

Efficient Computation of the Convex Hull on Sets of Points Stored in a k^2 -tree Compact Data Structure

Juan Felipe Castro

Universidad del Bío-Bío, Chile

Miguel Romero

Universidad del Bío-Bío, Chile

Gilberto Gutiérrez

Universidad del Bío-Bío, Chile

Mónica Caniupán

Universidad del Bío-Bío, Chile

Abstract

In this paper we present two algorithms to obtain the Convex Hull of a set of points that are stored in a compact data structure called k^2 -tree. This problem consists in given a set of points P in the Euclidean space obtaining the smallest convex region (polygon) containing P . Traditional algorithms to compute the convex hull do not scale well for large databases, such as, spatial databases, since the data does not resides in main memory. We use the k^2 -tree compact data structure to represent, in main memory, efficiently a binary adjacency matrix representing points over a 2D-space. This structure allows an efficient navigation in a compressed form.

The first algorithm, called CHk^2 is based on the idea of computing the four extreme points in the space containing P . These four points form an approximate convex hull for P called the *candidate convex hull*. Then, iteratively, the algorithm adjusts the candidate convex hull until a definitive solution is achieved. This is done by taking advantage of the properties of the k^2 -tree. The second algorithm, called CHk^2t is based on the first algorithm CHk^2 , and implements a heuristic that allows the reduction of the points to be explored to find a solution. We report experiments over real and synthetic data that show efficiency in terms of time and space of both algorithms. Also, we compare our solution with a naive implementation of the Graham Scan algorithm over a set of points taken from a k^2 -tree structure.

Keywords: algorithms, data structures, spatial databases, computational geometry

Email addresses: jucastr@egresados.ubiobio.cl (Juan Felipe Castro),
miguel.romero@ubiobio.cl (Miguel Romero), gutierrez@ubiobio.cl (Gilberto Gutiérrez),
mcaniupan@ubiobio.cl (Mónica Caniupán)

1. Introduction

The *Convex Hull* is a famous problem in computational geometry [1]. This problem is defined as follows: Given a set of points P in the Euclidean space, obtaining the smallest convex region (polygon) containing P . There are several proposed algorithms to compute the convex hull such as [2, 3, 4, 5]. The computation of the convex hull is relevant for different applications, such as, to describe the shape of clusters [6] in data mining, to represent patterns in pattern recognition applications [7], to identify clusters of points in spatial databases [8], among others.

Traditional algorithms to compute the convex hull do not scale well for large databases, such as, spatial databases, since the data does not reside in main memory, but mostly in disks. A good algorithm is the one reported in [8] where the set of points are stored in a multidimensional hierarchical data structure (such as a R-tree index [9]) that allows to speed up the computation. In this paper we propose the computation of the convex hull for a set of points represented in a compact data structure called in k^2 -tree [10].

Compact data structures are data structures that use small amount of space but allow for efficient query operations [11]. They permit to process large data sets in main memory avoiding partially or completely the access to external memory such as a disk. They can be located in the upper levels of the memory hierarchy (closed to the CPU), where the access time have decreased much faster than the lower levels of the hierarchy. Locating the data into the main may result in improvement of the performance of query processing [12]. Even though compact data structures are not useful for every kind of data, they can be successfully used to represent spatial data sets [13, 14].

The first algorithm, called CHk^2 is based on the idea of computing the four extreme points in the space containing P . These four points form an approximate convex hull for P called the *candidate convex hull*. Then, iteratively, the algorithm adjusts the candidate convex hull until a definitive solution is achieved. This is done by taking advantage of the properties of the k^2 -tree. The second algorithm, called CHk^2t is based on the first algorithm CHk^2 , and implements a heuristic that allows the reduction of points to be explored to find a solution. Both algorithms process the data directly from the compact data structure k^2 -tree without accessing the original data, and both require a minimum extra space, which is proportional to the number of vertices of the definitive convex polygon.

We report a set of experiments considering real and synthetic data. The experiments show that the algorithms are efficient to compute the convex hull in terms of time and space. We compared our algorithms with a naive implementation of the convex hull over the k^2 -tree, which consists in obtaining all the points from the k^2 -tree and apply the Graham Scan algorithm to obtain the convex hull [15].

The rest of the paper is organized as follows: Section 2 describes related work. Section 3 presents concepts that are needed to understand the algorithms we develop. Section 4 presents algorithms CHk^2 and CHk^2t that compute the convex hull of a set of points represented in a k^2 -tree structure. Section 5 describes experimentation results. Finally, Section 6 presents final remarks.

2. Related Work

In this section we describe the main algorithms reported in the literature to compute the convex hull for a set of points stored at main memory, and for points stored in disk.

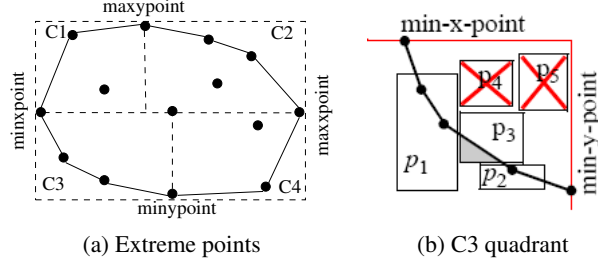


Figure 1: Extreme points and C3 quadrant [8]

2.1. Algorithms for Convex Hull on Points Stored in Main Memory

The algorithm proposed in 1972 by Ronald Graham [15] is one of the former efficient algorithms to compute the convex hull, running in $O(n \log n)$. The algorithm is based on the idea of finding the extreme points of the convex set. It works in five steps: (i) given a set of points $S = \{s_1, s_2, \dots, s_n\}$ finds a point p in the plane that is considered the centroid point, this is, p is not in the edges of the convex polygon, but it is in the interior of the convex set. (ii) Each point in S is expressed in polar coordinates with respect to p and to an arbitrary line defined from p . (iii) The points are ordered in terms of increasing polar coordinates. (iv) If two points are collinear, the algorithm deletes the one with the smaller amplitude. (v) It searches for the extreme points by taking, at each step, three consecutive points and analyzing the angle of the points with respect to p , points that are proved to be non-extreme are deleted. At the end of the process the algorithm returns the extreme points of the convex polygon.

Other interesting solution is the Quickhull Algorithm [4, 16]. It starts by finding two extreme points, the one having the minimum value in coordinate x and the one having the maximum value. Then it generates a line segment l_0 with the extreme points. After that, it defines the most distant point p located over l_0 or below l_0 (depending of the orientation of the searching), and generates a new segment l_1 with one of the extreme points and p . This process is repeated until the convex polygon is found. The algorithm works better when there are few extreme points. In the worst case the algorithm runs in $O(n^2)$, but in average it runs in $O(n \log n)$.

2.2. Algorithms for Convex Hull on Points Stored in Disk

In [8] the authors report algorithms to compute the convex hull of points stored in multidimensional indexes such as R-trees [9]. They implement two algorithms called, respectively, the *distance-priority algorithm* and the *depth-first algorithm*. The algorithms are based on the location of four extreme points, which are called the *min-x-point*, *min-y-point*, *max-x-point*, and *max-y-point* and are guarantee to be part of the convex hull (see Figure 1(a)). These points form four quadrants where the convex hull is searched. Consider quadrant C3, the *distance-priority algorithm* creates a segment from the min-x-point to the min-y-point which is called the *Tentative Convex Hull* (TCH). Then, it selects the pages that will be visited and excludes the unnecessary pages. For instance, page p_4 and p_5 in Figure 1(b) are excluded since it is impossible that a point stored there can be part of the convex hull, in fact, they are all contained in the convex polygon. However, pages p_1 and p_2 contain points in the convex hull, so

they are kept. Also, it is necessary to know the points stored in p_3 since some of them could be part of the convex hull. Algorithm *distance-priority algorithm* access all the pages that may contain points in the convex polygon, this is done efficiently by ordering the pages by decreasing distance between the lower left corner and the TCH. The *depth-first algorithm* processes the pages in sequence order of the points in the convex hull. It starts at the min-x-point and ends at the min-y-point. It uses the multidimensional indexes to perform the process efficiently. In this way, the points are inserted into and deleted from the TCH at the end of the process.

3. Preliminaries

In this section we introduce briefly the k^2 -tree compact data structure, and some other concepts we use in the algorithms presented in Section 4.

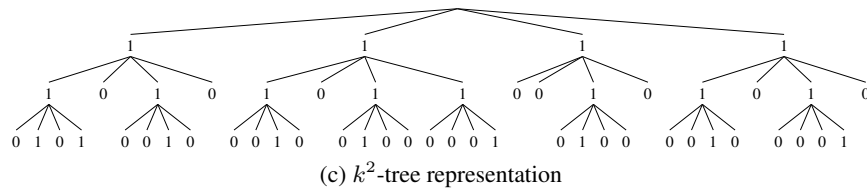
3.1. K^2 -trees

A k^2 -tree is a compact data structure used to represent binary adjacency matrices. It has been successfully used to represent the graph of the Web [10]. It allows an efficient representation of binary matrices and efficient navigation in a compressed form. Given a matrix M of size $n \times n$, the k^2 -tree representation is constructed as follows. First, M will be divided recursively in k^2 quadrants of similar size. For instance, if $k = 2$, the matrix will be divided in four quadrants, and each of them will be divided in the same number of quadrants, and so on. Each node in the tree contains either a 1 or a 0. The former represent internal nodes and 0 are always leaves, except for the last level where all nodes are leaves and a 1 represent a bit value from the matrix. The root of the tree is numbered with 0, it has k^2 children in level 1. The internal nodes, which have a value 1, have exactly k^2 children.

In this paper we use a k^2 -tree structure to represent points over a set of coordinates. Given a set of points S over a subspace $\mathcal{R} \subseteq \mathbb{Z}^2$ forming a rectangle with sides parallel to coordinates x and y . The adjacency matrix has cells from $(0, 0)$ to $(n - 1, n - 1)$ where columns represent values on coordinate x , and rows represent values on coordinate y , both ordered from negative values to positive values. Consider the subspace in Figure 2(a), cell $(0, 0)$ in matrix of Figure 2(b) represents the pair (min_x, min_y) in \mathcal{R} , this is, coordinate $(-4, -4)$. In the matrix, a 1 represents the presence of a point in the corresponding coordinate, for instance, point $(-3, 2)$ in Figure 2(a) is represented with a 1 in cell $(6, 1)$ of the matrix (see Figure 2(b)), because row 6 represents value 2 of coordinate y , and column 1 represents value -3 in coordinate x . Figure 2(c) shows the k^2 -tree representation for this matrix with $k = 2$.

The adjacency matrix is represented by two bit arrays, called T (Tree) and L (Leaves). T stores all the bits of the k^2 -tree but not those in the leaves. The bit array L stores the information of leaves. The bitmaps for the k^2 -tree in Figure 2(c) are: $T : 1111 - 1010 - 1011 - 0010 - 1010$, and $L : 0101 - 0010 - 0010 - 0100 - 0001 - 0100 - 0010 - 0001$.

Navigation in the tree is implemented through the *rank* and *select* operations over bitmaps. Consider a bitmap B , $rank_1(i)$ returns the number of 1s until position i in B , and $select_1(i)$ returns the position of the i -th 1 in B . To obtain the child of a node we use function $child_i(x) = rank_1(T, x) \times k^2 + i$, where i is the i -th child of node x in T (where the first position is 0). As an illustration, to find the first child of the fourth node in T we evaluate $child_0(3) = rank_1(T, 3) \times 2^2 + 0$, since $rank_1(T, 3) = 4$ (there are



$$CWR(A, B, C) \begin{cases} = 0 & \text{Points } A, B \text{ and } C \text{ are collinear} \\ > 0 & \text{Point } C \text{ is to the left of segment } \overline{AB} \\ < 0 & \text{Point } C \text{ is to the right of segment } \overline{AB} \end{cases}$$

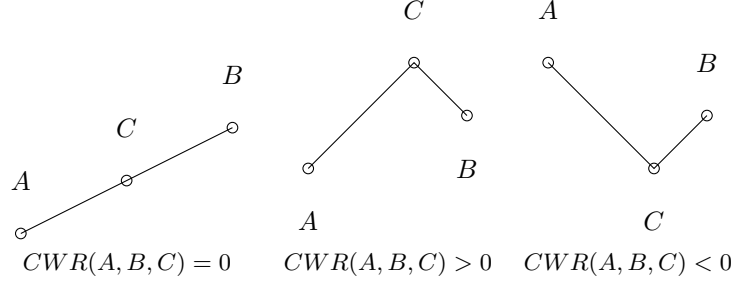


Figure 3: Clockwise rotation function

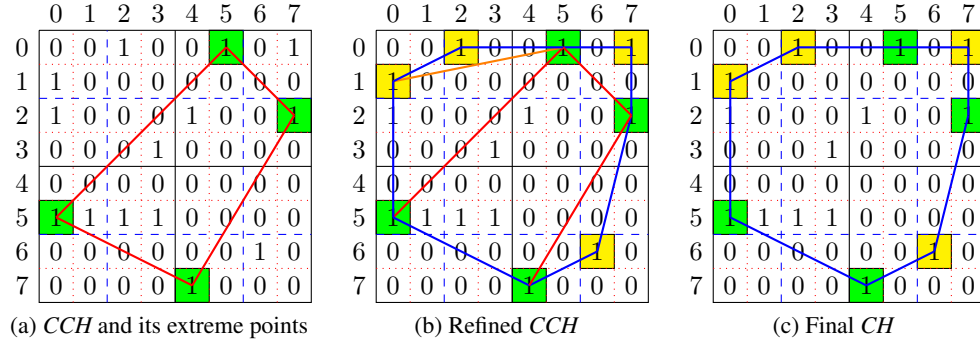


Figure 4: Computation of the Convex Hull (CH)

Definition 2 (Distance). Given three points $A(x_1, y_1)$, $B(x_2, y_2)$ and $C(x_3, y_3) \in \mathbb{R}^2$, the *distance* between C and segment AB is defined as follow:

$$Dist(C, \overline{AB}) = \frac{(x_2 - x_1)(y_3 - y_1) - (y_2 - y_1)(x_3 - x_1)}{\sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}}$$

4. Algorithms to Compute the Convex Hull

Our algorithms to compute the convex hull over a k^2 -tree structure storing points are inspired in the work presented in [8] that implements the convex hull over points stored in a R-tree [9]. Thus, we start by locating the four extreme points that will form, initially, the *Candidate Convex Hull* (CCH). The CCH will be refined by locating, for each edge, the farthest external point to the edge, which will be added into the final CH. For illustration purposes we will use the matrix in Figure 4(a) where the candidate convex hull has four initial extreme points. Figure 4(b) shows the refining process of the CCH, and Figure 4(c) shows the final CH.

4.1. Identification of Extreme Points in the k^2 -tree

The algorithms take advantage of the binary partition and hierarchical organization of the k^2 -tree to access portions of the structure (or the matrix). We also use a data

structure call $Node\langle h, i, x, y \rangle$ that identifies a node of the k^2 -tree with some extra information, h indicates the height of the node, i is the position of the node in the bitmap T (intermediate node) or L (a leaf), x and y are the coordinates of the superior left corner of the quadrant defining the subspace of the node. With this information is possible to obtain the four coordinates (rectangle) that defines the quadrant of a node and the coordinates of its children.

Algorithm 1: Searching for the extreme point $getMin_y$

```

input :  $k^2$ -tree  $K$ 
output: Point  $(x_e, y_e)$ 

/*Coordinate with the greatest value in coordinate  $y$  */
1  $x_e = y_e \leftarrow GetCoordinate(K.n - 1);$ 
/*Gets the root node of the  $k^2$ -tree */
2  $Node\ b \leftarrow GetNode(K);$ 
/*Insert the node into the min heap  $H$  ordering according coordinate
 $y$  of nodes */
3  $Insert(H, b);$ 
4 while  $H \neq \emptyset$  do
5    $b \leftarrow Delete(H);$ 
6   if  $b.y \leq y_e$  then
7     if  $internalNode(b, K)$  then
8       foreach child  $ch$  of  $b$  such that  $Access(K, ch) = 1$  do
9         /*Obtain the node for child  $ch$  */
10         $Node\ b_1 \leftarrow GenerateNode(ch, b, K);$ 
11        if  $Empty(H)$  OR  $Top(H).y \geq b_1.y$  then
12           $Insert(H, b_1);$ 
13      else
14        if  $b.y < y_e$  then
15           $y_e \leftarrow b.y;$ 
16           $x_e \leftarrow b.x;$ 
17      else
18      return  $x_e, y_e$ 
19 return  $x_e, y_e$ 

```

Algorithm 1 obtains the point with the lowest value in coordinate y in the k^2 -tree structure K . The other extreme points can be computed similarly. In Figure 4(a) this extreme point will be any of the 1s in row 0, since all of them have share the same minimum value in coordinate y . Initially, the algorithm assigns as extreme point the one in the inferior right corner (Line 1), which is a bad solution, since this point has the greatest value in coordinate y .

The algorithm uses a min heap H to organize nodes according the values of their coordinate y . In this way, the nodes of the k^2 -tree with lower values in coordinate y will be close to the root of the heap. The heap is processed between Lines 4-17 as follows, if the value in coordinate y of node b (the root of heap H) is greater than the already computed best solution y_e , then all the nodes in H are discarded since none of them will improve the solution (Line 6). Otherwise, we have two options:

- Node b is an internal node, this is, a node with children, in which case, all the children of node b whose value in coordinate y are lower than the value of coordinate y of node in the root of H are inserted into the heap (Lines 8-11).

Note that the algorithm only considers children with a 1 in the k^2 -tree (function $Access(\cdot, \cdot)$).

- Node b is a leaf, then we can improve the solution if the value of its coordinate y is lower than the current best solution (y_e) (Line 13).

4.2. Algorithm CHk^2

Algorithm 2 gets the Convex Hull (CH) from a given k^2 -tree structure K . It first obtains the four extreme points from the structure and inserts them in a stack (Line 2). The four points are ordered according their position as the extreme point at the north (N), south (S), east (E), and west (W). They form the vertices of the initial Candidate Convex Hull (CCH) with edges $(W, N), (N, E), (E, S), (S, W)$ (see Figure 4(a)), and they will also be part of the vertices of the final convex hull CH (see Figure 4(c)).

Thus, Algorithm 2 starts computing the CCH and recursively refines the CCH until obtaining the final CH . The refining process consists in searching, for every edge $a = (v_1, v_2)$ of the CCH , the farthest external point pa to a (the point that is not into CCH). This process is repeated until it is not possible to delete/add more points to the CCH . For instance, in Figure 4(a) for edge $a = (E, S)$ that goes from position $(2, 7)$ to $(7, 5)$, pa is the 1 located in position $(6, 6)$. Thus, the algorithm deletes edge $a = (v_1, v_2)$ and inserts edges (E, pa) and (pa, S) (Figure 4(b)). To locate the farthest point for an edge $a = (v_1, v_2)$ the algorithm searches from the root until the leaves of the k^2 -tree structure K visiting only relevant nodes of the structure (lines 7-19). The nodes that are visited are those that have points that could refine the corresponding edge $a = (v_1, v_2)$. This decision is taken based on the orientation of edge $a = (v_1, v_2)$ (Line 10). Table 1 shows the possible orientation of edges (column 1), and the form of the rectangle that will be explored (only if it has points) where p is a representative point of the node's rectangle. For instance, consider edge (E, S) in Figure 4(a) with orientation code 4 (column 2 in Table 1). In this case, the node that will be visited is the one described by rectangle in column 3, and that corresponds to the node identifying the inferior right corner of the matrix in Figure 4(a). This node has effectively a point in position $(6, 6)$ located in the exterior of edge $a = (v_1, v_2)$.

The *clockwise rotation* function (see Definition 1) $CWR(v_1, v_2, p)$ is used to determine if a node has a relevant point to refine an edge of the CCH (Line 18). In fact, if the value of the function is lower than 0 the node is inserted into a max-heap H , which is organized according to the distance between edge $a = (v_1, v_2)$ and point p .

When the algorithm finds a leaf with point pa (Line 13), the algorithm goes to Line 20 and inserts vertex v_2 into the CCH and continues refining edge (v_1, pa) . If the while cycle (Lines 11-19) ends because the stack is empty, it means that edge $a = (v_1, v_2)$ cannot be refined, and then vertex v_1 is inserted into the CH and will be part of the final solution (Lines 23-26).

4.3. Algorithm CHk^2t

Algorithm CHk^2t improves the performance of Algorithm CHk^2 (Algorithm 2) by implementing a translation of the edge that is being refined. This allows reducing the number of nodes that need to be explored to find the farthest point to the edge. We illustrate this with an example. Consider that we are refining edge a that goes from west to north in Figure 5(a) (with vertices $(5, 0)$ to $(0, 5)$). The quadrants (nodes) in grey that are outside of the edge will be explored by Algorithm 2, since they are nodes

Algorithm 2: Algorithm CHk^2

```

input :  $k^2$ -tree  $K$ 
output:  $CH$ 

/*CCH: candidate convex hull, CH: final convex hull */
1 Stack  $CCH, CH$ ;
/*Insert the extreme points  $W, S, E, N$  into Stack  $CCH$  */
2  $Put(CCH, GetExtremePoints(K))$ ;
/*Create structure  $Node$  from the  $k^2$ -tree */
3  $Node\ b \leftarrow GetNode(K)$ ;
/*MaxHeap is organized according to the greatest distance between the
point of the node (rectangle) and the edge of the  $CCH$  */
4  $Heap\ H$ ;
5  $Vertex\ v_1 \leftarrow Pop(CCH)$ ;
6  $Vertex\ v_2 \leftarrow Pop(CCH)$ ;
7 while  $Empty(CCH) = False$  do
8    $H \leftarrow \emptyset$ ;
9    $Insert(H, b, -\infty)$ ;
10   $ort \leftarrow GetOrientation(v_1, v_2)$ ;
11  while  $Empty(H) = False$  do
12     $Node\ b_1 \leftarrow Delete(H)$ ;
13    if  $Leaf(b_1, K)$  then
14       $\lfloor$  Break;
15    foreach  $child\ ch\ of\ b_1\ such\ that\ Access(K, ch) = 1$  do
16       $Node\ bb \leftarrow GenerateNode(ch, b_1, K)$ ;
17      /*Get  $p$  according Table 1 */
18       $Vertex\ p \leftarrow SelectPointRectangle(bb, ort, K)$ ;
19      if  $CWR(v_1, v_2, p) < 0$  then
20         $\lfloor Insert(H, bb, Dist(p, \overline{v_1v_2}))$ ;
21  if  $Leaf(b_1, K)$  then
22     $Push(CCH, v_2)$ ;
23     $v_2 \leftarrow SelectPointRectangle(b_1, ort, K)$ ;
24  else
25     $Push(CH, v_1)$ ;
26     $v_1 \leftarrow v_2$ ;
27     $v_2 \leftarrow Pop(CCH)$ ;
27 return  $CH$ 

```









Edge	Orientation code	Rectangle with point (p)
	1	
	2	
	3	
	4	

Table 1: Orientation of the edges of the CCH

with points, and therefore, one of them contains the farthest point pa to a . Algorithm CHk^2t will not explore all the quadrants but only the one formed with coordinates $(0, 0), (1, 1)$. Since this quadrant has the maximum distance $maxd$ between point p (coordinate $(0, 0)$) and edge a (case 1 in Table 1). This $maxd$ is a superior bound to point pa that has to exist in this quadrant, being the inferior bound the coordinate $(1, 1)$. Thus, edge a is virtually translated until intersecting coordinate $(1, 1)$ (see Figure 5(b)). All the points between a and the new edge a' are discarded and the searching of the new point to the CH is focalized in the first quadrant.

Algorithm 3 calculates the vertices of the new virtual edge a' for a given edge a . First the algorithm obtains the quadrant that has the maximum distance between edge a and point p (Line 1). Then, it obtains the opposite point p' that corresponds to the inferior bound to find point pa , and together with the parameters of the straight line equation, it obtains the new edge $a'(v'_1, v'_2)$ (Lines 3-9).

Algorithm CHk^2t consists in Algorithm CHk^2 (Algorithm 2) with the following modifications, the insertion of line $(v'_1, v'_2) \leftarrow MoveEdge(K, b_1, v_1, v_2, p)$, before line 15, and changing the condition of the *If* in line 18 by $CWR(v'_1, v'_2, p) < 0$. It is important to mention that the new line $a'(v'_1, v'_2)$ is only used to discard nodes.

5. Experimental Evaluation

In this section we report experimental evaluation of Algorithms CHk^2 and CHk^2t . We compared our algorithms in terms of execution times and space, with a naive algorithm called GSk^2 that obtain the points from the k^2 -tree structure and then compute the convex hull applying the Graham Scan algorithm [15]. We use both real and synthetic data sets.

Algorithms CHk^2 , CHk^2t and GSk^2 were implemented in C++ programming language with library `libcds` that allows the use of compact data structures. Programs were compiled using gcc 4.8.4. All the experiments were executed on an Intel(R) Core(TM)

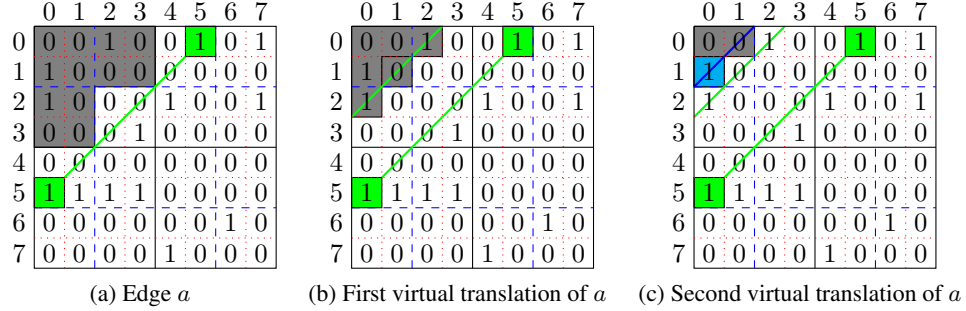


Figure 5: Translation of edges when refining an edge of the *CCH*

Algorithm 3: Virtual translation of edge a *MoveEdge*

```

input :  $k^2$ -tree  $K$ , Node  $m$ , Edge  $a$ , Point  $p$ 
output: Edge  $(v'_1, v'_2)$ 

/*Edge  $a = (v_1, v_2)$  */
/* $Q$  is the child of node  $n$  with  $Access(K, Q) = 1$  and with maximum value of
   $Dist(p, v_1 v_2)$  */
1  $Q \leftarrow GetNode(m, p)$ ;
  /* $p'$  is the opposite extreme point of  $Q$  with respect to  $p$  and edge  $a$  */
2  $p' \leftarrow GetPoint(Q, p, a)$ ;
3 if  $CRW(v_1, v_2, p') < 0$  then
4    $m \leftarrow \frac{v_2.y - v_1.y}{v_2.x - v_1.x}$ ;
5    $b \leftarrow p'.y - m \times p'.x$ ;
6    $y_1 \leftarrow m \times v_1.x + b$ ;
7    $y_2 \leftarrow m \times v_2.x + b$ ;
8    $v'_1 = Vertex(v_1.x, y_1)$ ;
9    $v'_2 = Vertex(v_2.x, y_2)$ ;
10 return  $(v'_1, v'_2)$ ;

```

computer with i7-4770 CPU, @ 3.40 GHz 3.40 GHz, 4 GB of RAM, and under Linux operative system, with Ubuntu 14.04.2 LTS distribution, and kernel 3.16.0-3-generic.

5.1. Data Sets for the Experimentation

We consider real and synthetic data sets. The former correspond to points representing roads from California (RCA), tiger streams (RTS) and census blocks (RCTB)¹. The latter were generated with the *GNU Octave* tool². Tables 2 and 3 show the size of matrices of synthetic and real data sets, respectively. For instance, for 1 million of synthetical points (see Tables 2) the sizes of matrices were $2^{10} \times 2^{10}$, $2^{12} \times 2^{12}$ and $2^{14} \times 2^{14}$. Figure 6 shows the graphical representation of synthetical data sets with different distributions. Figure 7 shows the graphical representation of real data sets.

¹Data sets are available at <http://chorochronos.datastories.org/>.

²<https://www.gnu.org/software/octave/>

Number of points (Millions)	Matrix size					
	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}	2^{15}
1	•		•		•	
2		•	•		•	
4			•	•	•	
8			•		•	•

Table 2: Synthetic data sets (Gauss and uniform distribution)

Set	Number of points	Matrix size			
		2^{11}	2^{12}	2^{13}	2^{14}
RCA	2,224,727	•	•	•	•
RTS	194,971	•	•	•	•
RCTB	556,696	•	•	•	•

Table 3: Real data sets

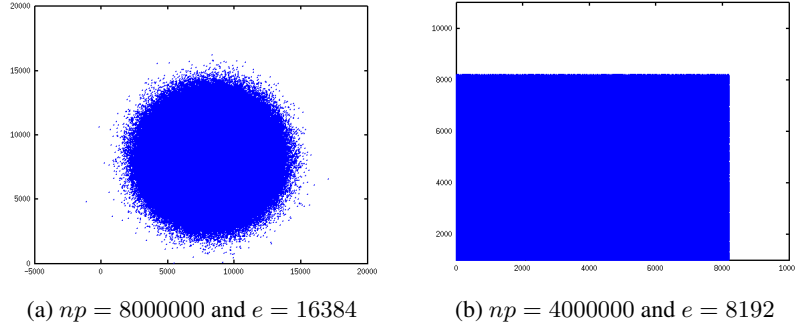


Figure 6: Graphical representation of synthetical data sets: (a) Gauss distribution, (b) uniform distribution

5.2. Results of the Experimentation

5.2.1. Synthetical Data

Figure 8 shows the execution time of the algorithms CHk^2 , CHk^2t and GSk^2 over the synthetical data sets with Uniform distribution (U) and Gauss distribution (G) considering different matrices sizes. We can appreciate that algorithms CHk^2 and CHk^2t have better execution times with respect to the GSk^2 algorithm. For instance, in the case of data sets of 1 million points and Gauss distribution (see Figure 8(a)), Algorithm CHk^2 requires between 0.0062% and 0.082% of the execution time required by Algorithm GSk^2 .

Moreover, Algorithms CHk^2 and CHk^2t are very similar in terms of execution times, especially over data sets with Uniform distribution (see $CHk^2(U)$, and $CHk^2t(U)$ in Figure 8). However, Algorithm CHk^2t runs better than algorithm CHk^2 over points that are grouped in some specific areas of the subspace. For instance, CHk^2t reduces in a 39% of the execution time of algorithm CHk^2 (Figure 8(d)). The main advantage of the CHk^2t algorithm is that it avoids processing quadrants or regions that, even though have points, they are not relevant to the computation of the convex hull.

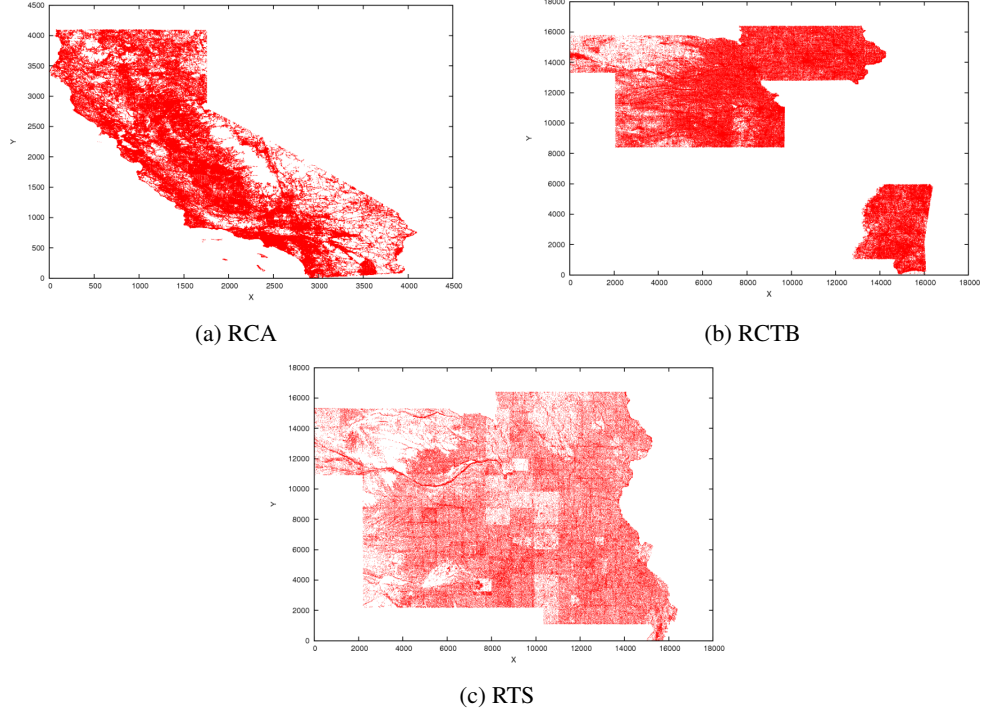


Figure 7: Graphical representation of real data sets

Figure 9 shows the data space requirements by each of the algorithms. Algorithms CHk^2 and CHk^2t requires less space for every data set than algorithm GSk^2 . For instance, considering 2 millions of points with Gauss distribution (see Figure 9(b)), CHk^2 needs only 0.02486%, 0.0055 % and 0.0041% than the space required by algorithm GSk^2 for matrices sizes of 2.048, 4.096 and 16.384, respectively.

5.2.2. Real Data Sets

Figures 10 and 11 show respectively the execution times and data space requirements for the algorithms over real data sets. As for the case of synthetical data, the algorithms perform very well for execution time over real data sets. For instance, experiments show that algorithm CHk^2 requires between 0.114% and 0.603% of the time required by algorithm GSk^2 (see Figure 10(c)). With respect to space requirement (see Figure 11) we can say that algorithm CHk^2 requires between 0.0204% and 0.9234% of additional memory than algorithm GSk^2 . This without considering the additional space required to represent the compact data structure k^2 -tree which is not relevant.

6. Conclusions

We presented two Algorithms CHk^2 and CHk^2t to compute the convex hull polygon over a set of points stored in a compact data structure called k^2 -tree. The algo-

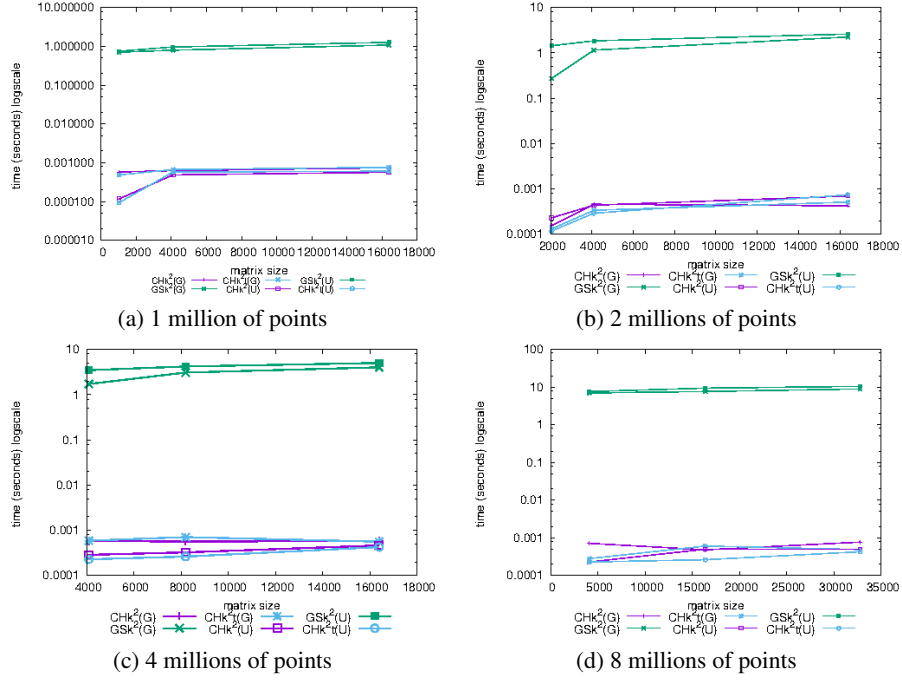


Figure 8: Execution times of algorithms CHk^2 , CHk^2t and GSk^2 over synthetic data sets

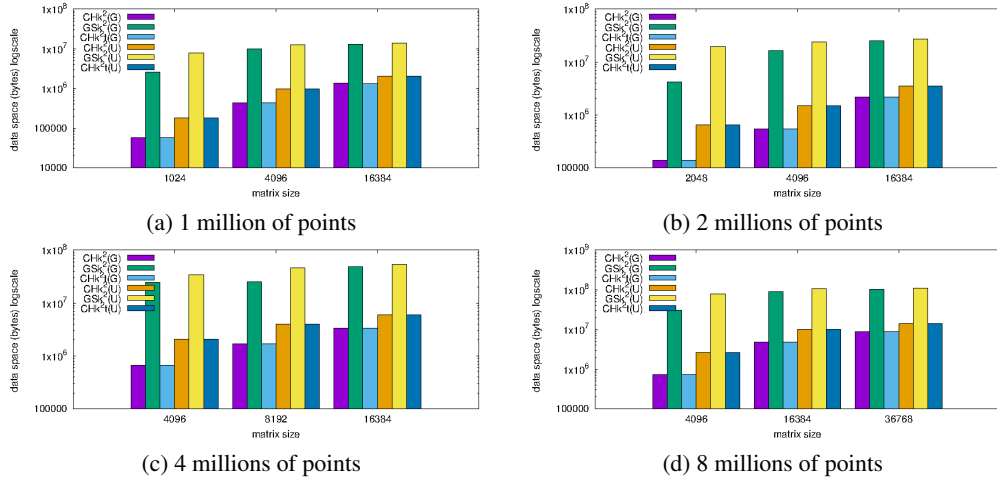


Figure 9: Data space of algorithms CHk^2 , CHk^2t and GSk^2 for synthetic data sets

algorithms process the data directly from the compact data structure without uncompressing the data. They use the known properties of the k^2 -tree to compute the extreme points which are the started point to the algorithms, and take advantage of the binary

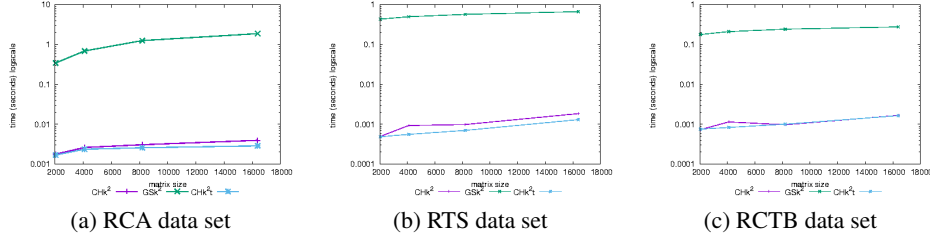


Figure 10: Execution times of algorithms CHk^2 , CHk^2t and GSk^2 over real data sets

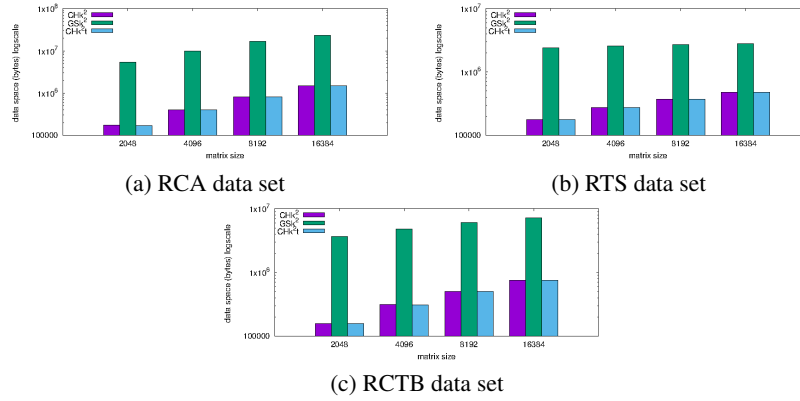


Figure 11: Data space of algorithms CHk^2 , CHk^2t and GSk^2 for real data sets

partition and hierarchical organization of the k^2 -tree to access portions of the structure.

Compact data structures have been used for representing several data types. For instance in [17, 10] compact data structures are used to represent graphs of the World Wide Web. In [18, 19, 20] compact data structures allow the representation of documents in the context of information retrieval. Also, in [13, 21] they are used to improve query efficiency in GIS (Geographical Information Systems). As far as we know algorithms CHk^2 and CHk^2t are the first proposal to compute the convex hull over a compact data structure with data representing in main memory.

We report experimentation over real and synthetical data sets and compare our solution with a naive implementation, called GSk^2 , that computes the convex hull of a set of points taken from a k^2 -tree structure by using the Graham Scan algorithm [15]. Experimental results show that the CHk^2 and CHk^2t algorithms run at least two order of magnitude faster than the GSk^2 algorithm, and require less than 1% of extra storage space than the GSk^2 algorithm in all data sets. Moreover, Algorithm CHk^2t runs better than algorithm CHk^2 over points that are grouped in some specific areas of the subspace. Also, algorithm CHk^2t avoids processing quadrants or regions with points that are not relevant to the convex hull.

References

- [1] M. De Berg, M. Van Kreveld, M. Overmars, O. Schwarzkopf, Computational geometry, Springer, 2000.
- [2] J. Sklansky, Finding the convex hull of a simple polygon, *Pattern Recogn. Lett.* 1 (2) (1982) 79–83.
- [3] R. Graham, F. Yao, Finding the convex hull of a simple polygon, *Journal of Algorithms* 4 (4) (1983) 324–331.
- [4] F. Preparata, S. Hong, Convex hulls of finite sets of points in two and three dimensions, *Commun. ACM* 20 (2) (1977) 87–93.
- [5] R. Liu, B. Fang, Y. Tang, J. Wen, J. Qian, A fast convex hull algorithm with maximum inscribed circle affine transformation, *Neurocomputing* 77 (1) (2012) 212–221.
- [6] J. Sander, M. Ester, H. Kriegel, X. Xu, Density-based clustering in spatial databases: The algorithm gdbscan and its applications, *Data Min. Knowl. Discov.* 2 (2) (1998) 169–194.
- [7] M. Mcqueen, G. Toussaint, On the ultimate convex hull algorithm in practice, *Pattern Recogn. Lett.* 3 (1) (1985) 29–34.
- [8] C. Böhm, H. Kriegel, Determining the convex hull in large multidimensional databases, in: *Proceedings of the Third International Conference on Data Warehousing and Knowledge Discovery*, 2001, pp. 294–306.
- [9] A. Guttman, R-trees: A dynamic index structure for spatial searching, in: *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, 1984, pp. 47–57.
- [10] N. Brisaboa, S. Ladra, G. Navarro, k2-trees for compact web graph representation, in: *Proceedings of the 16th International Symposium on String Processing and Information Retrieval*, 2009, pp. 18–30.
- [11] R. Raman, V. Raman, S. Rao, Succinct dynamic data structures, in: *Algorithms and Data Structures*, Vol. 2125 of *Lecture Notes in Computer Science*, 2001, pp. 426–437.
- [12] H. Plattner, A. Zeier, *In-Memory Data Management: An Inflection Point for Enterprise Applications*, 1st Edition, Springer Publishing Company, Incorporated, 2011.
- [13] N. Brisaboa, M. Luaces, G. Navarro, D. Seco, Space-efficient representations of rectangle datasets supporting orthogonal range querying, *Information Systems* 38 (5) (2013) 635–655.
- [14] S. Durocher, H. El-Zein, J. Munro, S. Thankachan, Low space data structures for geometric range mode query, *Theoretical Computer Science* 581 (2015) 97–101.

- [15] R. Graham, An efficient algorithm for determining the convex hull of a finite planar set, *Information processing letters* 1 (4) (1972) 132–133.
- [16] C. Bradford, D. Dobkin, H. Huhdanpaa, The quickhull algorithm for convex hulls, *ACM Transactions on Mathematical Software* 22 (4) (1996) 469–483.
- [17] F. Claude, G. Navarro, A fast and compact web graph representation, in: *Proceedings of the 14th International Symposium on String Processing and Information Retrieval (SPIRE)*, Vol. 4726 of *Lecture Notes in Computer Science*, 2007, pp. 105–116.
- [18] F. Claude, G. Navarro, Practical rank/select queries over arbitrary sequences, in: *Proceedings of the 15th International Symposium on String Processing and Information Retrieval*, 2009, pp. 176–187.
- [19] G. Navarro, Spaces, trees, and colors: the algorithmic landscape of document retrieval on sequences, *ACM Computing Surveys* 46 (4) (2014) 52:1–52:47.
- [20] G. Navarro, K. Sadakane, Fully functional static and dynamic succinct trees, *ACM Transactions on Algorithms* 10 (3) (2014) 16:1–16:39.
- [21] G. De Bernardo, S. Álvarez-García, N. Brisaboa, G. Navarro, O. Pedreira, Compact queriable representations of raster data, in: *String Processing and Information Retrieval - 20th International Symposium*, 2013, pp. 96–108.