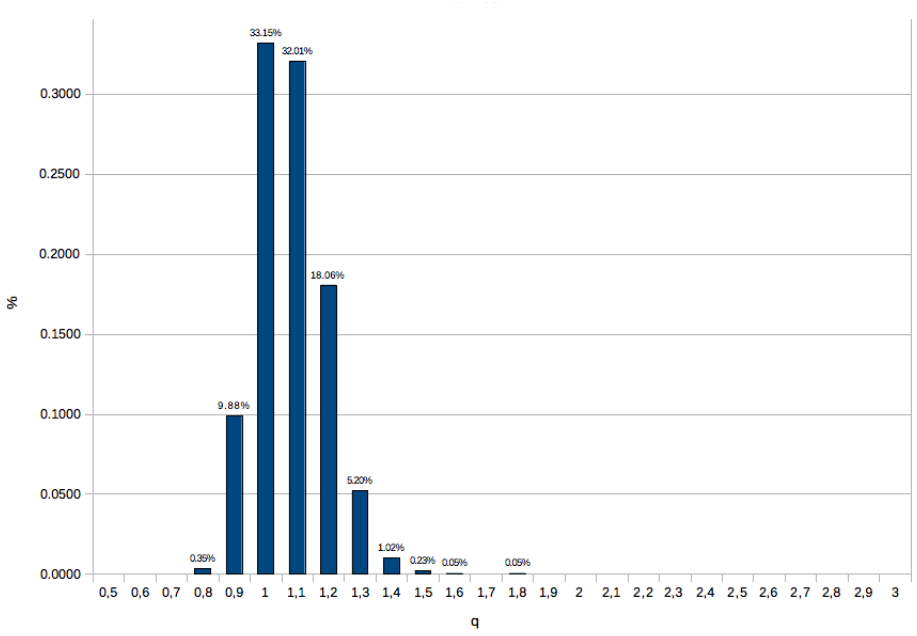



Figure 3.5: q results general histograms - type 3.

between height and extension length of an edge in order to find best thickest rectangle in current (sub)histogram. In this case, as in the previous one, we can see a symmetry centered on best values.

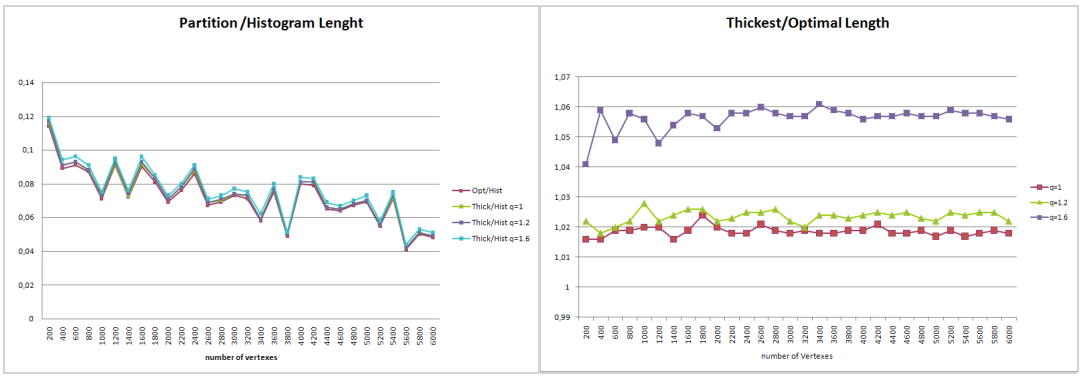
**Comparison between optimal and q-thickest**

In order to compare optimal and thickest algorithms, we've used {1,1.2,1.6} as $q$ values.

As in type-2 histograms, we can see the decrease of the value of ratio between partition and histograms length (both techniques) with the increase of the number of vertices. Another time, the presence of some "peaks" forces the insertion of horizontals segments that cut off them. From both figure (3.6.a and .b), we can affirm thickest algorithm(with $q = 1$ or $q = 1.2$) produces a good approximations of optimal method: the average ratio between optimal length solution and thickest length solution is about 1.02.

## 3.2 Random staircase histograms

In this part, we've studied histograms with a shape similar to a staircase, and, wlog, we've considered ascending staircase from left to right. As we've done for




(a) *Ratio between obtained partitions and histograms lengths*

(b) *Ratio between q-thickest and optimal partition lengths*

Figure 3.6: comparison optimal-thickest method on general histograms - type 3.

general shape histograms, we've defined two types of staircase histograms:

- with same fixed length as for vertical edges as for horizontal edges;
- "uniform" choice in the range $(o, n]$.

### 3.2.1 Fixed edge lengths

For these histograms, we have to modify the procedure 3.1, eliminating the lines where we decide whether to increase or decrease the height of current edge. The other part of the procedure can remain the same.

#### *q*-study

As we observe from figure 3.7 and table 3.4, best values are centered in 1, showing a good symmetry around it. The reason why $q = 1$ results to be the best choice is simple: with a high probability the thickest rectangle is in the "center" of the stair and, multiply $y$-coordinates by $q$, does not change too much the initial situation and its associated subproblems.

| **q** | 0,5 | 0,6 | 0,7 | 0,8 | 0,9 | 1 | 1,1 | 1,2 | 1,3 | 1,4 | 1,5 | 1,6 | 1,7 | 1,8 | 1,9 | 2 | 2,1 | 2,2 | 2,3 | 2,4 | 2,5 | 2,6 | 2,7 | 2,8 | 2,9 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0 | 0 | 0 | 3 | 72 | 148 | 66 | 9 | 2 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **2** | 0 | 0 | 0 | 4 | 20 | 45 | 144 | 76 | 7 | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 0 | 1 | 4 | 38 | 144 | 94 | 11 | 4 | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 3.4: q results staircase histograms - type 1.

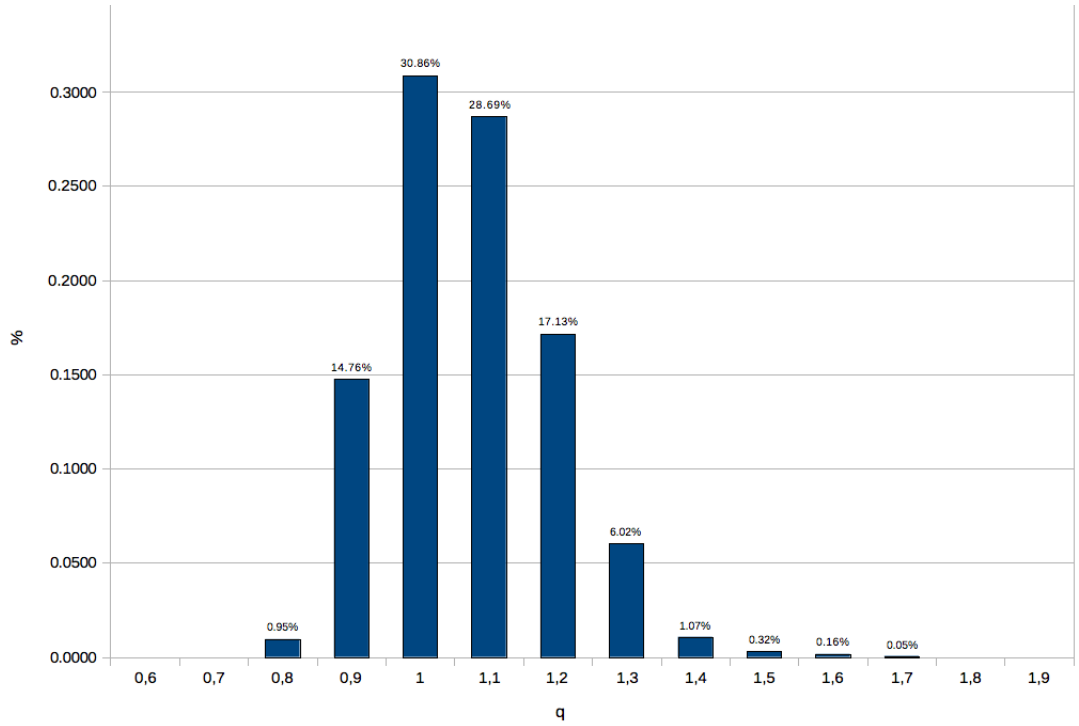



Figure 3.7: q results staircase histograms - type 1.

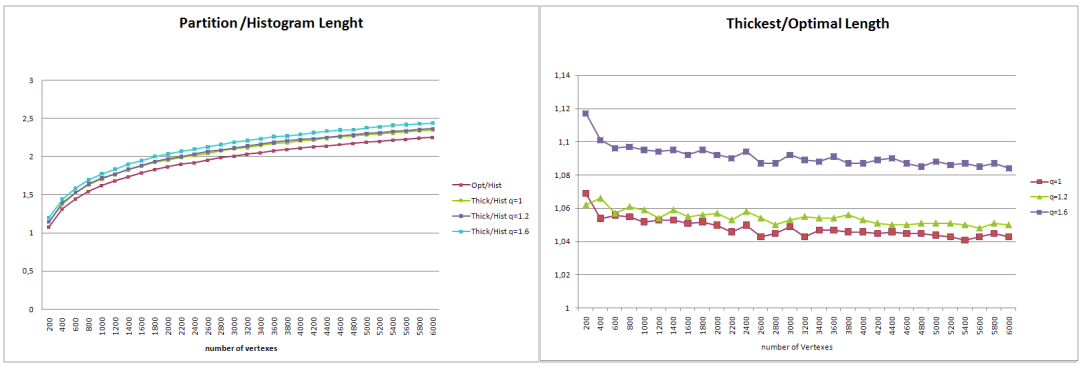
### Comparison between optimal and q-thickest

Differently from previous cases in which we've analyzed histograms with general shape, with staircase histograms we can note a precise behavior of the ration between partition length (of both techniques) and histogram length: it has a logarithmic growing trend with the increasing of the number of original vertices (Fig. 3.8.a).

Another time, with best value of $q$, results obtained by thickest method come out very near to those obtained by the application of optimal algorithm: the average ratio between the length of thickest and optimal solutions is in $(1.04 \div 1.05)$ (Fig. 3.8.b).

## 3.2.2 "Uniform" length

Like for the third type of general shape histogram, with the term "uniform", we mean that the only constraint during the histogram construction is to fix a maximal length of the base and the height of the histogram, in a manner the coordinates of the vertices are chosen inside the range $[0, max]$. We can obtain a




(a) *Ratio between obtained partitions and histograms lengths*

(b) *Ratio between q-thickest and optimal partition lengths*

Figure 3.8: comparison optimal-thickest method on staircase histograms - type 1.

procedure to do this starting with the procedure 3.3, with the addition of a simple control to avoid same values of any $y$-coordinates, as we've done for $x$-coordinates.

### *q*-study

For these histograms, the best values of $q$ remain 1 and 1.1, as in the case above. We can see in Fig. 3.9 and table 3.5 the obtained results from simulation tests. We can note that $q = 1.1$ has obtained and higher score even if it was the best less times than $q = 1$, due to score rules.

| **q** | 0,5 | 0,6 | 0,7 | 0,8 | 0,9 | 1 | 1,1 | 1,2 | 1,3 | 1,4 | 1,5 | 1,6 | 1,7 | 1,8 | 1,9 | 2 | 2,1 | 2,2 | 2,3 | 2,4 | 2,5 | 2,6 | 2,7 | 2,8 | 2,9 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0 | 0 | 2 | 13 | 70 | 107 | 81 | 21 | 5 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **2** | 0 | 0 | 1 | 12 | 29 | 39 | 95 | 92 | 26 | 4 | 2 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 1 | 6 | 5 | 11 | 30 | 96 | 102 | 33 | 10 | 2 | 3 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

Table 3.5: q results staircase histograms - type 2.

### Comparison between optimal and q-thickest

As with staircase histogram with fixed-length edges, we note (Fig. 3.10.a) that the ratio between partition length and histogram length has a logarithmic trend, but in this case it grows more slowly. Unlike the previous case, thickest solution appear to be less close to the optimal solution: we achieve an average ratio between thickest and optimal length of 1.07 with *q=1* (Fig. 3.10.a and 3.10.a).

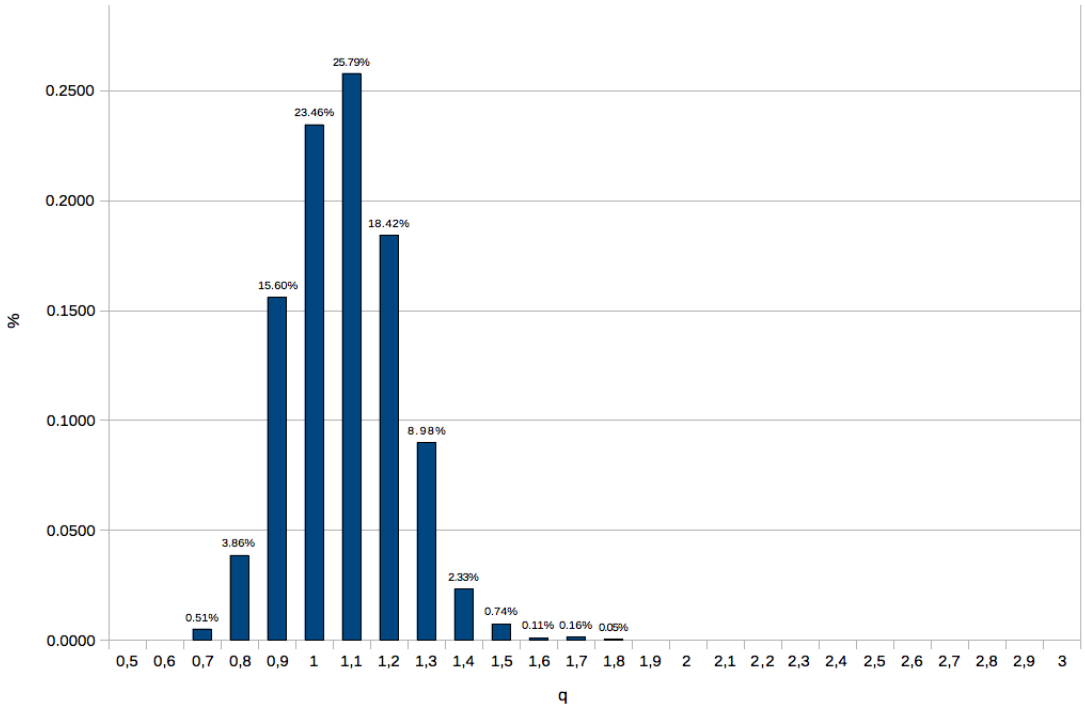
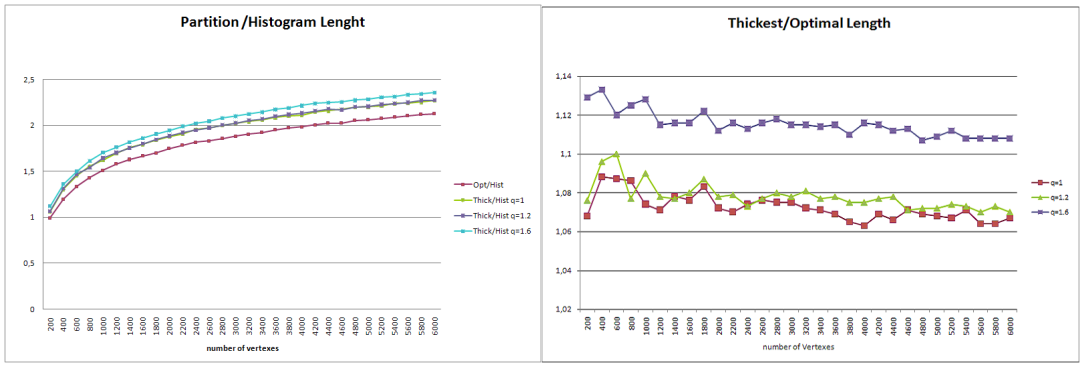



Figure 3.9: q results staircase histograms - type 2.




(a) *Ratio between obtained partitions and histograms lengths*

(b) *Ratio between q-thickest and optimal partition lengths*

Figure 3.10: comparison optimal-thickest method on staircase histograms - type 2.

## 3.3 Random symmetric histograms

This represent the last interesting case we've token into account. We can thinking about it as a sort of staircase histogram linked with its mirror image, so, with a little alteration of the procedure used for "uniform" staircase histograms, we are able to build histograms with that feature. Instead of generating all vertices heights, we produce only half of them, because the second half has the same $y$-coordinates.

### $q$-study

This last case represents (until now) the worst case for thickest first algorithm. Levcopoulos in [9] had studied it and he affirmed that with $q = \sqrt{2}$ every $q$-thickest partition has a length $\left(1 + \sqrt{2}\right) \times M(p)$. In our simulations, we've obtained like best values $q = \{1.4, 1.5, 1.6\}$ (see Fig. 3.11 and tab. 3.6).

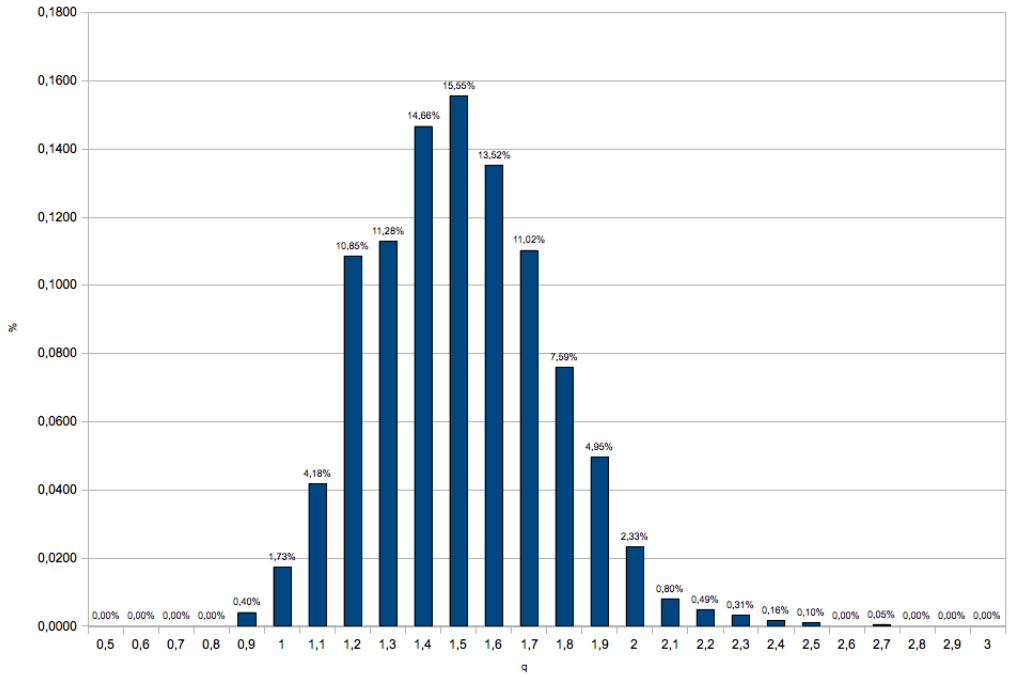



Figure 3.11: q results symmetric histograms.

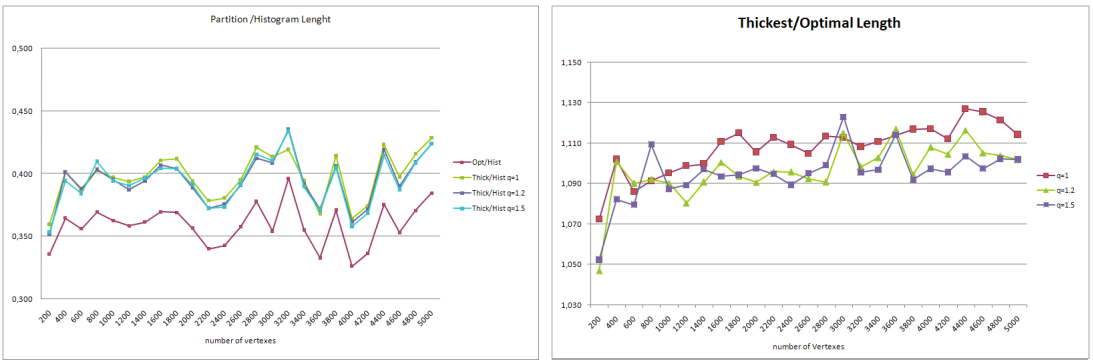
Also in this case, we note a symmetry on the distribution of $q$ values, centered on best values, but, differently from previous cases, there's not a big difference between the best value ($q = 1.5$) and the values near to it. In the following section we can see what this results means.

| q | 0,5 | 0,6 | 0,7 | 0,8 | 0,9 | 1 | 1,1 | 1,2 | 1,3 | 1,4 | 1,5 | 1,6 | 1,7 | 1,8 | 1,9 | 2 | 2,1 | 2,2 | 2,3 | 2,4 | 2,5 | 2,6 | 2,7 | 2,8 | 2,9 | 3 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **1** | 0 | 0 | 0 | 0 | 2 | 9 | 17 | 47 | 39 | 52 | 46 | 38 | 27 | 11 | 10 | 1 | 1 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **2** | 0 | 0 | 0 | 0 | 0 | 1 | 10 | 19 | 36 | 42 | 52 | 48 | 36 | 28 | 16 | 7 | 1 | 1 | 2 | 1 | 1 | 0 | 0 | 0 | 0 | 0 |
| **3** | 0 | 0 | 0 | 0 | 1 | 1 | 3 | 12 | 13 | 22 | 39 | 35 | 48 | 52 | 29 | 27 | 10 | 4 | 2 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |

Table 3.6: q results symmetric histograms.

### Comparison between optimal and q-thickest

This time, we change the set of values using $q = 1.5$ instead of $q = 1.6$, due to the fact $q = 1.5$ results to be better. As we can observe from the figure below (Fig. 3.12), this case is absolutely the worst: we confirm there are not large differences using different values of $q$ parameter, and the average ratio between thickest solution length and optimal one is about 1.10, only with general histograms of type 1 (fixed length of the edges) we have obtained a measure similar to this. A positive point is that, conversely from staircase histograms, we can note that the ratio between partition length and histogram perimeter is around 0.4.




(a) *Ratio between obtained partitions and histograms lengths*

(b) *Ratio between q-thickest and optimal partition lengths*

Figure 3.12: comparison optimal-thickest method on symmetric histograms.

## 3.4 Synthesis

In table 3.7 we have summarized the obtained results.

As we've already written in section 3.1.1, in all the cases we have considered, the trend of the thickest-first algorithm follows the trend of the optimal technique: when optimal method worsens (the ration between partition and histogram length increases), even the thickest one worsens.

| | | | Thickest ÷ Optimal | | | Thickest / Histogram | | Optimal / Histogram | |
|---|---|---|---|---|---|---|---|---|---|
| | | q | min | max | average | average | max | average | max |
| GENERAL | FIXED | 1,6 | 1,062 | 1,087 | 1,077 | 0,832 | 0,956 | 0,772 | 0,880 |
| | SEMI-FIXED | 1,2 | 1,016 | 1,036 | 1,023 | 0,017 | 0,087 | 0,016 | 0,085 |
| | UNIFORM | 1 | 1,016 | 1,024 | 1,019 | 0,074 | 0,116 | 0,072 | 0,114 |
| STAIRCASE | FIXED | 1 | 1,041 | 1,069 | 1,048 | 2,022 | 2,345 | 1,930 | 2,250 |
| | UNIFORM | 1 | 1,063 | 1,088 | 1,072 | 1,941 | 2,267 | 1,812 | 2,125 |
| SYMMETRIC | | 1,5 | 1,052 | 1,109 | 1,089 | 0,389 | 0,410 | 0,357 | 0,370 |

Table 3.7: Synthesis of the results.

The interesting point is the fact that, for each type of histograms we considered, the approximation algorithm obtained good results not too far from those obtained by the optimal method. In worst cases, represented by general histograms of type 1 and symmetric histograms, we have a multiplication factor equals to 1.1. Due to this obtained result, we can affirm that in practical situations, and in average cases, the bound $\left((1+\sqrt{2}) \times M(p)\right)$ is an over-stimation we have never reach.

## 3.5 Optimal comparison

In this section we analyze the different behaviors of the optimal algorithm in relation with the shapes of the histograms. We focus our attention on the number of subproblems it really solves[1], the computation time[2] and the ratio between length of the obtained partition and the length of the original histograms[3](even if this last is already shown in section above). All the values in tables 3.8 and 3.9, are an average of several tests on histograms with different sizes and with different shapes.

If we consider the ratio between the lengths of the solutions and the lengths of the histograms shapes, the worst cases are represented by the staircase histograms. With these polygons the ratio has a logarithmic trend increasing with the size of the problem. Interesting is the fact that with symmetric histograms, the ratio is almost constant. Instead, with general histogram of type 1, the trend tends to a slight increase.

Analysing the other two parameters, the computation time and the number of really solved subproblems, we note immediately the following fact: histograms with staircase and symmetric shapes, present a similar behaviour, especially on the number of subproblems. We can interpolate the curves in figure 3.13 using 3-

[1] *Prob* in tables 3.8 and 3.9.
[2] *Time* in tables 3.8 and 3.9.
[3] *Opt/Hist* in tables 3.8 and 3.9.

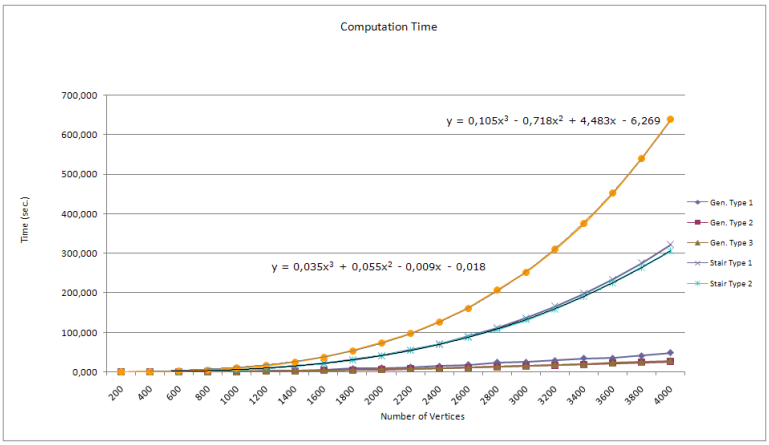
| | GENERAL HISTOGRAMS | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | Fixed | | | Semi-Fixed | | | Uniform | | |
| N | Opt/Hist | Time | Prob | Opt/Hist | Time | Prob | Opt/Hist | Time | Prob |
| 200 | 0,631 | 0,052 | 4,79E+04 | 0,085 | 0,052 | 3,60E+04 | 0,114 | 0,048 | 3,80E+04 |
| 400 | 0,571 | 0,260 | 2,40E+05 | 0,049 | 0,212 | 1,59E+05 | 0,089 | 0,204 | 1,45E+05 |
| 600 | 0,585 | 0,630 | 5,48E+05 | 0,040 | 0,478 | 3,33E+05 | 0,091 | 0,496 | 3,38E+05 |
| 800 | 0,736 | 1,304 | 1,62E+06 | 0,030 | 0,886 | 6,21E+05 | 0,087 | 0,896 | 6,30E+05 |
| 1000 | 0,667 | 1,682 | 1,48E+06 | 0,025 | 1,396 | 9,41E+05 | 0,071 | 1,466 | 9,67E+05 |
| 1200 | 0,755 | 3,414 | 4,04E+06 | 0,021 | 2,086 | 1,44E+06 | 0,091 | 2,154 | 1,44E+06 |
| 1400 | 0,722 | 4,484 | 4,60E+06 | 0,019 | 3,006 | 2,05E+06 | 0,072 | 3,060 | 1,92E+06 |
| 1600 | 0,742 | 6,538 | 7,15E+06 | 0,018 | 3,888 | 2,46E+06 | 0,090 | 4,084 | 2,47E+06 |
| 1800 | 0,746 | 8,676 | 9,87E+06 | 0,016 | 5,012 | 3,04E+06 | 0,081 | 5,170 | 3,08E+06 |
| 2000 | 0,761 | 10,312 | 1,26E+07 | 0,015 | 6,454 | 4,01E+06 | 0,069 | 6,626 | 4,00E+06 |
| 2200 | 0,787 | 12,028 | 1,41E+07 | 0,013 | 7,694 | 4,71E+06 | 0,076 | 8,162 | 4,97E+06 |
| 2400 | 0,797 | 14,702 | 1,74E+07 | 0,013 | 9,468 | 5,57E+06 | 0,086 | 9,494 | 5,48E+06 |
| 2600 | 0,788 | 18,642 | 2,22E+07 | 0,012 | 11,312 | 7,22E+06 | 0,067 | 11,596 | 6,94E+06 |
| 2800 | 0,772 | 24,190 | 2,97E+07 | 0,011 | 12,732 | 7,43E+06 | 0,069 | 13,512 | 7,97E+06 |
| 3000 | 0,846 | 24,810 | 3,45E+07 | 0,010 | 14,814 | 8,76E+06 | 0,073 | 15,378 | 8,86E+06 |
| 3200 | 0,824 | 29,472 | 3,96E+07 | 0,010 | 16,824 | 9,70E+06 | 0,071 | 17,740 | 1,02E+07 |
| 3400 | 0,802 | 34,374 | 4,33E+07 | 0,010 | 19,442 | 1,18E+07 | 0,058 | 19,722 | 1,11E+07 |
| 3600 | 0,880 | 35,684 | 5,24E+07 | 0,009 | 21,892 | 1,31E+07 | 0,075 | 22,796 | 1,34E+07 |
| 3800 | 0,853 | 41,408 | 5,52E+07 | 0,009 | 24,726 | 1,40E+07 | 0,049 | 25,348 | 1,44E+07 |
| 4000 | 0,853 | 48,718 | 6,33E+07 | 0,008 | 26,800 | 1,53E+07 | 0,080 | 28,454 | 1,68E+07 |

Table 3.8: Optimal method synthesis - General Histograms.

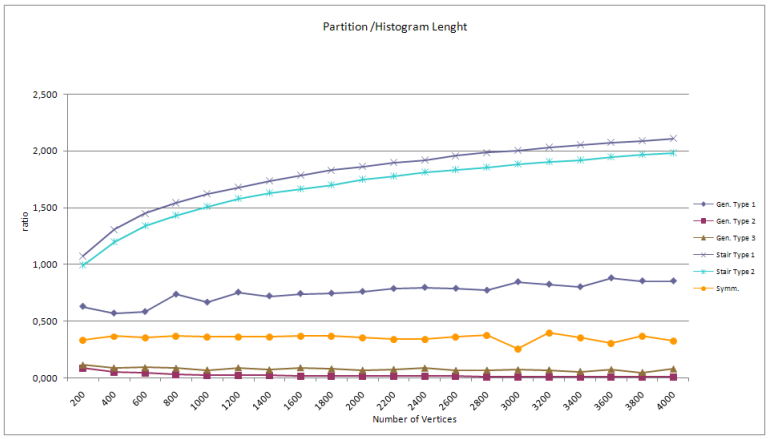
degree polynomial interpolation lines. While, with general histograms, we obtain a 2-degree interpolation line. Even if the staircase and symmetric cases are the worst we encountered, they are far below from the maximal theoretical bound (see Fig. 3.13.c). The small difference between these cases respect the computation time, is probably due to the managing of the insertions of horizontal segments. In symmetric case, for each horizontal edges we have another edge with same height. To govern this situation, the current implementation requires some additional controls and, of consequence, additional time.

| | STAIRCASE HISTOGRAMS | | | | | | SYMMETRIC HISTOGRAMS | | |
|---|---|---|---|---|---|---|---|---|---|
| | Fixed | | | Uniform | | | | | |
| N | Opt/Hist | Time | Prob | Opt/Hist | Time | Prob | Opt/Hist | Time | Prob |
| 200 | 1,076 | 0,078 | 1,76E+05 | 0,993 | 0,076 | 1,76E+05 | 0,336 | 0,106 | 1,84E+05 |
| 400 | 1,310 | 0,462 | 1,37E+06 | 1,199 | 0,458 | 1,37E+06 | 0,364 | 0,674 | 1,40E+06 |
| 600 | 1,448 | 1,436 | 4,59E+06 | 1,336 | 1,398 | 4,59E+06 | 0,356 | 2,202 | 4,66E+06 |
| 800 | 1,543 | 3,184 | 1,08E+07 | 1,431 | 3,110 | 1,08E+07 | 0,369 | 5,042 | 1,09E+07 |
| 1000 | 1,621 | 5,958 | 2,11E+07 | 1,509 | 5,792 | 2,11E+07 | 0,363 | 9,612 | 2,13E+07 |
| 1200 | 1,678 | 9,914 | 3,64E+07 | 1,579 | 9,646 | 3,64E+07 | 0,358 | 16,414 | 3,66E+07 |
| 1400 | 1,734 | 15,244 | 5,77E+07 | 1,628 | 14,846 | 5,76E+07 | 0,361 | 25,686 | 5,80E+07 |
| 1600 | 1,783 | 22,168 | 8,60E+07 | 1,666 | 21,618 | 8,60E+07 | 0,370 | 37,996 | 8,65E+07 |
| 1800 | 1,830 | 31,214 | 1,22E+08 | 1,699 | 30,274 | 1,22E+08 | 0,369 | 53,702 | 1,23E+08 |
| 2000 | 1,862 | 41,962 | 1,68E+08 | 1,748 | 41,128 | 1,68E+08 | 0,357 | 73,696 | 1,68E+08 |
| 2200 | 1,899 | 55,358 | 2,23E+08 | 1,781 | 54,250 | 2,23E+08 | 0,340 | 97,390 | 2,24E+08 |
| 2400 | 1,920 | 71,110 | 2,89E+08 | 1,816 | 69,676 | 2,89E+08 | 0,343 | 126,610 | 2,90E+08 |
| 2600 | 1,956 | 90,136 | 3,68E+08 | 1,831 | 88,064 | 3,67E+08 | 0,358 | 161,134 | 3,69E+08 |
| 2800 | 1,984 | 111,828 | 4,59E+08 | 1,858 | 107,722 | 4,59E+08 | 0,378 | 207,042 | 4,60E+08 |
| 3000 | 2,005 | 136,848 | 5,65E+08 | 1,881 | 132,060 | 5,64E+08 | 0,254 | 252,272 | 5,66E+08 |
| 3200 | 2,028 | 166,414 | 6,85E+08 | 1,903 | 159,588 | 6,84E+08 | 0,396 | 309,974 | 6,86E+08 |
| 3400 | 2,050 | 198,452 | 8,22E+08 | 1,923 | 191,318 | 8,21E+08 | 0,355 | 375,486 | 8,22E+08 |
| 3600 | 2,073 | 234,162 | 9,75E+08 | 1,948 | 226,098 | 9,75E+08 | 0,303 | 452,066 | 9,73E+08 |
| 3800 | 2,088 | 275,596 | 1,15E+09 | 1,972 | 264,734 | 1,14E+09 | 0,371 | 541,016 | 1,15E+09 |
| 4000 | 2,108 | 322,338 | 1,34E+09 | 1,983 | 306,468 | 1,34E+09 | 0,326 | 639,064 | 1,34E+09 |

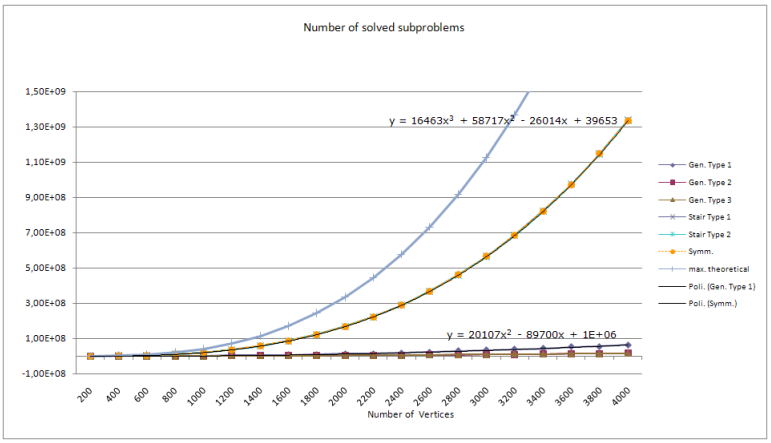
Table 3.9: Optimal method synthesis - Staircase and Symmetric Histograms.




(a) *Time analysis.*




(b) *Ratio between lengths of partitions and lengths of histograms.*




(c) *Number of really solved subproblem.*

Figure 3.13: Optimal method results.

## 3.6 Examples of partitions

The figures in this section are examples of how both algorithms work on the different kinds of histograms we have studied above.

In each example, we can note three different colours. These colours are the same as those used in the tool and they have been associated with the algorithms:

- cyan is the colour of the segments produced by the *optimal* method;
- red is the colour of the segments produced by the *q-thickest* method;
- violet is the colour of the common segments.

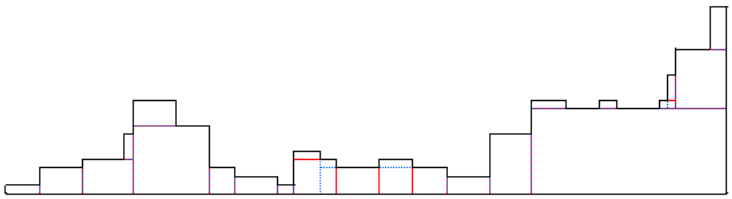


Figure 3.14: Example of partition on general histograms - type 1.

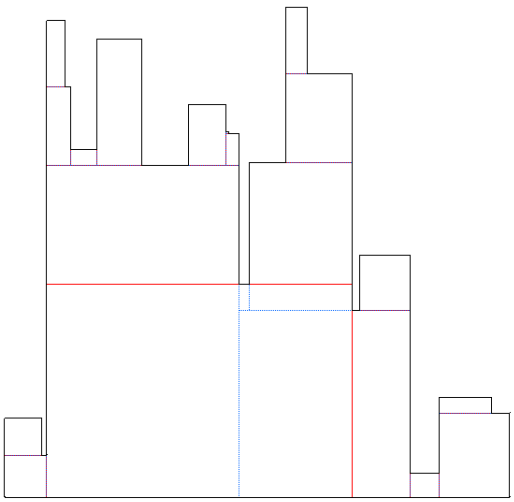


Figure 3.15: Example of partition on general histograms - type 2.

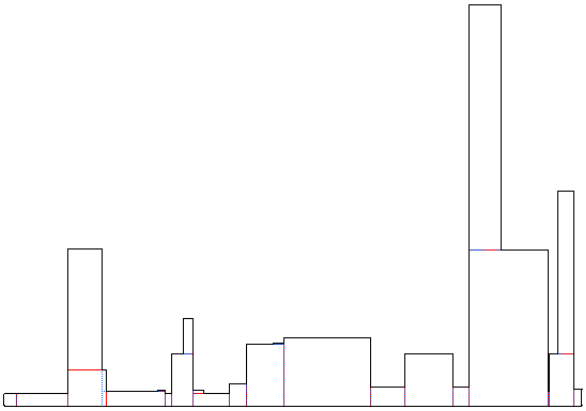


Figure 3.16: Example of partition on general histograms - type 3.

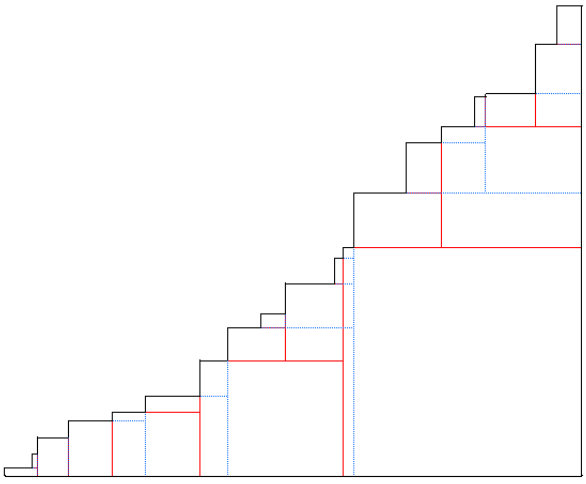


Figure 3.17: Example of partition on staircase histograms - type 1.

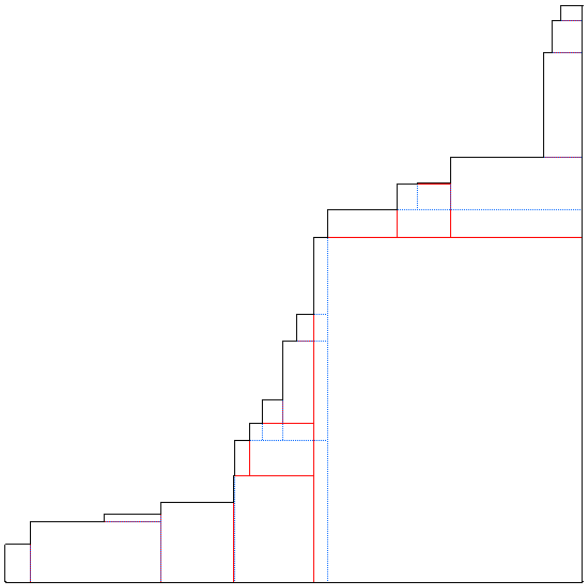


Figure 3.18: Example of partition on staircase histograms - type 2.

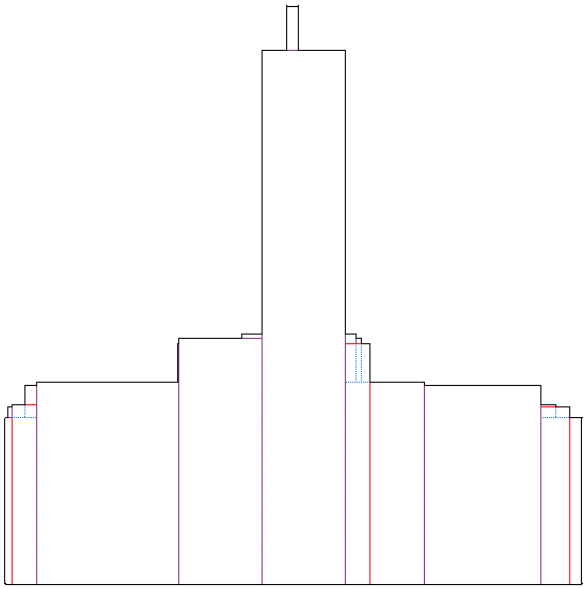


Figure 3.19: Example of partition on symmetric histograms.

# Chapter 4

# GUI tool

In this section, we will describe how the GUI tool we have implemented appears to users and how to use it.

The tool is an important part of the thesis work. We used it especially to test the validity of both implementations of the algorithms. In the initial step, we tested our source code by outputting simple text messages: we printed some information like the inserted segments or where is the position of the thickest rectangle, in each step. Using simple examples of histograms with few vertices, we computed the solutions by hand and checked if they overlapped; when testing histograms with tens of vertices, producing and testing hand solutions was impossible. This is the reason why the GUI tool was very helpful in the test phase. It was equally helpful in the second phase where we were studying the effect of choosing different $q$-values on histograms of various shapes. It was developed and utilized more in first part of my work, in parallel with the implementations of the methods. In the phase in which we have produced experimental results, we had to elaborate more simulations, so we created a few test files to do it automatically.

### How to use the tool

The GUI tool appears like in figure 4.1. It's possible to define five areas; each of them has a different utility.

The canvas stands at the center of the tool. It's used to visualize the histograms we want to compute and the sets of segments produced by the algorithms.

On the top there are *Models folder* and *Results folder* buttons. The first button gives the possibility to set the folder in which histograms can be stored for later use; the second one allows to fix the folder where the computed solutions have to be saved. The descriptions of the histogram have to be in a specific form: on the top any kind information about them can be included but, in the beginning of

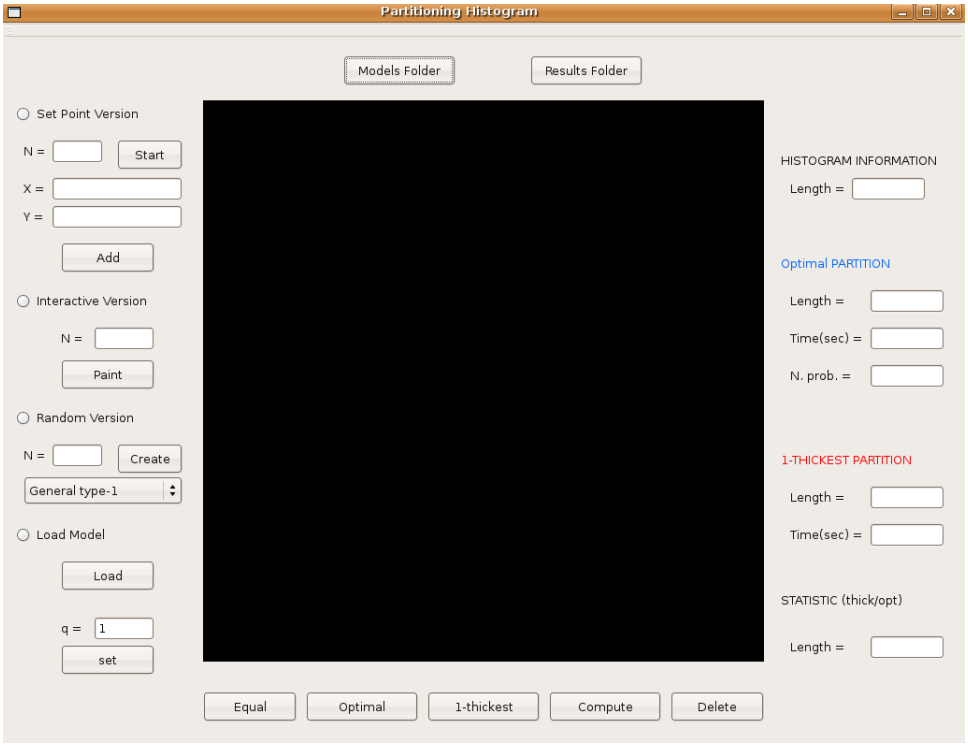



Figure 4.1: How the GUI tool appears.

each line, there has to be a character # (indicating it's a comment line); the first line used to read the model contains an integer which indicates the total number of vertices. After that there is the list of $x$ and $y$ coordinates of the vertices, separated by an empty space. Result files are similar to model files, but after a line with some information (commented) there is the list of the coordinates of the segments' endpoints in the partition set. Every time the GUI tool is opened, the paths of these folders are set to the path where the GUI tool itself resides.

The left zone is completely dedicated to the creation of a histogram. We have four possibilities:

- *Set Point Version.* It gives the chance to insert the vertices by "hand"; the first step is to set the number of original vertices of the histogram (it must be odd and positive). Then the user inserts the coordinates of the vertices. The points have to be inserted in clockwise order, starting with the left vertex of the base $(0, 0)$. The right endpoint is automatically detected with the coordinates of the penultimate one.

- *Interactive Version.* With this option, the user fixes the vertices with the use of the mouse, directly inside the canvas. In this case he sets the number

of horizontal edges instead of the number of original vertices. After that, the base of the histogram appears on the canvas and the user has to click with the mouse at the place of right endpoint of each single edge. For the last edge, only its height can be decided, since the right endpoint is obvious.

- *Random Version*. This way gives the user the possibility to create a random histogram only setting up the number of original vertices he wants. The user can choose the type of histogram he wishes. The types are those we have studied in the third chapter.

- *Load Model*. This version allows to load a model of a histogram. The user has only to search the path of the model he wants to compute through the use of a window that appears when he pushes the *load* button.

When a histogram is made, the associated model is automatically created and stored in the *Models folder*. The name of the model will be like *version_numVertex.hist*. There is also a label where the user can set the value of the $q$ parameter used in the thickest method. If an histogram is painted in the canvas, pressing the *set* button the tool (re)compute the *q-tickest* partition with the value just inserted.

At the bottom, there are five buttons: *Equal*,*Compute*, *Optimal*, *q-thickest* and *Delete*. From second to fifth, the buttons explain themselves: *Compute* (*Optimal* and *q-thickest*) serves to start the partitioning of the current histogram painted on the canvas; when a partition set is determined, a result file named *partition-Type_modelName.ris* will be created. *Delete* is utilized when the user wants to clear the canvas and the information on the right, and also to free the memory from the structures used by the methods. The *Equal* button allows the user to change the display scale of the canvas; by default there is the same scale on x- and y-axis, equal to the maximum value between the base and the height of the histogram. Unfortunately, when there's a big difference between these two values, we may have a histogram which looks too narrow or too flat. When it happens, it's difficult too see the shape of the histogram and its partition set. With this button, we use two different scales for the x- and y-axis, based respectively on the base and on the height of the histogram.

The right column provides some information about the current histogram and its partitions obtained by the optimal and q-tickest algorithms. They are given in the following order:

- the length of the perimeter of the current histogram;

- the length of the optimal partition;

- the time spent to find the optimal partition;

- the number of subproblems really solved;
- the length of the q-thickest partition;
- the time spent to find the q-thickest partition;
- the ratio between the lengths of q-thickest and optimal partition.

As we can see the words “OPTIMAL PARTITION” and “q-THICKEST PARTITION” are coloured; it was a choice taken to be able to differentiate the set of segments produced by the first method from that produced by the second one. In the canvas, we use the same colours to draw the segments associated with one of two algorithms. We may note a third colour (which is the union of the other two) that highlights the common segments (see Fig. 4.2).

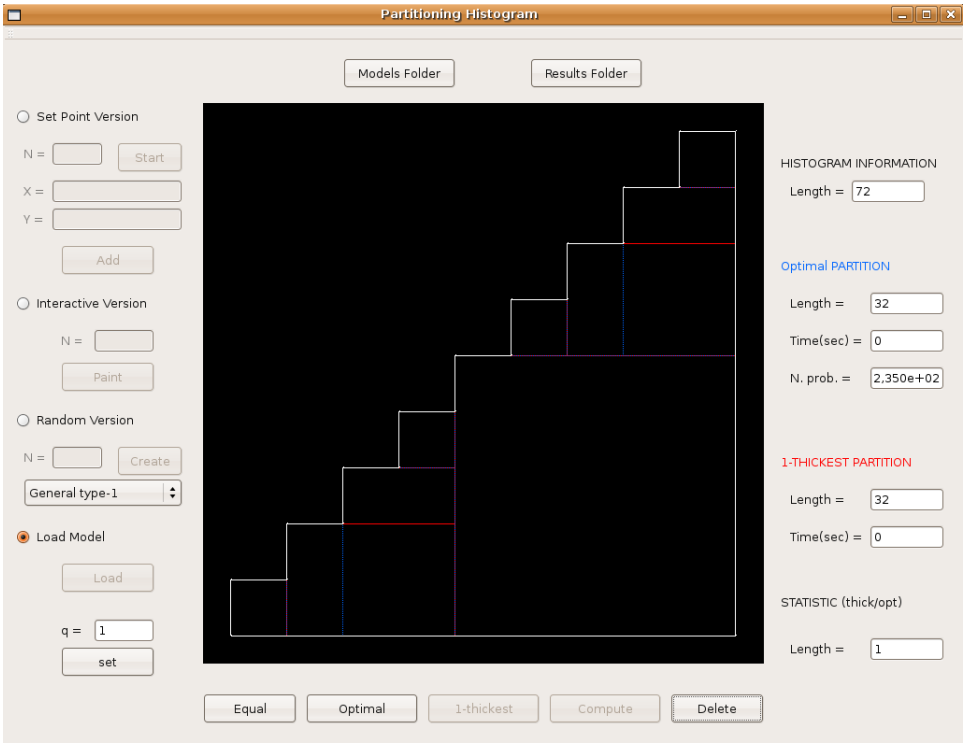



Figure 4.2: Example of GUI with solved histogram.

# Chapter 5

# Conclusions and future work

In this work, I have treat the problem of partitioning histograms in rectangles. A histogram is a special case of the family of hole-free isothetic polygons. I studied and compared two algorithms which can partition histograms defined by $n$ vertices. The two algorithms are the *optimal* method and *q-thickest first* method. Both based themselves on a simple idea: divide the original problem into sub-problems. The *optimal* method starts from the subproblems (*minimal* subproblems) with the aim of build the original solution using solutions of smaller problems. Instead, the *q-thickest first* technique, tries to find the solution splitting the original histogram into subhistogram with smaller size (in best case it divides the polygon in three sub-polygons with the same size, about one-third o the original size). I have implemented this algorithms in *C++* language and I've tried to explain, as clearly as possible, the choices made during their development. The target of these choices was to restrict the space e and computation time united with the use of simple structures.

The time complexities, resulting from the implementation and theoretically studied are:

| | |
|---|---|
| optimal method: | $\in O(n^3)$ |
| $q$-thickest method: | $\in O(n \log n)$[1] |

while the space complexities (specific to this implementation) are:

| | |
|---|---|
| optimal method: | $\in O(n^2)$ |
| $q$-thickest method: | $\in O(n)$ |

[1] this result must be associate in the average case, on worst case we obtain $O(n^2)$.

In the third chapter, I analysed the trend of the $q$-thickest method varying the value of the $q$ parameter in the range $[0.5, 3]$ on histograms with different shapes and sizes. Then, I compared the results of this method with the results obtained with *optimal* method. In this way, I was able to verify the goodness of the solutions obtained by the approximation algorithm, in terms of the ratio between the partitions lengths. The most interesting results are the following:

- the trend of $q$-thickest method proceeds as the trend of optimal one;
- all our experiments, we never reached the upper bound proposed in [9].

For this reasons, we can affirm that the $q$-thickest partition algorithm, with the appropriate $q$-value, produces a good approximation of the solution defined by *optimal* method[2].

At the end, I wanted to test the behaviour of the *optimal* method. I focused my attention on the computation time and the number of solved subproblems, with the aim to find the most problematic situation.

**Future work**

In problems like this one, where the goal is to minimise (or maximum) an objective function, there always are a lot things that can be improved or modified.

One of the crucial points has been when I was studying the $q$-value that best fit with a histogram of specific shape. In some cases, i.e., general histograms of type-2 (section 3.1.2), we have obtained a specific value for the parameter, but the cause is not clear. A possible future work will be understand these reasons, maybe changing the score rules used to decide best values of $q$ or using a completely different approach.

Another point that can be investigated, it is to check the real behaviour of the approximation algorithm. In sub-section 2.3.2, I have proposed the possible recurrence equation for q-thickest algorithm (eq. 2.11). Even if the experimental results were good, it would be stimulating find a model that is as close as possible to the action of the algorithm. In this work I've analysed six types of histograms, maybe there are other amusing cases which can lead to unexpected and new results.

Maybe, it is possible to look for some relation of this problem with the *minimum Manhattan network problem* where every pair of input points has to be linked with a rectilinear path of minimum length, and the objective function is to minimise the total length. This is an *NP-complete* problem.

Regarding the implementation of the algorithm, a possible work is to use other structures with the aim to speed up the time-computation, especially for optimal

[2] from average results obtained in simulations on histograms with different shapes.

method. For example, instead of the array it could be introduced a hash table. In this way the couple $[i, j]$ (indicating the sub-histogram induced by the vertices $i$ and $j$) can be used as the index of its position in the table.

The GUI tool could be also improved adding some features as the possibility to zoom some specific areas of the histograms. Useful, it may be enlarge the GUI to polygons, in order to have an idea on how good the algorithms work.

# Bibliography


[1] N. M. Amato, M. T. Goodrich, and E. A. Ramos, *Linear-time triangulation of a simple polygon made easier via randomization*, In Proc. 16th Annu. ACM Sympos. Comput. Geom., pages 201-212, Clear Water Bay, Hong Kong, China, June, 2000,

[2] M.G. Borgelt, *Fixed-Parameter Algorithms for Optimal Convex Partitions and other Results*, Department of Computer Science Lund University, Sweden, 2006

[3] B. Chazelle, *Triangulating a simple polygon in linear time*, Discrete Comput. Geom., 6(5):485-524, 1991

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein, *Introduction to Algorithms* 3rd edition, MIT Press , Massachusetts, USA 2009

[5] D. Du, Y. Zhang, *On Heuristics for Minimum Length Rectilinear Partitions*, Algorithmica Volume 5, Numbers 1-4, Springer New York, June, 1990

[6] A. Gonzalez-Gutierrez, T. F. Gonzalez,*Approximation Algorithms for the Minimum-Length Corridor and Related Problems*, Proceedings of the 19th Annual Canadian Conference on Computational Geometry, CCCG:253-256, Carleton University, Ottawa, Canada, August, 2007

[7] T. F. Gonzalez, M. Razzazi, M. Shing, S. Zheng, *On Optimal Guillotine Partitions Approximating Optimal D-box Partitions*, Comput. Geom. 4: 1-11, 1994

[8] C. Levcopoulos, A. Östlin, *Linear-Time Heuristics for Minimum Weight Rectangulation*, SWAT '96, 5th Scandinavian Workshop on Algorithm Theory, Reykjavk, Iceland, July, 1996

[9] C. Levcopoulos, *Heuristics for minimum decompositions of polygons*, Linköping Studies in Science and Technology. Dissertations. No. 155, Sweden, 1987

[10] A. Lingas, R.Y. Pinter, R.L. Rivest, A. Shamir, *Minimum Edge Length Partitioning of Rectilinear Polygons*, Proc. 20th Allerton Conf. on Comm. Control and Compt., Monticello, Illinois, 1982