

# cKdtree: a Compact Kdtree for Spatial Data<sup>\*</sup>

Gilberto Gutiérrez<sup>1</sup>, Rodrigo Torres-Avilés<sup>2</sup> and Mónica Caniupán<sup>2,\*</sup>

<sup>1</sup>*Departamento Ciencias de la Computación y Tecnologías de Información, Universidad del Bío-Bío, Chillán, Chile*

<sup>2</sup>*Departamento Sistemas de Información, Universidad del Bío-Bío, Concepción, Chile*

## Abstract

We introduce cKd-tree, a compact data structure designed to represent a Kd-tree index efficiently. The structure cKd-tree is essentially an encoding of the spiral code sequence of points within an implicit Kd-tree (iKd-tree) using Directly Addressable Codes (DACs). The unique feature of cKd-tree lies in its ability to perform spiral encoding and decoding of points by relying solely on knowledge of their parent points within the iKd-tree. This inherent property, combined with DACs' direct access capability to sequence elements, enables cKd-tree to traverse and explore the tree while decoding only the nodes relevant to queries. We compare cKd-tree with iKd-tree and  $k^2$ -tree data structures evaluating compression efficiency and execution time of *point queries* and the *range queries*. cKd-tree achieves a compression ratio comparable to that of  $k^2$ -tree, approximately 70%, demonstrating heightened efficiency, particularly in scenarios characterized by sparse data. Additionally,  $k^2$ -tree exhibits superior performance in querying individual points, whereas cKd-tree outperforms in the context of aggregate data queries, such as range queries.

## Keywords

Compression, indices, spatial data, spatial points, spatial queries

## 1. Introduction

Several multidimensional data structures have been devised for the storage and efficient querying of set of points in both primary and secondary memory. Noteworthy among these structures are the Kd-tree [1] and Quadtree [2], which enable the execution of single and aggregate queries without necessitating a full scan of the entire dataset. A **Kd-tree** is a hierarchical structure (binary tree), designed for recursive division of multi-dimensional space [1]. Within this structure, each node contains data representing a  $d$ -dimensional point in the space. When dealing with static sets, data structures can be focused on implementing query operations, which do not alter set size. This results in more cost-effective implementations in terms of storage, as the algorithm proposed in [3] for a static Kd-tree. This static Kd-tree is implicitly stored in an array, without using pointers, making it occupy less storage while maintaining navigation efficiency comparable to its dynamic counterpart.

---

AMW 2024: 16th Alberto Mendelzon International Workshop on Foundations of Data Management, September 30th–October 4th, 2024, Mexico City, Mexico

<sup>\*</sup>This work was funded by ANID Grant 1230647, ALBA group code 2130591 GI/VC, Project INES I+D 22-14 and Project 2130520 IF/R.

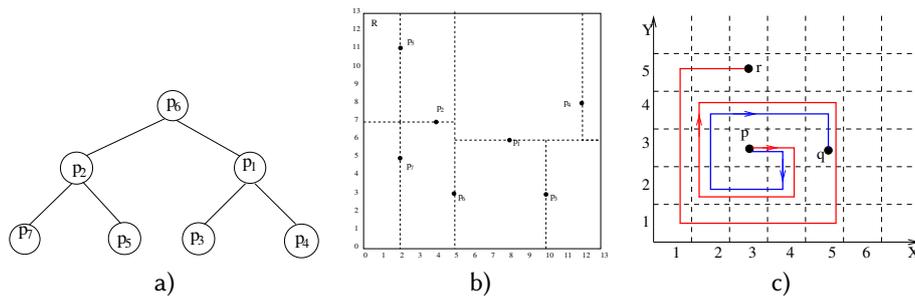
<sup>\*</sup>Corresponding author.

✉ ggutierr@ubiobio.cl (G. Gutiérrez); rtorres@ubiobio.cl (R. Torres-Avilés); mcaniupan@ubiobio.cl (M. Caniupán)

🆔 0000-0001-6059-1453 (G. Gutiérrez); 0000-0001-8286-3712 (R. Torres-Avilés); 0000-0003-1543-2378 (M. Caniupán)



© 2024 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).



**Figure 1:** a) iKd-tree representation for a set of points  $S$ . b) Partition generated by the iKd-tree for the points from  $S$ . c) Spiral encoding of a set of points in  $\mathbb{N}^2$ .

We refer to a balanced Kd-tree represented in an array as iKd-tree [4]. The construction process of iKd-tree is described in [3]. As an illustration, Figure 1 represents an iKd-tree constructed from a set of points  $S = \langle p_1(8, 6), p_2(4, 7), p_3(10, 3), p_5(2, 11), p_6(5, 3), p_4(12, 8), p_7(2, 5) \rangle$  with a resulting arrangement stored in array  $Q = [(2, 5), (4, 7), (2, 11), (5, 3), (10, 3), (8, 6), (12, 8)]$ . The algorithm ensures the creation of a balanced binary tree, with a tree depth of  $\log_2(n)$ , as depicted in Figure 1a). Navigating or exploring the iKd-tree is straightforward. Generally, the position of the root node within the subtree, defined by the initial positions  $li$  and final positions  $ls$  of array  $Q$ , is situated at position  $\lfloor \frac{li+ls}{2} \rfloor$ .

In this work, we introduce a compact version of the Kd-tree designed for storing a set of static points  $S \subseteq \mathbb{N}^2$ , named the cKd-tree. We focused our attention on the Kd-tree given its prominence as one of the most widely employed structures for indexing spatial and geometric data [5].

## 2. Related Work

The BSP-tree, very similar to the Kd-tree, is a multidimensional data structure that shares the concept of recursively partitioning space using  $d - 1$  dimensional hyperplanes. These hyperplanes in the BSP-tree need not be iso-oriented. In contrast to the Kd-tree, where partitions alternate between different dimensions, the BSP-tree adapts its partitions based on the distribution of objects within the space or subspace. Another important data structure is the Quad-tree [6]. This structure, like the Kd-tree, employs recursive division of space or subspace through iso-oriented hyperplanes. However, a key distinction lies in the fact that each internal node of the Quad-tree typically has  $2^d$  children. Unlike Kd-trees, these structures are predominantly designed for dynamic object sets and are commonly implemented using pointers.

Over the past few decades, there has been a pronounced surge in the exploration and development of compact data structures (CDS). These structures compress static data, minimizing storage requirements and facilitating data processing without the need for prior decompression allowing the processing of data directly in main memory.

Compact data structures offer an alternative for representing points, exemplified by the  $k^2$ -tree [7]. In this structure, points are derived from a binary matrix  $A$ , where a point  $p(x, y)$  in

the set is symbolized by a stored value of 1 in the cell  $A[x, y]$ . The  $k^2$ -tree allows to store static sets of points without relying on pointers, making it a foundational structure for comparative analysis in our study. This compact data structure has been used in diverse scenarios such as, graph representations [7], raster data [8], and points in  $\mathbb{N}^2$  [9].

### 3. Our Proposal: The cKdtree

cKd-tree is built upon the implicit version of a Kd-tree and leverages spiral encoding to represent a set of points in a multidimensional ( $d$ -dimensional) space. Moreover, cKd-tree employs DACs encoding for a sequence of integers [10]. This dual encoding approach enables the structure to store information more efficiently.

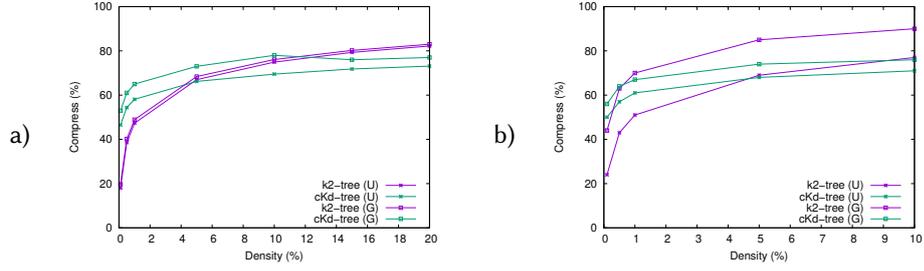
The **spiral encoding method** involves assigning a positive integer to a point  $p$  in  $\mathbb{N}^2$  based on another point  $q$ . Formally, considering points  $p(x, y)$  and  $q(x, y)$  from the set  $S \subseteq X \times Y$ , for  $X, Y \subseteq \mathbb{N}$  with dimensions  $|X|$  and  $|Y|$ , the spiral encoding of  $q$  with respect to  $p$  is denoted as a function  $sCode_p : \mathbb{N}^2 \rightarrow \{0, \dots, |X| \cdot |Y|\}$ . In this representation,  $sCode_p(q)$  represents the distance, measured in the number of cells, from the cell of  $p$  to the cell of  $q$  following a spiral path with the origin at  $p$  (cell 0). Figure 1c) illustrates the spiral paths used to encode points  $q$  and  $r$  starting from the reference point  $p$ . In this example,  $sCode_p(q)$  is equal to 10, and  $sCode_p(r)$  is equal to 22. The function  $sDecode_p(q) : \{0, \dots, |X| \cdot |Y|\} \rightarrow \mathbb{N}^2$  facilitates retrieving the coordinates of a point  $q$  from its spiral code with respect to the point  $p$ .

The **acronym DACs** [10] stands for Directly Addressable Codes, which represent a variable-length encoding of sequences of non-negative integers, typically arrays. This method divides the binary representation of the integers of the array into blocks of  $b$  bits, adding a bit in the most significant bit of the chunk, set to 0 when the chunk holds the most significant bits of the integer, and 1 otherwise. Then, the chunks of size  $b + 1$  are grouped together, in order, and each group is called a *level*. With this construction, DACs keeps direct access to any element of the encoded sequence without the need of any sampling method, therefore without using asymptotically any extra space.

Although it can be used a fixed block size  $b$  for DACs, it is possible to choose a different block size at each level  $l$  ( $b_l$ ). This flexibility can be advantageous to achieve specific goals, such as optimizing compression.

A cKd-tree is represented as a tuple  $\langle Seq, p \rangle$ , where  $Seq$  is a sequence of integers encoded using DACs, representing an iKd-tree, and  $p \in S$  denotes the root of the tree with  $S \subseteq \mathbb{N}^2$  a set of points. The sequence  $Seq$  is derived from the DACs encoding through a spiral codification of the points in the iKd-tree.

The algorithms for the construction of the cKd-tree together with the algorithms for processing queries on sets of points are presented in [11]. We addressed two fundamental queries. The *point query* which determines whether a given point  $q$  is part of the set of points represented by the cKd-tree, and the *range query* that retrieves all points stored in the cKd-tree within the bounds of the specified iso-oriented rectangle for the query range.



**Figure 2:** Data Compression Percentages on space of size  $16,384 \times 16,384$  and  $32,768 \times 32,768$ .

## 4. Experiments

We compare cKd-tree against iKd-tree and  $k^2$ -tree, with the latter being known for its efficiency in compression and query execution. The iKd-tree and cKd-tree structures were implemented in C++, while we utilized the  $k^2$ -tree implementation from [12]<sup>1</sup>.

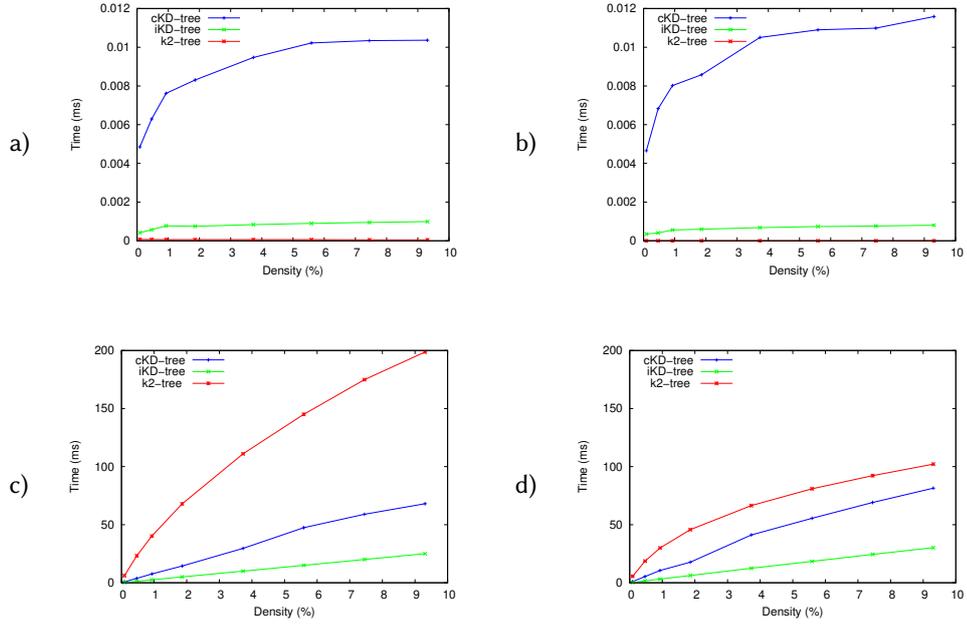
We utilized synthetic two-dimensional datasets comprising points. The dataset sizes ranged from 250,000 to 100,000,000, distributed according to both Uniform and Gaussian distributions in space. The datasets were generated within spaces (matrices) of dimensions  $2^{14} \times 2^{14}$  and  $2^{15} \times 2^{15}$ .

Figure 2 shows the compression percentages for both cKd-tree and  $k^2$ -tree compact data structures in two space scenarios (sizes  $16,384 \times 16,384$  and  $32,768 \times 32,768$ , respectively). The structures exhibit competitive compression percentages ranging from approximately 18% to 90%. Both structures improve the compression performance when density increases. At higher densities ( $\geq 12$ ) and for Gaussian distribution (G in the charts),  $k^2$ -tree marginally outperforms cKd-tree. Conversely, at lower densities ( $\leq 6$ ), cKd-tree outpaces  $k^2$ -tree. The enhanced compression performance of cKd-tree can be attributed to the spatial proximity of points. In such scenarios, the resulting spiral codes of the points are generally smaller, leading to a highly compressible sequence of integers by DACs.

Figure 3 a) and b) displays the execution times of three structures in solving the point query for Uniform and Gaussian distributions of points. As expected,  $k^2$ -tree outperforms cKd-tree by a significant margin (about 200 times faster), and iKd-tree also performs faster (11 times) than cKd-tree. The advantage of  $k^2$ -tree is attributed to its  $O(\log_2 m)$  query execution time for one point, while iKd-tree avoids the cost of decompressing points stored with DACs.

Figure 3 c) and d) illustrates the execution times of range query over a matrix of size  $32,768 \times 32,768$  for the range of  $3,276 \times 3,276$  over Uniform and Gaussian distribution. Notably, cKd-tree significantly outperforms  $k^2$ -tree achieving a speedup of approximately 4.7 times. And, as expected, iKd-tree outperforms cKd-tree in range query, albeit to a lesser extent, with an average speedup of about 3 times for range queries of size  $3,276 \times 3,276$ . For the Gaussian distribution  $k^2$ -tree has better results, but it is still outperformed by cKd-tree, in a lesser extent. The improved performance of  $k^2$ -tree in this distribution is attributed to its optimal operation

<sup>1</sup>Available at <https://github.com/simongog/sdsl-lite>



**Figure 3:** Point query and Range query performance on a  $32,768 \times 32,768$  matrix. a) Point query with Uniform distribution, b) Point query with Gaussian distribution, c) Range query with Uniform distribution and d) Range query with Gaussian distribution.

when points are concentrated in specific areas of space. This characteristic makes  $k^2$ -tree sensitive to point distribution, whereas both iKd-tree and cKd-tree exhibit similar behavior across different distributions.

## 5. Conclusions

We present a novel compact data structure named cKd-tree designed for representing an implicit static Kd-tree. Through experimental evaluations against  $k^2$ -tree and iKd-tree, our findings reveal that cKd-tree achieves compression rates ranging from 45% to 77% depending on the density. The percentage of compression is higher when the density increases.

In terms of execution time for point queries, cKd-tree exhibits a slower performance compared to  $k^2$ -tree, as expected. However, cKd-tree showcases superior execution times for range queries, significantly outperforming  $k^2$ -tree, especially in scenarios with Uniform distribution where cKd-tree is approximately 4.7 times faster. This continues to widen with increasing data density.

cKd-tree extends the versatility of Kd-tree by providing a competitive alternative, enabling the direct implementation of these algorithms in a compact form. As evidenced by our experiments, cKd-tree exhibits superior performance compared to  $k^2$ -tree, particularly when querying aggregate data. This suggests its potential for enabling faster execution on more intricate queries, including but not limited to  $K$  nearest neighbors and Pareto set calculations.

## References

- [1] J. L. Bentley, Multidimensional binary search trees used for associative searching, *Commun. ACM* 18 (1975) 509–517. URL: <https://doi.org/10.1145/361002.361007>. doi:10.1145/361002.361007.
- [2] H. Samet, The quadtree and related hierarchical data structures, *ACM Comput. Surv.* 16 (1984) 187–260. URL: <https://doi.org/10.1145/356924.356930>. doi:10.1145/356924.356930.
- [3] R. A. Brown, Building a balanced  $k$ -d tree in  $o(kn \log n)$  time, *Journal of Computer Graphics Techniques (JCGT)* 4 (2015) 50–68. URL: <http://jcgt.org/published/0004/01/03/>.
- [4] C. SanJuan-Contreras, G. Gutiérrez, M. A. Martínez-Prieto, D. Seco, cbik: A space-efficient data structure for spatial keyword queries, *IEEE Access PP* (2020) 1–1. doi:10.1109/ACCESS.2020.2997258.
- [5] V. Gaede, O. Günther, Multidimensional access methods, *ACM Computing Surveys* 30 (1998) 170–231. URL: <https://doi.org/10.1145/280277.280279>. doi:10.1145/280277.280279.
- [6] H. Samet, The quadtree and related hierarchical data structures, *ACM Computing Surveys (CSUR)* 16 (1984) 187–260.
- [7] N. Brisaboa, S. Ladra, G. Navarro, Compact representation of web graphs with extended functionality, *Information Systems* 39 (2014) 152–174.
- [8] S. Ladra, J. R. Paramá, F. Silva Coira, Scalable and queryable compressed storage structure for raster data, *Information Systems* (2017) 179–204.
- [9] J. Castro, M. Romero, G. Gutiérrez, M. Caniupán, C. Quijada-Fuentes, Efficient computation of the convex hull on sets of points stored in a  $k$ -tree compact data structure, *Knowledge and Information Systems* 62 (2020). doi:10.1007/s10115-020-01486-9.
- [10] N. Brisaboa, S. Ladra, G. Navarro, Dacs: Bringing direct access to variable-length codes, *Information Processing and Management* 49 (2013) 392–404.
- [11] G. Gutiérrez, R. Torres-Avilés, M. Caniupán, ckd-tree: A compact kd-tree, *IEEE Access* 12 (2024) 28666–28676. doi:10.1109/ACCESS.2024.3365054.
- [12] S. Gog, T. Beller, A. Moffat, M. Petri, From theory to practice: Plug and play with succinct data structures, in: *13th International Symposium on Experimental Algorithms, (SEA 2014)*, 2014, pp. 326–337.