

Faster Compressed Quadrees

Guillermo de Bernardo¹, Travis Gagie², Susana Ladra¹,
Gonzalo Navarro^{3,4} and Diego Seco^{3,5}

¹ Universidade da Coruña, CITIC, Database Lab, Spain

² Faculty of Computer Science, Dalhousie University, Canada

³ IMFD — Millennium Institute for Foundational Research on Data, Chile

⁴ Department of Computer Science, University of Chile, Chile

⁵ Department of Computer Science, University of Concepción, Chile

Abstract

Real-world point sets tend to be clustered, so using a machine word for each point is wasteful. In this paper we first show how a compact representation of quadtrees using $\mathcal{O}(1)$ bits per node can break this bound on clustered point sets, while offering efficient range searches. We then describe a new compact quadtree representation based on heavy-path decompositions, which supports queries faster than previous compact structures. We present experimental evidence showing that our structure is competitive in practice.

Key words: compact data structures, quadtrees, heavy-path decomposition, range queries, clustered points.

1. Introduction

Storing and querying two-dimensional points sets is fundamental in computational geometry, geographic information systems, graphics, and many other fields. Most researchers have aimed at designing data structures whose size, measured in machine words, is linear in the number of points. That is, data structures are considered small if they store a set of n points on a $u \times u$ grid in $\mathcal{O}(n)$ words of $\mathcal{O}(\log u)$ bits each. Using $\mathcal{O}(n \log u)$ bits is within a constant factor of optimality when the points are distributed uniformly at random over the grid, but we can often do better on real-world point sets because they tend to be clustered and, therefore, compressible.

Quadrees [1, 2] store the point's coordinates in implicit form, along a root-to-leaf path per point. Quadrees may have $o(n \log u)$ nodes when the points are clustered, because closer points tend to share a longer part of their path.

^{*}An early partial version of this paper appeared in *Proc. of the Data Compression Conference 2015*.

Still, classic quadtrees are implemented with pointers, which take $\Omega(\log u)$ bits per node, and since they use one node per point at the very least, they require $\Omega(n \log u)$ bits overall; the same happens if we store the explicit coordinates instead of the paths [3].

Recently, various authors [4, 5, 6] proposed quadtree representations based on succinct trees, which avoid pointers. These structures store the coordinates implicitly using the paths, and those paths use $\mathcal{O}(1)$ bits per quadtree node. Therefore, they are able to use $o(n \log u)$ bits of space, while offering the same asymptotic query times as traditional structures when supporting edge-by-edge navigation. Venkat and Mount [5] noted, however, that

“A method for compressing paths or moving over multiple edges at once using a succinct structure may speed up the many algorithms that rely on traversal of the quadtree.”

Some previous data structures, such as skip-quadtrees [7] and path-decomposed tries [8], are evidence that quadtree variants can indeed use $\mathcal{O}(1)$ bits per node while moving over multiple edges at once. The authors of skip-quadtrees only aimed at a space bound of $\mathcal{O}(n \log u)$ bits and did not give an implementation, while the authors of path-decomposed tries gave mainly experimental analyses.

This paper contains two main contributions:

1. We give a space analysis of quadtree data structures as a function of the amount of clustering of the point set, showing that compressed quadtrees can use $o(n \log u)$ bits of space on clustered points. We also show that quadtree queries speed up on clustered points.
2. We present the first compressed quadtree data structure that, within that space, uses heavy-path decomposition in order to provide one-step navigation over multiple edges, thereby speeding up queries.

After describing the compressed quadtree data structure in Section 2, contribution 1 is provided in Section 3, and contribution 2 in Sections 4 (which describes the new structure) and Section 5 (which gives the new query algorithms). In Section 6 we describe some practical improvements and in Section 7 we show experimentally that our structure is competitive, and in particular that it outperforms current alternatives when retrieving isolated points. We conclude in Section 8.

2. Basic Concepts

2.1. Model of computation

Like most of the work on compressed data structures, we assume the RAM model of computation, where the machine word holds $\Theta(\log u)$ bits and can perform all the usual arithmetic and bitwise operations on words in constant time. Note $\log n = \mathcal{O}(\log u)$ because $n \leq u^2$.

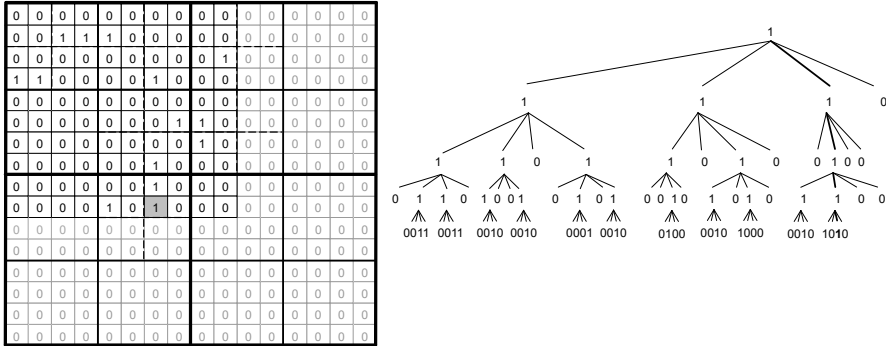


Figure 1: A set of points, indicated by 1s, on a 16×16 grid (left); the quadtree for those points (right). The heavy lines in the quadtree indicate the path to the leaf corresponding to the shaded point on the grid.

2.2. Bitvectors

A bitvector is an array $B[1, n]$ of bits. We are interested, apart from accessing any bit $B[i]$, in implementing two operations: $rank_b(B, i)$ counts the number of times bit b appears in $B[1, i]$, whereas $select_b(B, j)$ is the position of the j th occurrence of bit b in B . All these operations can be computed in constant time using only $o(n)$ extra bits on top of B [9, 10].

2.3. Quadtrees

There are many kinds of quadtrees. Our definition corresponds to the so-called MX-Quadtree [11, 2].

Definition 1. Let \mathcal{P} be a set of n points on a discrete grid $[1, u]^2$. If n is 0, then the quadtree for the grid is a leaf storing 0. If $u = 1$, then the quadtree is a leaf storing 1 if the cell contains a point and 0 if not. Otherwise, the quadtree is a tree whose root stores a 1 and has four children, which are the quadtrees of the grid's four quadrants. We say that a node covers the area of its subgrid and that it is an ancestor of the points in that subgrid.

Example. Figure 1 shows an example, taken from Brisaboa et al. [6]. Notice the order of the quadrants is top-left, top-right, bottom-left, bottom-right, instead of the counterclockwise order customary in mathematics. This is called the Morton or Z-ordering and it is useful because, assuming u is a power of 2 and the origin is at the top right — without loss of generality, since we can manipulate the coordinate system to make it so — the obvious binary encoding of a root-to-leaf path is the interleaving of the binary representations of the corresponding point's y - and x -coordinates.

For example, if we imagine the edges descending from each internal node in Figure 1 are labelled 0, 1, 2, 3 from left to right, then the thick edges are labelled 2, 1, 1, 2; the obvious binary encoding for this path is 10010110. The

coordinates for the shaded point, which corresponds to the leaf at the end of this path, are (6, 9), so interleaving the binary representations 1001 and 0110 of its y - and x -coordinates also gives 10010110. We can interleave a point's coordinates in $\mathcal{O}(1)$ time using, for example, pre-computed tables. \square

We now summarize a few simple facts on quadtrees.

Fact 1. *A quadtree for the set of points \mathcal{P} on a grid $[1, u]^2$ has height at most $\lg u$ and $\mathcal{O}(n \log u)$ nodes. A node at depth j covers a square area of size $2^{1-\lg(u)-j} \times 2^{1-\lg(u)-j}$. A leaf storing a 1 is at depth $\lg u$ and hence covers a single cell; there is exactly one such leaf per point in \mathcal{P} .*

A quadtree can efficiently find the points lying on a region of the grid.

Definition 2. *A query $R \subseteq [1, u]^2$ aims to retrieve the points of \mathcal{P} that lie within R . The result is denoted $\mathcal{P} \cap R$. When R is a rectangle, the query is called a range query, and the special case $R = [x, x + 1) \times [y, y + 1)$ is called a membership test for the point (x, y) .*

Given a query region R , the quadtree computes $\mathcal{P} \cap R$ by starting at the root and visiting all the nodes whose subgrids overlap R , reporting the coordinates of every leaf storing 1. In a range query we can determine in constant time whether a node's area overlaps R . This reduces the problem of computing the cost of solving a query to that of computing the size of a quadtree.

Fact 2. *Let $Q = \mathcal{P} \cap R$ be the output of a range query. Then the cost of enumerating Q by traversing all the nodes overlapping R in a quadtree for \mathcal{P} is proportional to the number of nodes in a quadtree for Q .*

The Quadtree Complexity Theorem [12, 13, 2] establishes that the number of maximal-area quadtree nodes inside a rectangle R of size $p \times q$, plus their ancestors, is $\mathcal{O}(p + q + \log u)$. We traverse all those nodes to solve the query $\mathcal{P} \cap R$, plus the paths towards every point inside R . If we pessimistically add $\lg u$ nodes to account for each such path, we obtain the following result.

Theorem 3. *The time complexity for solving the range query $Q = \mathcal{P} \cap R$, where R is a rectangle of size $p \times q$, on a quadtree for \mathcal{P} over a $[1, u]^2$ grid, is $\mathcal{O}(p + q + (|Q| + 1) \log u)$.*

If the points in \mathcal{P} are clustered, however, then intuitively the root-to-leaf paths in the quadtree will share many nodes and we will use less space and time. The query time can be refined to $\mathcal{O}(p + q + \log u + |Q|(1 + \log(pq/|Q|)))$ [14, p. 361], which shows that the time per reported point decreases on smaller or denser query ranges. We further exploit this idea to provide more refined space and time bounds on clustered points in the Section 3.

Quadtrees can be generalized to $d \geq 2$ dimensions, in which case each node has 2^d children. On a universe $[1, u]^d$, the quadtree still has height $\log_{2^d} u^d = \lg u$. The Quadtree Complexity Theorem formula generalizes to $\mathcal{O}(dq^{d-1} + \log u)$ for a hypercube R of side q , and consequently the search time complexity for the query $Q = \mathcal{P} \cap R$ becomes $\mathcal{O}(dq^{d-1} + (|Q| + 1) \log u)$.

2.4. Compressed quadtrees

Brisaboa, Ladra and Navarro [4] proposed a compressed quadtree representation called k^2 -tree (a quadtree corresponds to using $k = 2$). It represents a quadtree using exactly 1 bit per node, by collecting the 0s and 1s of the tree in levelwise order (omitting the root). They show that, by adding *rank* and *select* support to this concatenation of bits, the quadtree can be navigated towards children and parent in constant time: if we identify the node v with the position i so that the 4 bits describing its children are in $B[4i - 3..4i]$ (so the root is 1), then the identifier of the j th child of v is $\text{rank}_1(B, 4(i - 1) + j) + 1$, and that of the parent of v is $\lceil \text{select}_1(B, i - 1)/4 \rceil$.

Example. The quadtree on the right of Figure 1 is represented as a bitvector B concatenating the bits 1110 (the first level), followed by 110110100100 (the second level), and so on. To traverse the path in bold, we start at the root node, $i = 1$, and take the third child ($j = 3$) with $i' = \text{rank}_1(B, 4 \cdot 0 + 3) + 1 = 4$. Indeed, the child is the 4th node in a levelwise traversal of the quadtree. Its second child ($j = 2$) is $i'' = \text{rank}_1(B, 4 \cdot 3 + 2) + 1 = 10$. Again, the child is the 10th node in the levelwise traversal. The parent of node i'' is $\lceil \text{select}_1(B, 9)/4 \rceil = 4 = i'$. \square

There are several other variants of this representation [5, 6], as well as various techniques to further reduce space. A major improvement in compression [4] can be obtained in practice by exploiting small-scale regularities that arise in many real-world datasets. To do this, they consider small submatrices of a predefined size (for instance, 4×4 or 8×8), and only represent the tree up to those submatrices, effectively trimming the lower levels of the tree. The different submatrices that arise are then sorted by frequency and stored explicitly in a matrix vocabulary, and a sequence of matrix identifiers is used as a last level of the tree. Directly-Addressable Codes [15] (DACs) are used to store and access the sequence. The k^2 -tree representation can be naturally extended to d dimensions, hence becoming a k^d -tree.

3. Tighter Bounds on Quadtrees of Clustered Point Sets

We first bound the size of a quadtree when the points can be distributed in c clusters of the same level; then we generalize the result to hierarchical clustering.

Theorem 4. *Let \mathcal{P} be a set of points on the discrete grid $[1, u]^2$. Let $\mathcal{P} = \mathcal{P}_1 \uplus \dots \uplus \mathcal{P}_c$ be a partition of \mathcal{P} into c clusters, so that each \mathcal{P}_i contains $|\mathcal{P}_i| = n_i$ points lying on a square region of side ℓ_i (the square regions are not necessarily disjoint). Then the quadtree of \mathcal{P} has $\mathcal{O}(c \log u + \sum_i n_i \log \ell_i)$ nodes.*

Proof. Let S be any $\ell \times \ell$ square on the grid and $N = S \cap \mathcal{P}$ be the points of \mathcal{P} that lie within S . Let A be the set of ancestors of the points in N , and A' be the ancestors of the corners of S (those corners may or may not be in \mathcal{P}). Since the quadtree of \mathcal{P} has maximum height $\lg u$ and S has 4 corners, it holds

$$|A| \leq |A \cup A'| \leq |A \setminus A'| + |A'| < |A \setminus A'| + 4 \lg u.$$

By Fact 1, any ancestor v of a point in N that has depth at most $\lg(u/\ell)$ covers all the points in a square of size at least $2^\ell \times 2^\ell$. Therefore, the square must contain at least one corner of S , and thus $v \in A'$. It follows that

$$|A \setminus A'| \leq |N|(\lg u - \lg(u/\ell)) = |N| \lg \ell,$$

so $|A| < |N| \lg \ell + 4 \lg u$. Since each cluster \mathcal{P}_i has $|N| = n_i$ points that lie within a square of size $\ell_i \times \ell_i$, the result follows. \square

Theorem 5. *Let \mathcal{P} be a set of points on the discrete grid $[1, u]^2$, and \mathcal{T} be a tree with root r . Every node $t \in \mathcal{T}$ stores a set $\mathcal{P}_t \subseteq \mathcal{P}$ of n_t points, which is the union of the points stored at its children. The sets \mathcal{P}_t of all the nodes $t \in \mathcal{T}$ at the same depth form a partition of \mathcal{P} into clusters. The points \mathcal{P}_t of every node $t \in \mathcal{T}$ lie on a square region of side ℓ_t ; those regions need not be disjoint. Then the quadtree of \mathcal{P} has $\mathcal{O}\left(\sum_{t \in \mathcal{T} \setminus \{r\}} \log \ell_{p(t)} + \sum_{t \in L} n_t \log \ell_t\right)$ nodes, where $p(t)$ is the parent of t in \mathcal{T} and $L \subseteq \mathcal{T}$ is the set of leaf nodes in \mathcal{T} .*

Proof. Applying Theorem 4 on the non-hierarchical clustering induced by the $c = |L|$ leaves of \mathcal{T} , we obtain the upper bound $\mathcal{O}(|L| \log \ell_r + \sum_{t \in L} n_t \log \ell_t)$. We now refine the first term of the bound, which comes from adding up the ancestors of the 4 corners of each of the regions in L . Instead of adding up their ancestors up to the root, let us count those ancestors in a finer-grained mode. Consider the 4 corners of a square of size $\ell_t \times \ell_t$ containing the points in \mathcal{P}_t . Their $\mathcal{O}(\log \ell_{p(t)})$ ancestors of depth over $\lg u - \lg(u/\ell_{p(t)})$ are charged to the node t . The higher ancestors, however, cover a square of size at least $2^{\ell_{p(t)}} \times 2^{\ell_{p(t)}}$ by Fact 1, and therefore are also ancestors of some of the 4 corners of the area of the parent of t , $p(t)$. We then do not need to account for those higher ancestors of the corners of the area of t . The result follows. \square

For example, consider a hierarchical clustering where each cluster lying on a square region of side ℓ distributes its points evenly into c sub-clusters lying on squares of side ℓ/s , for $\log_s u$ levels of clustering. By Theorem 5, the quadtree has $\mathcal{O}(n \log s)$ nodes, and its compressed representation uses $\mathcal{O}(n \log s)$ bits, which can be $o(n \log u)$.

Due to Fact 2, these result also bound the cost of a query on the quadtree of a set of clustered points, because we traverse precisely the quadtree nodes that lead to the output points.

All those results easily generalize to d dimensions, by enclosing each cluster in a hypercube with 2^d corners.

Corollary 6. *Let \mathcal{P} be a set of points on the discrete grid $[1, u]^d$ and \mathcal{T} be a tree defined as in Theorem 5, except that now the points \mathcal{P}_t of every node $t \in \mathcal{T}$ lie on a hypercube of side ℓ_t ; those hypercubes need not be disjoint. Then the quadtree of \mathcal{P} has $\mathcal{O}\left(\sum_{t \in \mathcal{T} \setminus \{r\}} 2^d \log \ell_{p(t)} + \sum_{t \in L} n_t \log \ell_t\right)$ nodes.*

We can combine these results with the improvements that favor dense clusters [14], though the formulas are messier: $n_t \log \ell_t$ becomes $n_t \log \min(\ell_t, u/n_i^{1/d})$.

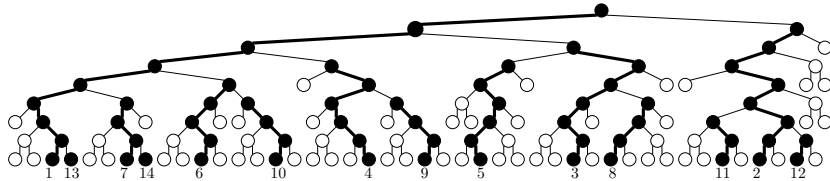


Figure 2: The heavy-path decomposition of the binary tree for the example from Figure 1. Nodes storing 1s are black; nodes storing 0s are shown hollow, and discarded; thick edges belong to heavy paths. The numbers below the black leaves indicate our path ordering.

4. A Compressed Quadtree Representation based on Heavy Paths

We describe a new compressed quadtree representation for two-dimensional points which, like the k^2 -tree, uses $\mathcal{O}(1)$ bits per node and supports the basic navigation towards parent and children in $\mathcal{O}(1)$ time. In the next section we show that this representation can support queries faster than the k^2 -tree and, in general, than the standard quadtree representations. We will also generalize our representation to higher dimensions.

4.1. Data structure

To store a quadtree, we first replace each internal node by a binary tree of height 2 and remove any node that has no descendant storing a 1. Let T be the resulting binary tree. The number of nodes in T is $1/2$ (when every quadtree node has only one child with a 1) to $3/2$ (when all quadtree nodes children have 1s) of those in the quadtree. In addition to simplifying our construction, this modification makes quadtrees more practical in higher dimensions [16], which we will also consider at the end of this section.

We then perform a heavy-path decomposition [17] of T , as follows.

Definition 3. *A heavy-path decomposition of T is a recursive decomposition of T into paths called heavy paths. The first heavy path goes from the root to a leaf, so that if the path contains a node v then it also contains the child of v with the most leaf descendants (breaking ties arbitrarily). Once the first heavy path is defined, its nodes are cut off the tree T , leaving a forest of former subtrees of T . We then recursively decompose every remaining subtree into heavy paths.*

A well-known property of this decomposition is that every root-to-leaf path in T consists of $\mathcal{O}(\log n)$ initial segments of heavy paths. In the sequel we call heavy paths simply paths.

Example. Figure 2 shows the heavy-path decomposition of the binary tree for our example of Figure 1. \square

We encode each path h as a binary string whose 0s and 1s indicate which of h 's nodes are left children and which are right children, respectively (considering the root as a left child), in increasing order of their depths. Note that all the

```

H    000000110 10010100 1100010 110111 001001 10010 1010 1000 1110 1110 010 10 1 1 ,
L0  1-----,
L1  -1-----0-----,
L2  --1-----0-----1-----,
L3  ---1-----0-----0-----0-----0-----,
L4  ----1-----0-----1-----1-----0-----1-----,
L5  -----0-----1-----0-----0-----0-----0-----0-----0-----0-----,
L6  -----0-----1-----0-----0-----0-----0-----0-----0-----0-----0-----,
L7  -----1-----0-----0-----0-----0-----0-----0-----1-----0-----0-----0-----0-----;

```

```

P[1..9] = (63, 61, 58, 42, 37, 25, 18, 10, 1)
N[1..9] = (12, 11, 10, 6, 5, 3, 2, 1, 0)

```

Figure 3: The bitvectors H and L_d and the arrays P and N for the tree of Figure 2. Dashes and spaces are shown only to indicate how the bits in L_d and H correspond.

paths end at the same depth, and thus their length plus the depth of their topmost node is the same for all.

We then sort the set of all those path encodings in decreasing order of their length. Ties between two paths h and h' of the same length are broken as follows: if the topmost nodes of h and h' are v and v' , respectively, the paths are ordered in the same way of the paths containing the parents of v and v' . Notice that v and v' cannot have the same parent, since they have the same height and the tree is binary. The numbers below the leaves in Figure 2 indicate how we order the paths in our example.

Our first structure is a bitvector H that concatenates the encodings of all the paths, once sorted as described. This bitvector has exactly $|T|$ bits, one representing each node of T . We say that the bit $H[i]$ corresponds to the node v if $H[i]$ indicates whether v is a left child or a right child.

For each depth $d < 2 \lg u$ (considering the root to have depth 0 and leaves to have depth $2 \lg u$), we store a bitvector L_d with 1s indicating which nodes at that depth in T have two children. These bitvectors have as many bits as there are internal nodes in T . Figure 3 shows them for our running example.

Our final structures are much smaller: an array P of $2 \lg u + 1$ entries stores in $P[\ell]$ the position in H where the first path of length ℓ (measured in number of nodes) is encoded (or null if there are no paths of that length). Similarly, an array N stores in $N[\ell]$ the number of paths longer than ℓ . We give support to perform predecessor queries on P and N : $\text{pred}(P, i)$ gives the minimum length ℓ for which $P[\ell] \leq i$. Because we sorted the paths by decreasing length in H , ℓ is the length of the path $H[i]$ belongs to. Similarly, $\text{pred}(N, k)$ tells the length ℓ of the k th path in H .

Definition 4. *Our compressed quadtree representation for n points on a $u \times u$*

grid has the following components, whose precise contents are defined above:

- A bitvector H of $|T|$ bits concatenating all heavy paths.
- Bitvectors L_d , for $0 \leq d < 2 \lg u$, with less than $|T|$ bits in total.
- Arrays P and N , of $\mathcal{O}(\log u)$ integers overall.

Theorem 7. *Our compressed quadtree representation on a set of points in $[1, u]^2$ uses $\mathcal{O}(1)$ bits per quadtree node, plus $\mathcal{O}(\log^2 u)$ bits.*

Proof. Bitvector H and all the L_d s take $\mathcal{O}(1)$ bits per node in T and, therefore, $\mathcal{O}(1)$ bits per node of the original quadtree. The arrays P and N , with predecessor data structures, require just $\mathcal{O}(\log^2 u)$ further bits. \square

4.2. Navigation

In this section we show how the basic parent/child navigation can be supported on our data structures.

4.2.1. Moving to the parent

Suppose $H[i]$ corresponds to node v in T . As explained, we obtain the length ℓ of the path containing v with $\ell = \text{pred}(P, i)$. Further, the path of v is the k th (in our ordering) of length ℓ and v is the j th top-down node in its path, where $k = \lceil (i - P[\ell] + 1) / \ell \rceil$ and $j = (i - P[\ell] + 1) - (k - 1)\ell$. Because the top node in the path of v is at depth $2 \lg u - \ell$, the depth of v is $d = 2 \lg u - \ell + j$.

If $j > 1$, v is not the topmost node in its path, and then its parent u corresponds to $H[i - 1]$. If $j = 1$, instead, u belongs to another path. Since u is at depth $d - 1$, it is mentioned in L_{d-1} . Further, the nodes at depth $d - 1$ with a child starting a path are exactly those that have two children (the other child continues the path of its parent). Finally, because we order the paths according to the order of their parent node, it turns out that, since v starts the k th path at depth d , its parent u is the k th node at depth $d - 1$ having two children. The position of u in L_{d-1} is then found with $p = \text{select}_1(L_{d-1}, k)$.

Because the paths are deployed on H by increasing starting depth (or decreasing length), all the paths having a node at depth $d - 1$ precede those that do not. Further, since the nodes in L_{d-1} appear in the same order of H , we have that the node at $L_{d-1}[p]$ is the node at depth $d - 1$ in the p th path of H . The length of that path is found with $\ell' = \text{pred}(N, p)$, and it is the k' path of length ℓ' , for $k' = p - N[\ell']$. The position i' of u in H is then computed as $i' = P[\ell'] + (k' - 1)\ell' + (d - 1) - (2 \lg u - \ell') - 1$: the paths of length ℓ' start at $P[\ell']$, then we have the preceding $k' - 1$ paths of length ℓ' , and in our path we want the node with absolute depth $d' - 1$, which we convert to an offset (i.e., relative depth) by subtracting the depth of the first node in the path, $2 \lg u - \ell'$.

Example. Let v be the top node in the ninth path in our ordering (see the first node in the path labeled 9 in Figure 2). It corresponds to the underlined position $H[i = 50]$. Its parent u is the second node in the fourth path, at $H[i' = 26]$. To find u from $i = 50$, we first compute $\ell = \text{pred}(P, 50) = 4$, the length of the path v belongs to. We also compute $k = \lceil (50 - 42 + 1)/4 \rceil = 3$ and $j = (50 - 42 + 1) - 2 \cdot 4 = 1$, so v is the first node in its path, which is the 3rd of length 4. The depth of v is $d = 8 - 4 + 1 = 5$. Since $j = 1$, u lies in another path. It is of depth $d - 1 = 4$, and thus mentioned at position $p = \text{select}_1(L_4, 3) = 4$. To find its position in H , we compute $\ell' = \text{pred}(N, 4) = 6$, the length of its path, as well as $k' = 4 - N[6] = 1$, so that the path of u is the first of length 6. We then compute its position $i' = P[6] + 0 + 4 - (8 - 6) - 1 = 26$. \square

4.2.2. Moving to a child

We compute ℓ , k , j , and d for v as before. If the depth d of v is $2 \log u$, then v is a leaf; otherwise it has left and/or right children. Further, one of the children of v is at $H[i + 1]$: the left child if $H[i + 1] = 0$ and the right if $H[i + 1] = 1$.

To determine if v has another child, and where, we must locate v in L_d . We compute $r = N[\ell] + k$, the rank of v 's path in H , thus v is the r th node of depth d . Therefore, v has another child iff $L_d[r] = 1$.

This child is the top node of another path, which starts at depth $d + 1$ and thus is of length $\ell'' = 2 \log u - d + 1$. Since there are $s = \text{rank}_1(L_d, r)$ nodes of depth d from where new paths start, the child of v is at $H[i'']$, where $i'' = P[\ell''] + (s - 1)\ell''$.

Example. Reversing our previous example, we have for u the values $i = 26$, $\ell = 6$, $k = 1$, $j = 2$, and $d = 4$. Since u 's depth is $d = 4 < 8$, u is not a leaf. One of its children is at $H[26 + 1]$; since $H[27] = 0$, this is the left child of u . To find if there is a right one, we compute $r = N[6] + 1 = 4$, so the path of u is the 4th in H and u is the 4th node of depth 4. Since $L_4[4] = 1$, u has a right child. This child starts a path at depth $d + 1 = 5$, of length $\ell'' = 8 - 5 + 1 = 4$, and it is the 3rd of those because $s = \text{rank}_1(L_4, 4) = 3$. We then find the right child at $H[i'']$, where $i'' = P[4] + 2 \cdot 4 = 50$, where indeed we find v . \square

Theorem 8. *Our compressed quadtree representation on a set of 2-dimensional points can move to the parent of a node or to any desired child in $\mathcal{O}(1)$ time.*

Proof. Moving to the parent or to a child in the quadtree requires a constant number of steps on T , each of which is dominated by the time of the predecessor operations on P and N . Since these arrays have a logarithmic number of elements, predecessors can be computed in constant time [18]. \square

More practically, we note that, on a downward traversal from the root, we always know the length ℓ of the current path, and therefore we can perform the operations without the need of predecessor queries. In Section 5 we show how heavy paths speed up the specific operations to query quadtrees.

4.3. Higher dimensions

In dimension d , each quadtree node has 2^d children, and the quadtree is still of height $\lg u$. Therefore our binary tree T introduces $d - 1$ levels per quadtree edge, reaching height $d \lg u$. Simulating a quadtree move towards the parent or children takes $\mathcal{O}(d)$ steps in T . Each such step may require predecessor queries on the arrays P and N , which now contain $\mathcal{O}(d \log u)$ elements. Those predecessor queries can be carried out in time $\mathcal{O}\left(\log \frac{\log d}{\log \log u}\right)$ [19].

The addition of new nodes in T may increase their number by a factor of $\frac{2^{d+1}-2}{2^d} < 2$ with respect to the number of nodes in the original quadtree (if a quadtree node has 2^d children, T adds $2^d - 2$ new nodes between the parent and the children). If we consider the number of 1s in the original quadtree, then T has at most d nodes per 1 (i.e., a path of length d towards the 1-child of every quadtree node), and thus $\mathcal{O}(d \log u)$ nodes per represented point.

Corollary 9. *Our compressed quadtree representation on a set of points in $[1, u]^d$ uses $\mathcal{O}(1)$ bits per quadtree node (or, alternatively, $\mathcal{O}(d \log u)$ bits per point), plus $\mathcal{O}(d \log^2 u)$ bits. It can move to the parent of a node or to any desired child in time $\mathcal{O}\left(d \log \frac{\log d}{\log \log u}\right)$.*

These space and time factors are similar to what can be obtained on previous compressed quadtree representations [4, 14] on high dimensions. Although they can move to parents and children in constant time, their space may grow up to 2^d bits per node, that is, exponentially with the dimension, because each node has 2^d children and most of them are 0s. This can be alleviated with a bitvector representation for B that exploits sparsity [20], which recovers the $\mathcal{O}(1)$ bits per 1 in the quadtree. In exchange, operation *rank* takes time $\mathcal{O}(d)$, so moving to a child takes time $\mathcal{O}(d)$, while moving to the parent still takes $\mathcal{O}(1)$ time.

Although both solutions seem then comparable in terms of space and basic operations, we show next how to leverage the heavy-path representation to support root-to-leaf traversals in time $\mathcal{O}(d \log d + \log n)$ instead of $\mathcal{O}(d \log u)$. This is particularly relevant for membership queries.

5. Membership and Range Queries

Suppose we want to determine whether the point (x, y) is in the set. The first step is to obtain the Morton code M of the point, by interlacing the bits that describe the integers x and y . We can do this in time $\mathcal{O}(1/\epsilon)$, for any constant ϵ , with a table using $\mathcal{O}(u^\epsilon)$ space. This table does not depend on the data points: for any two chunks of $(\epsilon/2) \lg u$ bits, the table returns their interlacing. We can then find the Morton code of (x, y) by pieces of $\epsilon \lg n$ bits, via $2/\epsilon$ accesses to the table with the consecutive pieces of $(\epsilon/2) \lg u$ bits of x and y .

We now enter the binary tree T from the root, using the successive bits of M to decide whether to go left or right. That is, each bit of M corresponds to an edge to follow. Instead of processing M bit by bit and descending in T edge by edge, however, we descend path by path.

Algorithm 1: Membership (x, y)

```
1  $M[1..2 \lg u] \leftarrow$  Morton code of  $(x, y)$ ;  
2  $s \leftarrow 1$ ;  
3  $\ell \leftarrow 2 \lg u + 1$ ;  
4  $p \leftarrow 1$ ;  
5  $d \leftarrow 0$ ;  
6 while true do  
7    $m \leftarrow \text{LCP}(M[d+1..], H[p+1..])$ ;  
8   if  $m = \ell - 1$  then return yes;  
9    $r \leftarrow N[\ell] + s$ ;  
10  if  $L_{d+m}[r] = 0$  then return no;  
11   $s \leftarrow \text{rank}_1(L_{d+m}, r)$ ;  
12   $\ell \leftarrow \ell - m - 1$ ;  
13   $p \leftarrow P[\ell] + \ell \cdot (s - 1)$ ;  
14   $d \leftarrow d + m + 1$ ;
```

We first determine the prefix of the first path (the one starting at the root) that we must follow. For this sake, we compute $m = \text{LCP}(M[d+1..], H[p+1..])$, where $d = 0$, $p = 1$, and $\text{LCP}(X, Y)$ is the length of the longest common prefix between bitstrings X and Y (we start from $p + 1$ because the first bit of the first path, $H[1]$, is spurious, whereas $H[2]$ refers to the first edge). Note that the length of the first path, $H[p..]$, is $\ell = 2 \lg u + 1$, with $\ell - 1$ edges.

If $m = \ell - 1$, then $M[d + 1..]$ matches the whole path starting at $H[p]$, and then we know that the point is stored in the quadtree. If not, then $M[d + 1..]$ shares its first m edges with the path starting at $H[p]$, matching up to node $H[p + m]$, but not $H[p + m + 1]$ (e.g., if $m = 0$ and $p = 1$ then M matches only the root node). We must then determine if $H[p + m]$ has two children and, if so, move to the other child. This is done as described in Section 4.2. If $H[p + m]$ has only one child, then (x, y) is not in the set. Otherwise, letting $H[p']$ be the other child of $H[p + m]$, we know that $H[p']$ starts a path of length $\ell' = \ell - m - 1$, with $\ell' - 1$ edges. We then update $p \leftarrow p'$, $d \leftarrow d + m + 1$, $\ell \leftarrow \ell - m - 1$.

This process is repeated until we find (x, y) or determine it is not in the set of points. Since we switch to another descendant heavy path at each step in our process, we perform $\mathcal{O}(\log n)$ steps [17]. Algorithm 1 gives the pseudocode.

Note that we always know the length ℓ of the path we are navigating, and therefore moving to a child requires only $\mathcal{O}(1)$ time. Just the *rank* functionality on the bitvectors, without using *select* nor predecessor queries, is needed. We can also compute $m = \text{LCP}(X, Y)$ in $\mathcal{O}(1)$ time if $|X|$ and $|Y|$ are $\mathcal{O}(\log u)$ (we apply LCP on $H[p + 1..]$, but it suffices to consider only the first $2 \lg u$ bits of that suffix). With $Z = X \text{ xor } Y$, the m highest bits become 0 and the $(m + 1)$ th becomes 1. We then use a constant-time technique to find the highest 1 in Z [21]. We can also compute LCP using tables of size u^ϵ , as before.

Example. To perform a membership query for $(6, 9) = (0110_2, 1001_2)$ in our

quadtrees of Figure 1 (the shaded cell), we first interlace the bitstrings to obtain the path label, $M = 10010110$, and then use it to traverse the path-decomposed tree T of Figure 2. We first try to match $M[1..]$ with the longest path, of length 9, from $H[2..] = 00000110\dots$ (see Figure 3). The common prefix is of length $m = 0$, so we cannot go past the root by that path. We then see that the root has another child, which is $H[10]$, starting a path of length $9 - 0 - 1 = 8$ (our algorithm computes $r = 1$, $s = 1$, $\ell = 8$, $p = 10$, and $d = 1$, continuing because $L_0[1] = 1$). Since $m = \text{LCP}(M[2..], H[11..]) = 5$, we can advance up to $H[15]$, where M wants to go right ($M[7] = 1$) but the path goes left ($H[16] = 0$). We then find that $H[15]$ has another child, $H[61]$, which starts a path of length $8 - 5 - 1 = 2$ (our algorithm computes $r = 2$, $s = 1$, $\ell = 2$, $p = 61$, and $d = 7$, continuing because $L_6[2] = 1$). We finally compute $m = \text{LCP}(M[8..], H[62]) = 1$, so we have arrived at a leaf ($m = \ell - 1$) and report that the point exists. \square

In the worst case, we must traverse $\mathcal{O}(\log n)$ paths along this process. This contrasts with the $\mathcal{O}(\log u)$ time needed with the classical representation, showing that our structure should be faster on sparse points sets. Further, we need fewer path switches in the way to isolated points. The next theorem shows that the membership time indeed improves on those points.

Theorem 10. *With the help of a constant table using u^ϵ space (for any constant $\epsilon > 0$), our compressed quadtree representation supports a membership query for (x, y) in $\mathcal{O}(\log n)$ time. Further, the time is $\mathcal{O}(\min_g \{\log(u/g) + \log k_g\})$, where k_g is the number of points in \mathcal{P} within distance g of (x, y) .*

Proof. The $\mathcal{O}(\log n)$ bound follows from the heavy-path decomposition. Further, any ancestor v of (x, y) of depth at least $2\lg(u/g) + 2$ in T covers a subgrid of size at most $g/2 \times g/2$, whose points that are then at distance at most $(g/2)\sqrt{2} < g$ from (x, y) . Thus, v covers at most k_g points of \mathcal{P} . It follows that the path from v to the deepest ancestor $w \in T$ of (x, y) consists of $\mathcal{O}(\log k_g)$ initial segments of heavy paths. To see why, consider that if we ascend from w to v , every time we move from the topmost node in one heavy path to its parent in another heavy path, the number of leaf descendants in the subtree below us at least doubles. Since the path from the root to v has length $\mathcal{O}(\log(u/g))$, the path from the root to w consists of $\mathcal{O}(\log(u/g) + \log k_g)$ initial segments of heavy paths. \square

The following corollary, which combines Theorems 5 and 10, suggests that our structure should be particularly suited to applications in which points are highly clustered (e.g., towns) but queries are chosen uniformly or according to a different distribution (e.g., seismic activity).

Corollary 11. *Let \mathcal{P} be a set of points on the discrete grid $[1, u]^2$ and \mathcal{T} be a tree defined as in Theorem 5. Let \mathcal{T}_ℓ be the set of nodes at level ℓ in \mathcal{T} . Then a membership query for (x, y) takes $\mathcal{O}(\min_\ell \max_{t \in \mathcal{T}_\ell} \log(u/d(\mathcal{P}_t, (x, y))))$ time, where $d(\mathcal{P}_t, (x, y))$ is the minimum distance between a point in \mathcal{P}_t and (x, y) .*

Proof. Let $m_\ell = \max_{t \in \mathcal{T}_\ell} \log(u/d(\mathcal{P}_t, (x, y))) = \log(u/\min_{t \in \mathcal{T}_\ell} d(\mathcal{P}_t, (x, y)))$. Let us define $g = (1/2) \min_{t \in \mathcal{T}_\ell} d(\mathcal{P}_t, (x, y))$, so $k_g = 0$ and $m_\ell = \mathcal{O}(\log(u/g))$.

By Theorem 10, the cost of the membership query is then $\mathcal{O}(\log(u/g) + \log k_g) = \mathcal{O}(m_\ell)$. This bound holds for every level ℓ , so the time is $\mathcal{O}(\min_\ell m_\ell)$. \square

5.1. Range queries

In order to output all the points in $\mathcal{P} \cap R$ given a query region $R = [x_1, x_2] \times [y_1, y_2]$, we first traverse via heavy paths towards the lowest ancestor of R in the quadtree. For this sake, we compute $m = \min(\text{LCP}(x_1, x_2), \text{LCP}(y_1, y_2))$ and define x and y as the first m bits of x_1 and y_1 , respectively. We then interlace (x, y) into a bitstring M of length $2m$, and traverse towards M in the quadtree as described in the main part of this section. The node v we arrive at is the ancestor of all the points in R . We now traverse edge by edge towards all the descendants of v using the method described in Section 4.2, in constant time per edge traversed, avoiding to enter into nodes whose area does not intersect R , and reporting all the leaves found.

The node v can be very high in the tree, even for small regions, and thus this method is no faster in the worst case than the classical one. We expect, however, it to be faster in many cases when the query region is small. We inherit the refined bounds for classic quadtrees [14, p. 361].

Corollary 12. *Our compressed quadtree representation outputs $Q = \mathcal{P} \cap R$, for a rectangle R of size $p \times q$, in time $\mathcal{O}(p + q + \log u + |Q|(1 + \log(pq/|Q|)))$.*

5.2. Higher dimensions

In dimension d , the description of the point sought, (x_1, \dots, x_d) , has $d \lg u$ bits, and it might not fit in a computer word of size $\mathcal{O}(\log u)$. We can still use the precomputed table of size u^ϵ described at the beginning of this section to obtain the bitstring M (of length $d \lg u$) in time $\mathcal{O}((1/\epsilon)d \log d)$, as follows. Build a binary tree where the root represents all the d coordinates, $1, \dots, d$. Its left child represents the odd positions of the parent, $1, 3, 5, \dots$ and the right child the even positions, $2, 4, 6, \dots$. This division, taking odd and even positions, continues until the leaves represent only one dimension $1 \leq i \leq d$ and store the bitstrings x_i . Now, bottom up, every internal node merges the bitstrings of its two descendants by chunks of $(\epsilon/2) \lg u$ bits, until the root obtains M . It is easy to see that the tree has $\lg d$ levels and that the total work per level is $\mathcal{O}(d/\epsilon)$.

Once we obtain M , the membership query proceeds as for two dimensions. The only difference is that a single LCP query may take time $\mathcal{O}(d)$, because the prefix may coincide in $m = \mathcal{O}(d \log u)$ bits. However, the sum of all those m values along the search is also $\mathcal{O}(d \log u)$, because we advance in M by $m + 1$ positions each time. Therefore, the time to descend to the leaf (x_1, \dots, x_d) or to determine it does not exist is $\mathcal{O}(d + \log n)$. Note that we do not require predecessor queries to determine membership.

Corollary 13. *With the help of a constant table using u^ϵ space (for any constant $\epsilon > 0$), our compressed quadtree representation supports a membership query for (x_1, \dots, x_d) in $\mathcal{O}(d \log d + x_1 n)$ time.*

Our finer results can be similarly extended to d dimensions; we leave them as exercises to the reader.

6. Practical Optimizations and Implementation Variants

For the creation of the path labels, we use a precomputed table of 256 entries to compute the interleaving byte-wise, together with some arithmetics to build the final path. For the 3-dimensional case we have also used an implementation based on magic numbers. In practice, the computation of the initial path has negligible effect on the total query times.

The query algorithms described in previous sections always use a top-to-bottom traversal of the conceptual tree T , which leads to a number of practical optimizations. A first practical choice, already mentioned, is the adjustment of the traversal algorithms to keep track of the current depth, thereby avoiding the need for predecessor structures in P and N . In this section we describe other practical variants that can reduce the space usage of our structure.

A first significant space improvement can be obtained by removing information in H that can be deduced during top-down traversals. Bitvector H stores, for each path, a bitstring representing its nodes, marking whether each is a left or a right child. During top-down traversal, we always start at the beginning of the first path, and whenever we switch to a new path we always start at its beginning. Note that when switching paths, the first bit of the new path can be inferred, since it is the opposite of the next bit in the current path. Indeed, in the membership query of Section 5 we always skip that first bit, $H[p]$, and compare $M[d + 1..]$ with $H[p + 1..]$. Therefore, we can remove the first bit of each path in H and still perform top-down traversals on the tree. Since all the paths are shortened in the same way, the navigational properties remain the same and only minor changes are required. Overall, we save one bit per path, or which is the same, per point in \mathcal{P} .

Another improvement in space can be obtained by noting that there are only n 1s across all the bitvectors L_d , at the starting node of each path. In contrast, their total length can be up to $2n \lg u$ (i.e., one path in T per point), getting closer to that maximum on sparse datasets. We can then use for the bitvectors L_d a compressed bitvector representation [22] that supports constant-time access and *rank* queries but uses less space when the bitvector has many more 0s than 1s. With that representation, the whole set of bitvectors L_d fits within $\mathcal{O}(n \log \log u)$ bits. In practice this representation is slower, though.

Finally, we can also apply to our structure the matrix-vocabulary compression applied on the last levels of the quadtree [4], described at the end of Section 2.4. We can trim T a few levels above the last one, and use DACs to replace the removed levels by a matrix vocabulary and a DAC-encoded sequence of matrix identifiers. All the query algorithms remain the same, though they stop at a smaller depth d' . Once this depth is reached, we find the corresponding submatrix, and perform single-cell access or range queries over the submatrix in the same ways as the classical compressed quadtree [4].

7. Experimental Evaluation

7.1. Experimental framework

We tested the performance of our solution on real datasets from different domains. We consider grids extracted from geographic information systems (GIS), social networks (SN), Web graphs (WEB) and RDF datasets (RDF).

- The datasets *dblp* and *enwiki* are network data corresponding to the social network datasets *dblp-2011* and *enwiki-2013*, provided by the Laboratory for Web Algorithmics¹ [23, 24].
- Collections *indochina* and *uk* are obtained from Web graph crawls, from the *indochina-2004* and *uk-2002* datasets provided by the Laboratory for Web Algorithmics.
- We build three datasets storing geographic information by processing the Geonames dataset, which stores over 9 million locations, discretizing them on a grid. We build three different datasets (*GIS-sparse*, *GIS-med*, *GIS-dense*) by varying the resolution of the grid.
- We build three datasets storing RDF-based grids, by parsing the DBpedia dataset². RDF stores triples (S,P,O) that represent labeled edges in a graph, where the predicate P represents the label. We partition the dataset by predicate, so each individual element can be regarded as a binary grid, and select three different datasets, *RDF-sparse*, *RDF-med*, and *RDF-dense*, with significantly different number of points in the grid.

These datasets aim at testing the performance of our technique on a wide variety of real-world applications. The selection of GIS-based and RDF-based datasets also aims at providing insights on its relative performance depending on the sparsity of the data, which is a key element for our structure. Table 1 describes the main characteristics of the studied datasets, including the grid size (all grids are square, of size $u \times u$) and the number n of points in the grid.

We compare our representation with the k^2 -tree [4], the best known compressed quadtree representation, which has been shown to achieve very good compression on most of those domains, especially on Web graphs and RDF data. We use two different implementations of the k^2 -tree: k^2 -tree^p is a direct implementation of the structure, with all the bitmaps stored in plain form, and using $k = 2$ in all levels of the tree; k^2 -tree^{DAC} is an enhanced version that applies a number of improvements over the basic approach: it uses $k = 4$ in the first 6 levels of decomposition and $k = 2$ in the remaining levels; also, DACs are used to replace the last 3 levels of the tree by submatrices of size 8×8 .

¹<http://law.di.unimi.it>

²<http://wiki.dbpedia.org/Downloads351>

File	Type	Grid size (u)	Points (n)
dblp	SN	986,324	6,707,236
enwiki	SN	4,206,785	101,355,853
indochina	WEB	7,414,866	194,109,311
uk	WEB	18,520,486	298,113,762
GIS-sparse	GIS	67,108,864	9,335,371
GIS-med	GIS	4,194,304	9,328,003
GIS-dense	GIS	524,288	9,188,290
RDF-sparse	RDF	66,973,084	138,303
RDF-med	RDF	66,973,084	7,936,138
RDF-dense	RDF	66,973,084	98,714,022

Table 1: Description of the datasets used in our experiments

We also compare our representation with path-decomposed tries (PDT) [8]. We use two of the configurations proposed by the authors that provide a reasonable space-time tradeoff: the centroid hollow monotone-hash technique (PDT-hollow) and the centroid compressed trie, where labels are compressed using RePair (PDT-RP). Note that PDT is designed to represent string dictionaries, and only supports membership queries. In order to transform the point grids into collections of strings suitable for PDT, we use the Morton code for each individual point in the collection and build the PDT representation of the collection of Morton codes. Membership queries are directly translated into PDT operations, but queries involving rows/columns or ranges are not specifically supported and are therefore transformed into a number of membership queries.

We test four different implementations of our proposal, `hpqt`, considering two main variables. First, bitvectors L_d can be stored in plain form (`hpqtp`) or compressed with the so-called RRR technique [22] (`hpqtc`). Second, we may use our basic implementation, with all paths stored completely, or use the DAC-based compression of the submatrices in the lower levels. This compression leads to two further variants, `hpqtp+DAC` and `hpqtc+DAC`.

We implemented the `hpqt` variants in C++, using LibCDS 2³ to provide the bitvector implementations used. Both k^2 -tree implementations are provided by the authors and implemented in C. PDT is implemented in C++, and obtained from the original author’s repository⁴. All implementations were compiled using GCC with full optimization enabled. Experiments were executed on a machine with Intel Xeon E5-2470@2.3GHz (8 cores) CPU, and 64GB of RAM. The operating system was Debian 9.8 (kernel 4.9.0-8-amd64).

7.2. Space usage

In this section we compare the compression performance of `hpqt` with k^2 -tree variants. Table 2 displays the compression obtained, in bits per point,

³<https://github.com/fclaude/libcds2>

⁴https://github.com/ot/path_decomposed_tries

Dataset				DAC			PDT	
	hpqt ^p	hpqt ^c	k ² -tree ^p	hpqt ^p	hpqt ^c	k ² -tree	hollow	RP
dblp	11.62	9.23	10.76	10.08	8.88	9.84	<u>9.00</u>	25.43
enwiki	17.56	13.49	16.96	15.01	<u>13.37</u>	14.66	9.11	31.07
indochina	3.29	2.92	2.57	1.28	<u>1.27</u>	1.22	7.51	16.10
uk	4.04	3.73	3.30	2.08	2.02	<u>2.04</u>	7.99	17.04
GIS-sparse	44.19	29.66	44.01	38.32	<u>28.48</u>	38.02	8.23	51.15
GIS-med	30.61	21.28	30.10	25.42	<u>20.72</u>	24.83	8.22	40.64
GIS-dense	17.37	<u>13.05</u>	16.55	13.85	13.21	13.17	8.14	29.81
RDF-sparse	45.01	30.35	45.69	39.81	<u>29.63</u>	46.98	11.06	57.36
RDF-med	11.19	9.02	9.80	7.36	<u>7.09</u>	6.93	9.28	24.30
RDF-dense	31.94	22.22	31.61	27.28	<u>21.54</u>	26.93	9.31	43.42

Table 2: Space required by all implementations (in bits per point). We put in bold the best and underline the second best space for each dataset.

in all the test datasets. Let us first focus on the comparison between hpqt variants and the k^2 -tree. The results show that hpqt^c variants achieve better compression than the k^2 -tree in almost all the datasets. The k^2 -tree only obtains the best compression results in indochina and RDF-med, two datasets where the differences between representations are not very high in general. Plain versions, hpqt^p, are still slightly larger than the equivalent k^2 -tree, both with basic representations and with DAC.

Let us compare the compression obtained by PDT variants, displayed in the last two columns of Table 2. The space usage of PDT follows patterns completely different from the other alternatives: PDT-hollow is very consistent, using 8–10 bits per point in all the datasets, whereas PDT-RP requires much more space, ranging from 16 to 57 bits per point depending on the dataset. The consistency of PDT-hollow makes it much more efficient than hpqt to represent datasets with no clear regularities in the points, such as clustering. For instance, in the GIS-sparse dataset, PDT-hollow is 3–5 times smaller than the hpqt variants, and in RDF-sparse it is roughly 3–4 times smaller. However, in the Web graph datasets, PDT-hollow is far from the compression offered by hpqt or k^2 -tree variants, becoming 2–4 times larger than our proposal. Note that the space partitioning of hpqt and k^2 -tree is expected to work well on sparse grids with clustered points, whereas PDT does not explicitly consider any of this.

Overall, the results show that hpqt outperforms k^2 -tree in space in almost all cases, achieving good compression especially on Web graphs. Alternatives like PDT-hollow, which do not exploit point regularities, outperform both hpqt and k^2 -tree in space on datasets where the points are distributed more randomly, for example on GIS. We recall, however, that PDT is not designed for representing point grids, as it does not support range queries. The next sections complement this analysis by testing the query performance of all these representations.

7.3. Membership queries

We now test the performance of our technique for membership queries, which check whether a given point exists in the grid or not. We test separately for empty and filled cells, and perform a third test on isolated filled cells. For each

dataset and query type, we build a collection of 100,000 query points. For empty and filled cells we select these points at random, whereas for isolated cells we select the 100,000 points that are farthest away from their closest neighbor. We run each full query set 100 times in each dataset and measure the average.

Figure 4 displays the result of membership queries for empty cells on all the datasets. The datasets are grouped by family, and we describe the tendency of each method in the datasets of the same family using lines. One first general conclusion is that the DAC variants are smaller and faster than those representing all the nodes in bitvectors. In general, the best variants by far are always $\text{hpqt}^{p+\text{DAC}}$ and $k^2\text{-tree}^{\text{DAC}}$, the latter being always slightly smaller and almost always faster, by a smaller or a larger margin, with the exception of the dense GIS datasets. All $k^2\text{-tree}$ and hpqt variants improve in general on sparser or more clustered point sets, because isolated empty cells tend to be higher in the quadtree, but the $k^2\text{-tree}$ exploits this effect better because its search cost is directly proportional to the depth of the leaf sought. Our compressed variants, hpqt^c , are significantly slower than the plain ones, hpqt^p , and provide relevant space-time tradeoffs only on the sparser GIS and RDF datasets, where points distribute uniformly and then the L_d bitvectors tend to have about $2\lg u$ bits per 1. In many cases PDT-hollow offers very attractive space, though it is also significantly slower. The variant PDT-RP is never competitive.

Figure 5 displays the result for filled cells, following the same grouping of datasets used in Figure 4. For these queries, hpqt^p and $\text{hpqt}^{p+\text{DAC}}$ become clearly faster than the $k^2\text{-tree}$ variants, while using similar space. In the sparser GIS and RDF datasets, even the slow compressed variants, hpqt^c and $\text{hpqt}^{c+\text{DAC}}$, outperform the $k^2\text{-tree}$ both in space and time. On the other hand, PDT-hollow and PDT-RP are also much more competitive, especially on the GIS and RDF datasets, where PDT-hollow is still the smallest by far (except on RDF-med) but now its speed is much more competitive. In turn, PDT-RP is by far the fastest in many cases, though it is considerably larger in general. Note that the query times of PDT do not change significantly with respect to Figure 4, whereas accessing filled cells is much more expensive for hpqt and especially for $k^2\text{-tree}$, because filled leaves are always in the deepest level.

Finally, Figure 6 displays the query times for isolated filled cells, where our structure excels, becoming 2–3 times faster than for random filled cells. Now even the bitvector-compressed hpqt variants are faster than the $k^2\text{-tree}$. As in the previous cases, PDT-hollow provides a competitive space-time tradeoff in some datasets, especially on the sparser RDFs, and PDT-RP is sometimes the fastest (though also the largest), but the margin is much narrower than before.

7.4. Range queries

We now consider range queries, which ask for all the points in a defined window of the grid. For these experiments, we selected fixed square window sizes (4, 16, 64, 256, 1024), and for each window size and dataset we built sets of 1,000 random window queries.

Figure 7 displays the query times obtained on the social networks and Web graphs, with varying window size. In this type of queries, hpqt^p , $\text{hpqt}^{p+\text{DAC}}$,

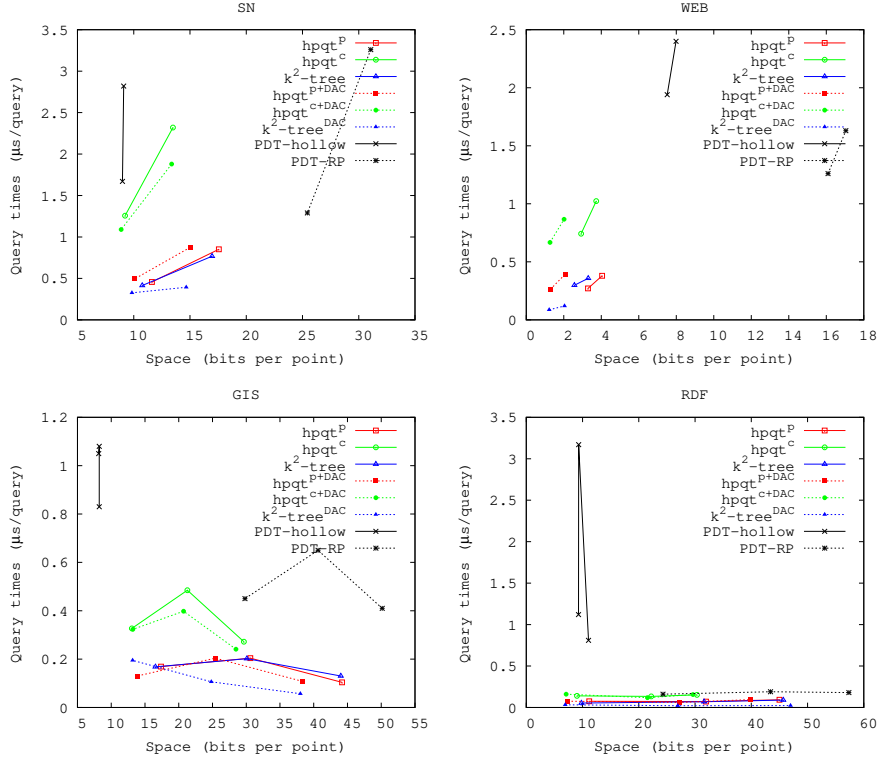


Figure 4: Query times of membership queries for empty cells. Times are in $\mu\text{s}/\text{query}$. Datasets are grouped by family, and lines join the points on different datasets of the same family: SN includes, from left to right, *dblp-enwiki*; WEB includes *indochina-uk*; GIS includes *GIS-dense-GIS-med-GIS-sparse*; RDF includes *RDF-med-RDF-dense-RDF-sparse*.

and $k^2\text{-tree}^{\text{DAC}}$ are always the fastest. On small windows, hpqt is more efficient to reach the deepest node that contains the window, whereas on larger windows $k^2\text{-tree}^{\text{DAC}}$ takes over, generally by a small margin. Note that the bitvector-compressed variant $\text{hpqt}^{\text{c+DAC}}$ is always competitive in time as well. Finally, note that PDT-hollow and PDT-RP are orders of magnitude slower even on the smallest windows, because they do not support range queries and we must resort to individual searches of all the possible points in the query window.

Figure 8 displays the results on the GIS and RDF datasets, which are more difficult to compress. On those, hpqt^{p} and $\text{hpqt}^{\text{p+DAC}}$ are the fastest in almost every case. The sparser datasets, displayed at the top, yield as expected the greatest difference in performance, with hpqt^{p} being 2–4 times faster than $k^2\text{-tree}^{\text{DAC}}$. On the other hand, hpqt^{c} and $\text{hpqt}^{\text{c+DAC}}$ are slower than $k^2\text{-tree}$, but get very close. The PDT variants are again much slower in all range queries.

We can conclude that the hpqt is generally faster than the $k^2\text{-tree}$ at range queries, particularly for smaller windows. The PDT structure is not competitive

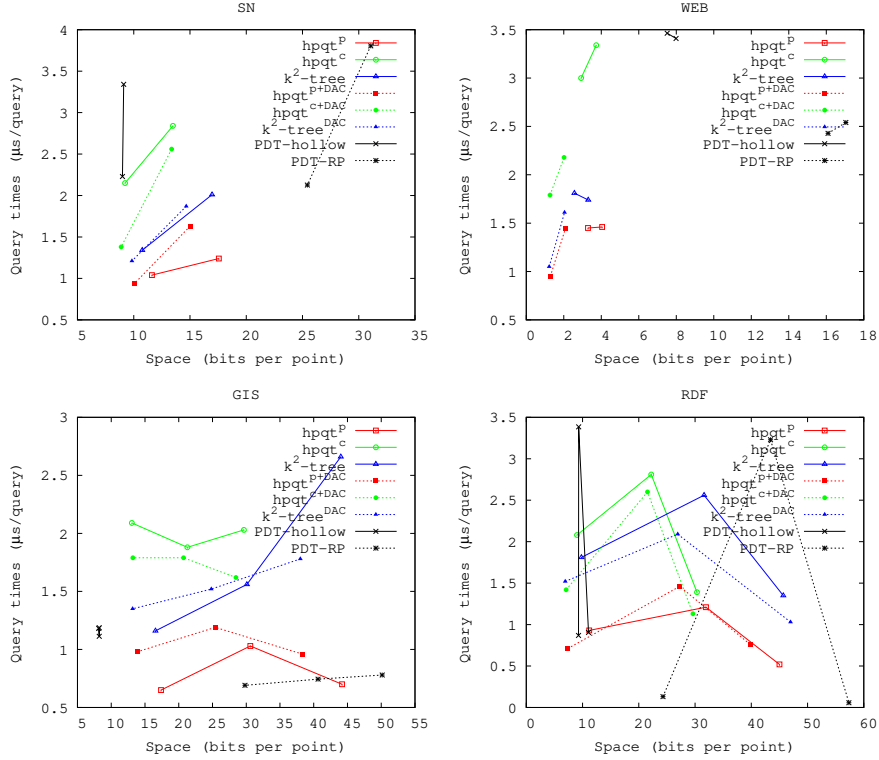


Figure 5: Query times of membership queries for filled cells. Times are in $\mu\text{s}/\text{query}$.

for these queries.

Other kinds of queries, such as row/column queries (requesting all points in a row/column of the grid), are frequent when representing graphs. On those queries the k^2 -tree is slightly faster in general, since the efficiency of the hpqt to locate the submatrix enclosing the query window does not produce any advantage.

7.5. Higher dimensions

Finally, we test the applicability of our proposal to higher dimensions. We compare hpqt with an implementation of the k^3 -tree, the extension of the k^2 -tree to 3 dimensions. We used a set of datasets, mdt500 , mdt700 , and mdtmed , which had previously been evaluated for the k^3 -tree [6]. Those 3-dimensional grids are obtained from elevation rasters, by considering the value stored in the raster of values as the third dimension. Table 3 shows their main characteristics.

We will focus only on the variants including DAC compression ($\text{hpqt}^{p+\text{DAC}}$, $\text{hpqt}^{c+\text{DAC}}$, and $k^3\text{-tree}^{\text{DAC}}$, the k^3 -tree with matrix vocabulary), because the $k^3\text{-tree}^{\text{DAC}}$ is the only available implementation of the k^3 -tree.

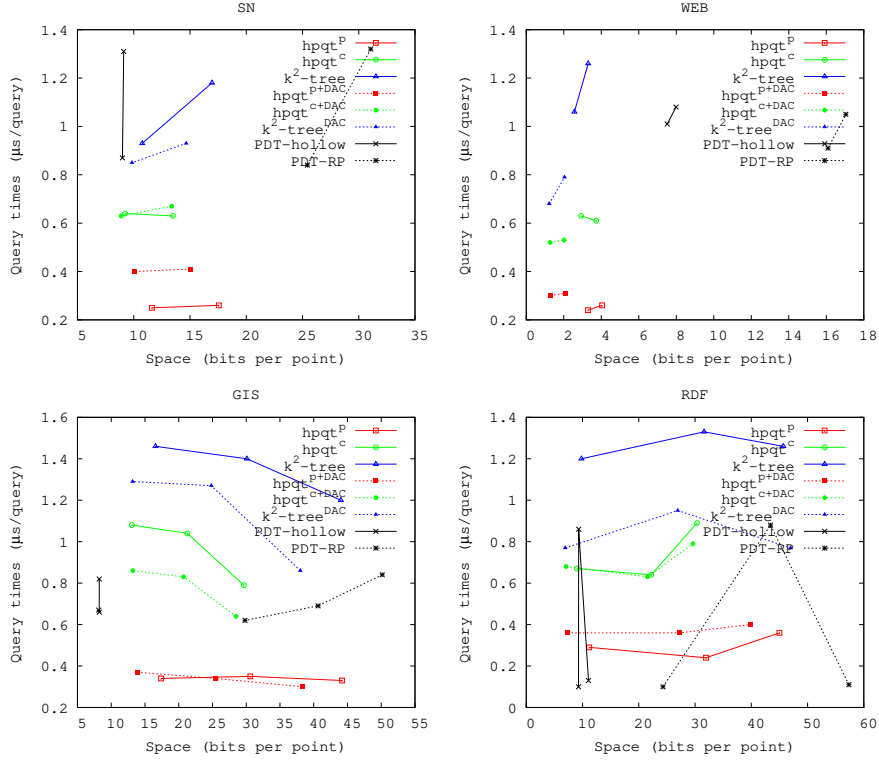


Figure 6: Query times of membership queries for isolated filled cells. Times are in $\mu\text{s}/\text{query}$.

We first study compression and query performance for membership queries. For each dataset, we perform a membership query for each of the points it contains, and measure the average query time. Figure 9 displays the results obtained for all the datasets. The results are similar in all cases: $\text{hpqt}^{p+\text{DAC}}$ and $\text{hpqt}^{c+\text{DAC}}$ are slightly larger than $k^3\text{-tree}^{\text{DAC}}$, but this difference is very small (less than 5% for $\text{hpqt}^{p+\text{DAC}}$, and around 1% for $\text{hpqt}^{c+\text{DAC}}$). On the other hand, both of our solutions are significantly faster than $k^3\text{-tree}^{\text{DAC}}$: $\text{hpqt}^{p+\text{DAC}}$ is about 2.5 times faster, and the compressed variant $\text{hpqt}^{c+\text{DAC}}$ is still 25% faster than $k^3\text{-tree}^{\text{DAC}}$ in all the datasets.

We now analyze the performance on range queries. For each dataset, we run sets of 100,000 random window queries, for different window sizes with the same side in all dimensions: $4 \times 4 \times 4$ to $256 \times 256 \times 256$.

Figure 10 displays the query times on all the datasets, for varying window sizes. As on 2-dimensional data, hpqt variants are faster for small query windows, and all the times become close on larger windows. In particular, $\text{hpqt}^{p+\text{DAC}}$ is significantly faster than $k^3\text{-tree}^{\text{DAC}}$ in all cases for the smallest window sizes. The compressed variant, $\text{hpqt}^{c+\text{DAC}}$, is the slowest, but still close

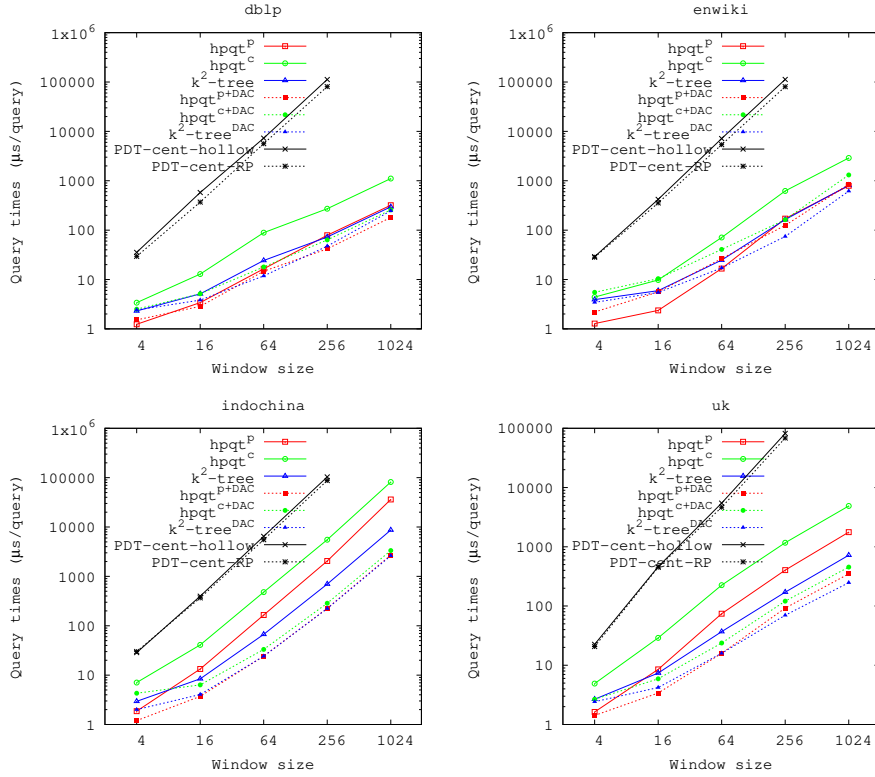


Figure 7: Query times for window queries, with varying window size, for SN (top) and WEB (bottom) datasets. Results are in $\mu\text{s}/\text{query}$. Log-scale is used in both axis.

to $k^3\text{-tree}^{\text{DAC}}$ in all cases.

Overall, we observe in general that the performance gap between hpqt and $k^2\text{-tree}$ widens on three dimensions compared to the two-dimensional case, which is in line with our theoretical expectations.

8. Conclusions

We have introduced a fast space-efficient representation of quadtrees based on heavy-path decompositions, answering in the affirmative to the conjecture of Venkat and Mount [5]. Our structure represents a quadtree on n points in a grid of size $u \times u$ using $\mathcal{O}(1)$ bits per quadtree node, and answers membership queries in $\mathcal{O}(\log n)$ time. Other compressed quadtree representations [4, 5], instead, require $\mathcal{O}(\log u)$ time, which can be significantly higher on sparse grids. We also prove that the space and time of our structure benefits from sparse and clustered point sets, which are common in various applications. Some, but not all, of those benefits extend to other quadtree representations as well.

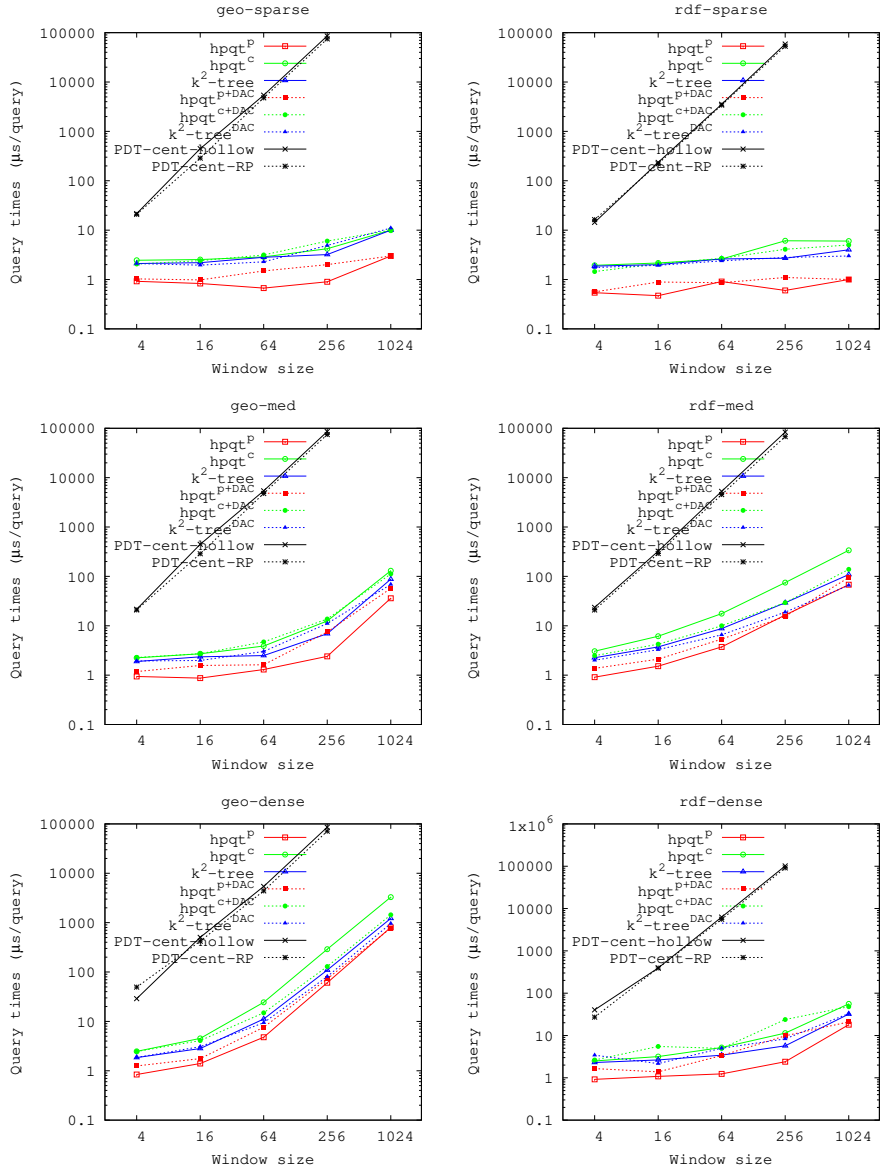


Figure 8: Query times for window queries, with varying window size, for GIS (left) and RDF (right) datasets. Results are in $\mu\text{s}/\text{query}$. Log-scale is used in both axis.

We implemented our structure, demonstrating that it is also practical and competitive. The space requirements of our new representation are similar to other space-efficient representations of quadtrees, such as the k^2 -trees [4], but our structure is typically faster at retrieving existing points, especially isolated

Dataset	Grid size ($r \times c \times d$)	Points
mdt500	$4,001 \times 5,841 \times 578$	23,051,888
mdt700	$3,841 \times 5,841 \times 472$	15,662,092
mdtmed	$7,721 \times 11,081 \times 978$	84,028,401

Table 3: Raster datasets used.

ones. Our structure is also generally faster to handle range queries, and on higher dimensions. Previous structures, instead, are faster when querying large empty areas of the grid.

One future work direction is to explore how the heavy-path decomposition can be used to speed up other more sophisticated queries, like approximate-range and nearest-neighbor searches, by exploiting its ability to efficiently arrive at a desired submatrix.

Another interesting future work challenge is to make our structure dynamic, enabling point insertions and deletions, as done for the k^2 -tree and variants [5, 25, 26]. Every point insertion requires, in principle, marking that a new node has now two children (i.e., flipping a bit in some bitvector L_d) and adding a new path to the representation, somewhere inside bitvector H . Removing a point reverses this process. This can be supported in time $\mathcal{O}(\log u / \log \log u)$ if we use dynamic bitvectors [27], which is also the blowup factor induced on the other operations. Other approaches, which do not affect query times, might be possible [28].

Acknowledgements

Partially funded by the European Union’s Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie grant agreement No 690941. GdB and SL funded by MCIN/AEI/10.13039/501100011033 [grants PID2020-114635RB-I00 (EXTRACompact), PID2019-105221RB-C41 (MAGIST)], by MCIN/AEI/10.13039/501100011033, “NextGenerationEU/PRTR” [grants PDC2021-120917-C21 (SIGTRANS), PDC2021-121239-C31 (FLATCITY-POC)], by GAIN/Xunta de Galicia [grant ED431C 2017/53 (GRC)], and also supported by the Centro de Investigación de Galicia “CITIC”, funded by Xunta de Galicia, FEDER Galicia 2014-2020 80%, SXU 20% [grant ED431G 2019/01 (CSI)]. TG funded by NSERC Discovery Grant RGPIN-07185-2020. GN and DS funded by ANID – Millennium Science Initiative Program – Code ICN17_002, Chile. GN funded by Fondecyt Grant 1-200038, Chile.

References

- [1] G. M. Morton, A computer oriented geodetic data base; and a new technique in file sequencing, Tech. rep., IBM Ltd. (1966).
- [2] H. Samet, Foundations of Multidimensional and Metric Data Structures, Morgan Kaufmann, 2006.

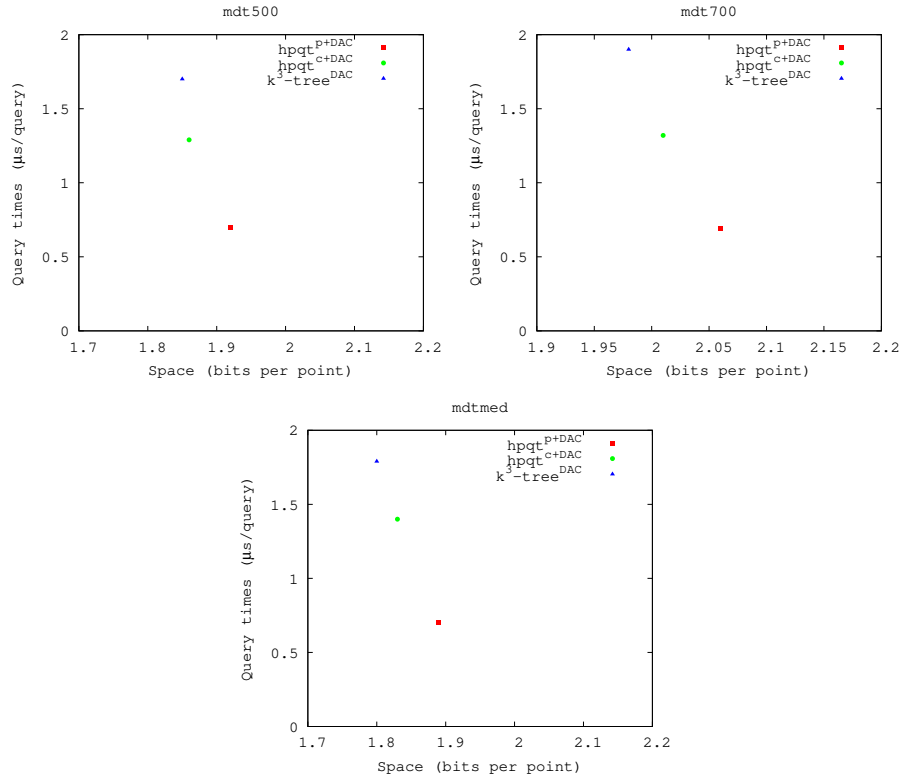


Figure 9: Space and times of membership queries for filled cells. Times are in $\mu\text{s}/\text{query}$.

- [3] I. Gargantini, An effective way to represent quadtrees, *Communications of the ACM* 25 (1982) 905–910.
- [4] N. Brisaboa, S. Ladra, G. Navarro, Compact representation of web graphs with extended functionality, *Information Systems* 39 (1) (2014) 152–174.
- [5] P. Venkat, D. M. Mount, A succinct, dynamic data structure for proximity queries on point sets, in: *Proc. 26th Canadian Conference on Computational Geometry (CCCG)*, 2014, p. article 32.
- [6] N. R. Brisaboa, A. Cerdeira-Pena, G. de Bernardo, G. Navarro, O. Pedreira, Extending general compact querieable representations to GIS applications, *Information Sciences* (2020) 196–216.
- [7] D. Eppstein, M. T. Goodrich, J. Z. Sun, Skip quadtrees: Dynamic data structures for multidimensional point sets, *International Journal of Computational Geometry and Applications* 18 (1/2) (2008) 131–160.
- [8] R. Grossi, G. Ottaviano, Fast compressed tries through path decompositions, *ACM Journal of Experimental Algorithmics* 19 (1).

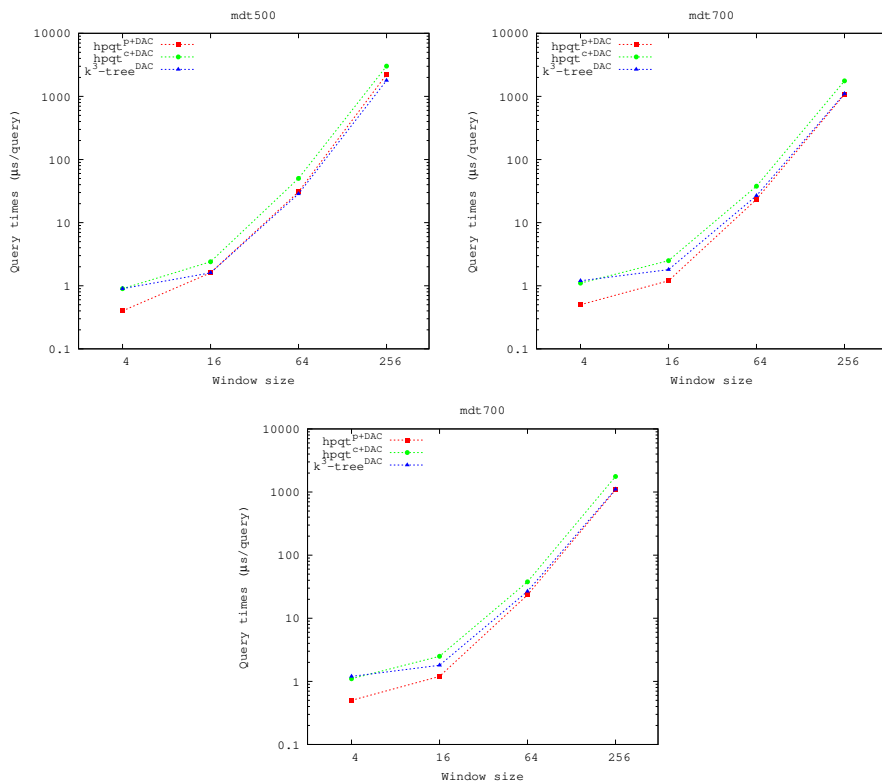


Figure 10: Times of raster queries for varying window sizes. Times are in $\mu\text{s}/\text{query}$. Log-scale is used in both axis.

[9] D. R. Clark, Compact PAT trees, Ph.D. thesis, University of Waterloo, Canada (1996).

[10] J. I. Munro, Tables, in: Proc. 16th Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS), 1996, pp. 37–42.

[11] D. S. Wise, J. Franco, Costs of quadtree representation of nondense matrices, *Journal of Parallel and Distributed Computing* 9 (3) (1990) 282–296.

[12] A. Klinger, Patterns and search statistics, in: *Optimizing Methods in Statistics*, Academic Press, 1971, pp. 303–337.

[13] G. M. Hunter, K. Steiglitz, Operations on images using quad trees, *IEEE Transactions on Pattern Analysis and Machine Intelligence* 1 (2) (1979) 145–153.

[14] G. Navarro, *Compact Data Structures – A practical approach*, Cambridge University Press, 2016.

- [15] N. Brisaboa, S. Ladra, G. Navarro, DACs: Bringing direct access to variable-length codes, *Information Processing and Management* 49 (1) (2013) 392–404.
- [16] N. Berezky, A. Duch, K. Németh, S. Roura, Quad-K-d trees, in: *Proc. 11th Latin American Symposium on Theoretical Informatics (LATIN)*, 2014, pp. 743–754.
- [17] D. D. Sleator, R. E. Tarjan, A data structure for dynamic trees, *Journal of Computer and System Sciences* 26 (3) (1983) 362–391.
- [18] M. L. Fredman, D. E. Willard, Trans-dichotomous algorithms for minimum spanning trees and shortest paths, *Journal of Computer and System Sciences* 48 (3) (1994) 533–551.
- [19] M. Pătraşcu, M. Thorup, Time-space trade-offs for predecessor search, in: *Proc. 38th Annual ACM Symposium on Theory of Computing (STOC)*, 2006, pp. 232–240.
- [20] D. Okanohara, K. Sadakane, Practical entropy-compressed rank/select dictionary, in: *Proc. 9th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2007, pp. 60–70.
- [21] D. E. Knuth, *The Art of Computer Programming*, volume 4: Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams, Addison-Wesley Professional, 2009.
- [22] R. Raman, V. Raman, S. Rao, Succinct indexable dictionaries with applications to encoding k -ary trees, prefix sums and multisets, *ACM Transactions on Algorithms* 3 (4) (2007) 43.
- [23] P. Boldi, S. Vigna, The WebGraph framework I: Compression techniques, in: *Proc. 13th International World Wide Web Conference (WWW)*, 2004, pp. 595–601.
- [24] P. Boldi, M. Rosa, M. Santini, S. Vigna, Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks, in: *Proc. 20th International Conference on World Wide Web (WWW)*, 2011, pp. 587–596.
- [25] N. Brisaboa, A. Cerdeira-Pena, G. de Bernardo, G. Navarro, Compressed representation of dynamic binary relations with applications, *Information Systems* 69 (2017) 106–123.
- [26] D. Arroyuelo, G. de Bernardo, T. Gagie, G. Navarro, Faster dynamic compressed d -ary relations, in: *Proc. 26th International Symposium on String Processing and Information Retrieval (SPIRE)*, 2019, pp. 419–433.
- [27] G. Navarro, K. Sadakane, Fully-functional static and dynamic succinct trees, *ACM Transactions on Algorithms* 10 (3) (2014) article 16.

- [28] M. E. Coimbra, J. Hrotkó, A. P. Francisco, L. M. S. Russo, G. de Bernardo, S. Ladra, G. Navarro, A practical succinct dynamic graph representation, *Information and Computation* 285B (2022) article 104862.