Faster Compressed Quadtrees

Travis Gagie[1], Javier I. González-Nova[2],
Susana Ladra[3], Gonzalo Navarro[4] and Diego Seco[2]

[1] Helsinki Institute for Information Technology (HIIT) and
Department of Computer Science, University of Helsinki, Finland
[2] Department of Computer Science, University of Concepción, Chile
[3] Database Laboratory, University of A Coruña, Spain
[4] Department of Computer Science, University of Chile, 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 bound the number of nodes in the quadtree for a point set in terms of the points' clustering. We then describe a quadtree data structure that uses $\mathcal{O}(1)$ bits per node and supports faster queries than previous structures with this property. Finally, we present experimental evidence that our structure is practical.

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 sparsely and randomly over the grid, but we can often do better on real-word point sets because they tend to be clustered and, therefore, compressible.

Quadtrees [12] tend to have $o(n \log u)$ nodes when the points are clustered, but pointer-based quadtree data structures can still take $\Omega(n \log u)$ bits. One way to avoid storing pointers is to store the points' coordinates instead [9], but that also takes $\Omega(n \log u)$ bits. Hudson [11] gave a structure that uses $\mathcal{O}(n)$ bits when the points are spaced appropriately and we are willing to tolerate some distortion of the points' positions. Recently, de Bernardo et al. [7] and Venkat and Mount [16] independently proposed similar structures based on static and dynamic succinct tree representations, respectively (see, e.g., [1, 6, 13]). Both structures use $\mathcal{O}(1)$ bits per node in the quadtree and have the same asymptotic query times as traditional structures, which support only edge-by-edge navigation. Venkat and Mount noted, however, that

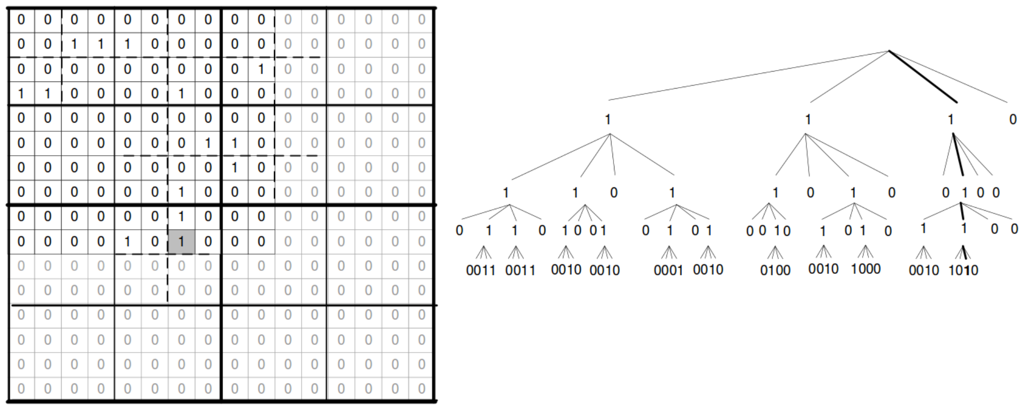

Figure 1: A set of points, indicated by 1s, on a 16×16 grid (left); the quadtree for those points (right). The heavy lines in the quadtree indicate the path to the leaf corresponding to the shaded point on the grid.

> "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."

In Section 2 we review the ideas behind quadtrees and prove a simple upper bound on the number of nodes in terms of the points' clustering. In Section 3 we describe a quadtree data structure that uses $\mathcal{O}(1)$ bits per node in the quadtree and allows us to move over multiple edges at once. In Section 4 we show how this lets us perform faster membership queries. Finally, in Section 5 we present experimental evidence that our structure is practical. We leave as future work making our structure dynamic.

2 Space Bound

Let P be a set of n points on a $u \times u$ grid. If n is 0 or u^2 points, then the quadtree for the grid is only a root storing either 0 or 1, respectively; otherwise, the root stores 1 and has four children, which are the quadtrees of the grid's four quadrants. Figure 1 shows an example, taken from [7]. 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 $10\,01\,01\,10$. 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 $1\mathit{0}\,0\mathit{1}\,0\mathit{1}\,1\mathit{0}$. We can interleave a point's coordinates in $\mathcal{O}(1)$ time using, e.g., pre-computed tables.

The quadtree has height at most $\lg u$ and $\mathcal{O}(n \log u)$ nodes, and a subtree rooted at depth d encodes the points on a $2^{\lg u - d} \times 2^{\lg u - d}$ square. Given a query rectangle R, we can find all the points in $P \cap R$ by starting at the root and visiting all the nodes whose subtrees' squares overlap R, recording the leaves storing 1s. This is called range reporting or, in the special case $R = [x, x + 1) \times [y, y + 1)$, a membership test for (x, y). If we report k points then we visit $\mathcal{O}(k \log u)$ nodes. If the points in 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.

Theorem 1. *Suppose we can partition P into c clusters, not necessarily disjoint, with $n_1, \ldots, n_c$ points and diameters $\ell_1, \ldots, \ell_c$. Then the quadtree has $\mathcal{O}(c \log u + \sum_i n_i \log \ell_i)$ nodes.*

Proof. Let S be an $\ell \times \ell$ square on the grid, let $C = S \cap P$, and let A be the set of ancestors in the quadtree of the points in C. (For simplicity, we identify points in P with their corresponding leaves in the quadtree.) Let A' be the ancestors of only the corners of S (which may or may not be in P). Notice

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

For any ancestor v of a point in C that has depth at most $\lg(u/\ell)$, v's subtree contains 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, so $v \in A'$. It follows that

$$|A \backslash A'| \leq |C| (\lg u - \lg(u/\ell)) = |C| \lg \ell\,,$$

so $|A| < |C| \lg \ell + 4 \lg u$. □

The proof above is something like a two-dimensional analogue of Gupta, Hon, Shah and Vitter's [10] analysis of tries. In the full version of this paper we will consider higher dimensions and give bounds with respect to hierarchical clustering.

3 Structure

The structure by de Bernardo et al. [7] mentioned in Section 1 is a variation of Brisaboa, Ladra and Navarro's [5] k^2-tree structure. Brisaboa, Ladra and Navarro designed k^2-trees to compress the Web graph, and de Bernardo et al. adapted it to other domains, such as geographic data. The main difference is that if P contains all the $2^{\lg u - d} \times 2^{\lg u - d}$ points encoded by the subtree of a node at depth d, then in de Bernardo et al.'s structure that subtree is only the node itself, conforming to the definition of a quadtree; in Brisaboa, Ladra and Navarro's structure, that subtree has height $\lg u - d$ and $2^{2(\lg u - d)}$ leaves. Thus, the original k^2-tree can have more nodes than the quadtree.

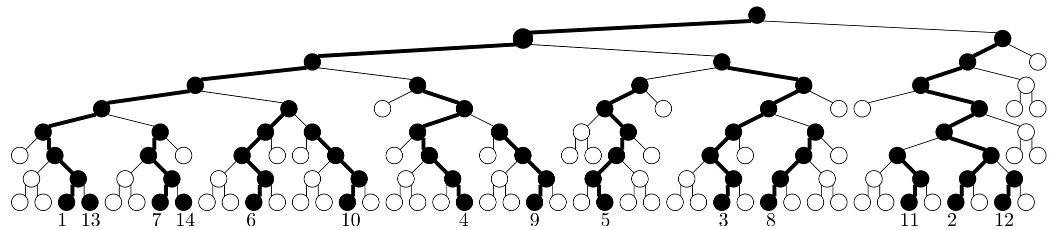

Figure 2: The heavy-path decomposition of the binary tree for the example from Figure 1. Nodes storing 1s are black; nodes storing 0s are shown hollow, and discarded; thick edges belong to heavy paths. The numbers below the black leaves indicate our ordering of the paths.

For many applications it is rare that point sets contain large squares that are completely filled. Therefore, in this version of this paper we make the simplifying assumption that each internal node of the quadtree has at least one descendant storing a 0, so both versions of the k^2-tree have the same number of nodes as the quadtree. We will remove this assumption in the full version.

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; this increases the size of the whole tree by a factor of at most 7/5. Let T be the resulting binary tree. In addition to simplifying our construction, this modification makes quadtrees more practical in higher dimensions [2], which we will also consider in the full version of this paper.

We then perform a heavy-path decomposition [15] of T. That is, we partition T into root-to-leaf paths, called heavy paths, such that the path containing a node v also contains the child of v with the most leaf descendants (breaking ties arbitrarily). One well-known property of this decomposition is that each root-to-leaf path in T consists of $\mathcal{O}(\log n)$ initial segments of heavy paths. Figure 2 shows the heavy-path decomposition of the binary tree for our example from Figure 1.

We encode each heavy path h as a binary string whose 0s and 1s indicate which of h's nodes are left children and which are right children (considering the root as a left child, say, for simplicity) in increasing order by their depths. We sort the encodings into decreasing order by length, breaking ties such that if two paths h and h' have the same length and their topmost nodes are v and v', then the encodings of h and h' appear in the same order as the encodings of the paths containing the parents of v and v'. (Notice 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. We store the concatenation H of the paths' encodings, which consists of $|T|$ bits. We say 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 $d < \lg u$ (considering the root to have depth 0 and leaves to have depth $\lg u$), we store a bitvector L_d with 1s indicating which nodes at that depth in T have two children; see, e.g., [14] for a discussion of bitvectors. These bitvectors have as many bits as there are internal nodes in T. For our example,

$$H \;=\; \texttt{000000110 10010100 1100010 110111 001001 10010 1010 1000 1110 1110 010 10 1 1}\,,$$

L_0 = `1--------`,
L_1 = `-1------- 0-------`,
L_2 = `--1------ -0------ 1------`,
L_3 = `---1----- --0----- -0----- 0----- 0-----`,
L_4 = `----1---- ---0---- --1---- -1---- -0---- 1----`,
L_5 = `-----0--- ----1--- ---0--- --0--- --0--- -0--- 0--- 0--- 0--- 0---`,
L_6 = `------0-- -----1-- ----0-- ---0-- ---0-- --0-- -0-- -0-- -0-- -0-- 0--`,
L_7 = `-------1- ------0- -----0- ----0- ----0- ---0- --1- --0- --0- --0- -0- 0-`;

dashes and spaces are only for legibility. Storing H and all the L_ds takes $\mathcal{O}(1)$ bits per node in T and, therefore, also $\mathcal{O}(1)$ bits per node in the original quadtree. For each length, we also store the starting position in H of the first encoding with that length; this also takes a total of $\mathcal{O}(T)$ bits. In the full version of this paper we will give more details and discuss ways in which we can slightly reduce our space usage.

Suppose $H[i]$ corresponds to node v in T. Given i, in $\mathcal{O}(1)$ time we can determine the length of the heavy path containing v, that path's rank in our ordering, v's depth in T, and whether v is the top node in its path. If v is the top node in the jth path starting at depth d then, by our choice of ordering, v's parent u is the jth node at depth $d-1$ that L_{d-1} indicates has two children. It follows that we can compute in $\mathcal{O}(1)$ time which bit of H corresponds to u, via a select query on L_{d-1} and some arithmetic. (Since select queries are quite slow in practice, however, even in applications involving ascents it may be better for us simply to backtrack.) For example, if v is the top node in the ninth path in our ordering, which is the third path starting at depth 5, then u is the third node at depth 4 which L_4 indicates has two children. Since the third 1 in L_4 is its fourth bit, u is the node at depth 4 in T in the fourth path in our ordering.

Once we know v's depth d in T and its path's ranking in our ordering, we know immediately whether v is a leaf and we can check L_d to see whether v has two children. If v is the jth node at depth d which has two children then the child of v that is not in the same path as v, is the top node in the jth path starting at depth $d+1$. It follows that we can compute in $\mathcal{O}(1)$ time which bit of H corresponds to w, via a rank query on L_d and some arithmetic; the other child of v corresponds to $H[i+1]$, and that bit tells us which child is which. Reversing the example in the previous paragraph, if v is the node at depth 4 in T in the fourth path in our ordering, then it is the third node at depth 4 which L_4 indicates has two children. Therefore, one of v's children is the node at depth 5 in the same path, while v's other child is the top node in the third path starting at depth 5, which is the ninth path in our ordering.

4 Membership

Suppose we want to perform a membership query for (x, y). We compute the label on the path to (x, y) in $\mathcal{O}(1)$ time, as described in Section 1. We set v to be the root of T, then repeat the following steps until we reach (x, y) or can descend no further: we find the longest common prefix of the remainder of the path label for (x, y) and the encoding of the heavy path starting at v (except that we ignore the first bit of H, which is 0 and corresponds to the root), which takes $\mathcal{O}(1)$ time because the path label and the encoding are $\mathcal{O}(\log u)$ bits; we descend the initial segment of the heavy path encoded by that common prefix; if we reach (x, y), then we report $(x, y) \in P$; if the node we are currently visiting has only one child, then we report $(x, y) \notin P$; otherwise, we set v to be the child of the node we are currently visiting that is not in the same heavy path, and continue. In total, the query takes time proportional to the number of initial segments we traverse. Since we only descend, we never need select queries.

To perform a membership query for $(6, 9)$ in our example, we compute the path label 10010110 and set v to be the root; we find that this label does not share any non-empty prefix with the encoding of the heavy path starting at v (ignoring the leading 0 because v is the root, although in this case it makes no difference); and we set v to be the root's right child. We then find that the remainder of the path label (which is all of it) shares a prefix of length 6, 100101, with the encoding of the heavy path starting at v; we descend to the 6th node on that path; we set v to be that node's right child. Finally, we find the remainder of the path label, 10, shares a prefix of length 2, all of 10, with the encoding of the heavy path starting at v; we descend to the 2nd node on that path; and we report $(6, 9) \in P$.

Since each root-to-leaf path in T consists of $\mathcal{O}(\log n)$ initial segments of heavy paths, our data structure obviously supports membership queries in $\mathcal{O}(\log n)$ time. If the query point is isolated, we use even less time.

Theorem 2. *We can store P using $\mathcal{O}(1)$ bits per node in the quadtree such that a membership query for (x, y) takes $\mathcal{O}(\min_g\{\log(u/g) + \log k_g\}) \subseteq \mathcal{O}(\log n)$ time, where k_g is the number of points in P within distance g of (x, y).*

Proof. Any node v at depth at least $2\lg(u/g)+2$ in T whose subtree's square contains (x, y), has at most k_g leaf descendants. It follows that the path from v to the deepest node w of T whose subtree's square contains (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 top-most 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. □

Combining Theorems 1 and 2 suggests 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 3. *Suppose we can partition a set P of n points into c clusters, not necessarily disjoint, with $n_1, \ldots, n_c$ points and diameters $\ell_1, \ldots, \ell_c$. Then we can store P in $\mathcal{O}(c \log u + \sum_i n_i \log \ell_i)$ bits such that a membership query for (x, y) takes $\mathcal{O}(\min_g\{\log(u/g) + \log k_g\}) \subseteq \mathcal{O}(\log n)$ time, where k_g is the number of points in P within distance g of (x, y).*

5 Experiments

We have implemented the data structure from Section 3 and compared it experimentally with the k^2-tree. Due to space constraints, we present the results only for membership queries; in the full paper we will present results for range reporting. All the experiments presented here were performed in an Intel Core i7-3820@3.60GHz, 32GB RAM, running Ubuntu server (kernel 3.13.0-35). We compiled with gnu/g++ version 4.6.3 using -O3 directive.

In our implementation we make use of the bitvector implementations available in LibCDS (https://github.com/fclaude/libcds). Specifically, we use three types of bitvectors. Our first variant, a heavy-path implementation with plain bitvectors, we call HP^p; for our second variant, named HP^c, we use compressed bitmaps. LibCDS provides several implementations of compressed bitmaps, and we use either the Raman, Raman and Rao (RRR) implementation or Sadakane's SDArray (for each dataset, we select the one achieving better compression).

We compare these two variants with two configurations of k^2-trees. The first variant, named k^2-tree^b, consists of a basic version of the k^2-tree where the degree $k = 2$ for all the levels of the tree. The second variant, named k^2-tree^h, is the configuration considered as optimal for the k^2-tree [5], which includes a hybrid approach with different k values for the levels of the tree and a vocabulary of leaf submatrices to obtain better compression. We did not compare to Venkat and Mount's structure because they have not yet made an implementation available. We also did not compare to the classical quadtree representations since the k^2-tree is an order of magnitude smaller and has better access time [7].

For our experimental evaluation we use grid datasets from different domains: geographic information systems (GIS), social networks (SN), web graphs (WEB) and RDF datasets (RDF). For GIS data we use the Geonames dataset, which contains more than 9 million populated places, and convert it into three grids with different resolutions: Geo-sparse, Geo-med, and Geo-dense. (The higher the resolution, the sparser the matrix.) For SN and WEB we consider the grid associated with the adjacency matrix of two Web graphs (indochina-2004, uk-2002) and two social networks (dblp-2011, enwiki-2013) obtained from the Laboratory for Web Algorithmics[1] [4, 3]. Finally, we use RDF data obtained from the dbpedia dataset[2]. This RDF dataset contains triples (S,P,O) indicating subjects that are related to objects with a specific predicate. Thus, each predicate defines a binary relation among subjects and objects that can be represented in a grid with points. We create three different grids for our experiments, selecting predicates with different numbers of related objects (triples-sparse, triples-med, and triples-dense).

Table 1 gives the main characteristics of the datasets used: name of the dataset, size of the grid (u), number of points it contains (n) and the space achieved by the four representations compared: k^2-tree^b, k^2-tree^h, HP^p, and HP^c. The space is measured in bits per points (bpp), dividing the total space of the structure by the number

[1]http://law.dsi.unimi.it

[2]http://wiki.dbpedia.org/Downloads351

Table 1: Description of the datasets and space comparison

File	Type	Grid (u)	Points (n)	Space (bpp) k^2-tree^b	k^2-tree^h	HP^p	HP^c
Geo-dense	GIS	524,288	9,188,290	16.68	**13.27**	18.50	15.34
Geo-med	GIS	4,194,304	9,328,003	30.27	24.97	31.77	**21.84**
Geo-sparse	GIS	67,108,864	9,335,371	44.19	39.67	45.36	**28.55**
dblp-2011	SN	986,324	6,707,236	10.76	**9.84**	12.62	10.69
enwiki-2013	SN	4,206,785	101,355,853	16.96	**14.66**	18.56	15.33
indochina-2004	WEB	7,414,866	194,109,311	2.57	**1.22**	4.29	4.09
uk-2002	WEB	18,520,486	298,113,762	3.30	**2.04**	5.04	4.94
triples-dense	RDF	66,973,084	98,714,022	9.80	**6.93**	12.19	10.40
triples-med	RDF	66,973,084	7,936,138	31.61	26.95	32.94	**23.26**
triples-sparse	RDF	66,973,084	138,303	45.69	46.98	45.96	**29.97**

of points (n) in the grid. We can observe that HP^P obtains the worst compression among all the alternatives, but HP^c obtains better results than k^2-tree^h for some of the datasets, which is remarkable as this configuration of the k^2-tree exploits several compression techniques that may be considered for future extensions of this proposal and may allow the HP^c variant to reduce its space. HP^c clearly outperforms k^2-tree^h for very sparse grids.

We now analyze the time performance of our proposed structure. We distinguish three different types of membership queries: empty cells, filled cells and isolated filled cells (top 100,000 most isolated filled cells), and measure average times per query in nanoseconds. We show in Figure 3 the results obtained by the four representations over two kinds of grid datasets, GIS and SN. (Due to space constraints we omit results for the WEB and RDF datasets.) We can observe that for both scenarios we obtain similar performance. The k^2-tree representation obtains better results when querying empty cells, as the computation for reaching a zero node in the k^2-tree is lighter than using heavy paths. However, HP becomes the best alternative when querying filled cells: our non-compressed data structure is always the fastest one for cells with values, and much faster for isolated points. In this latter case, even the compressed variant of our structure outperforms the most optimized k^2-tree both in time and space.

6 Conclusions

We have presented a fast space-efficient representation of quadtrees, answering in the affirmative to the conjecture of Venkat and Mount [16]. Our structure has nice theoretical bounds and it is practical. Space requirements are similar to other space-efficient representations of quadtrees, e.g., the k^2-trees, but our structure is faster handling isolated filled cells. In the full version of this paper we will generalize our structure to higher dimensions, give bounds in terms of hierarchical clustering and present experimental results for range reporting. It would be interesting to study also other types of queries, e.g., approximate-range and nearest-neighbour searching.

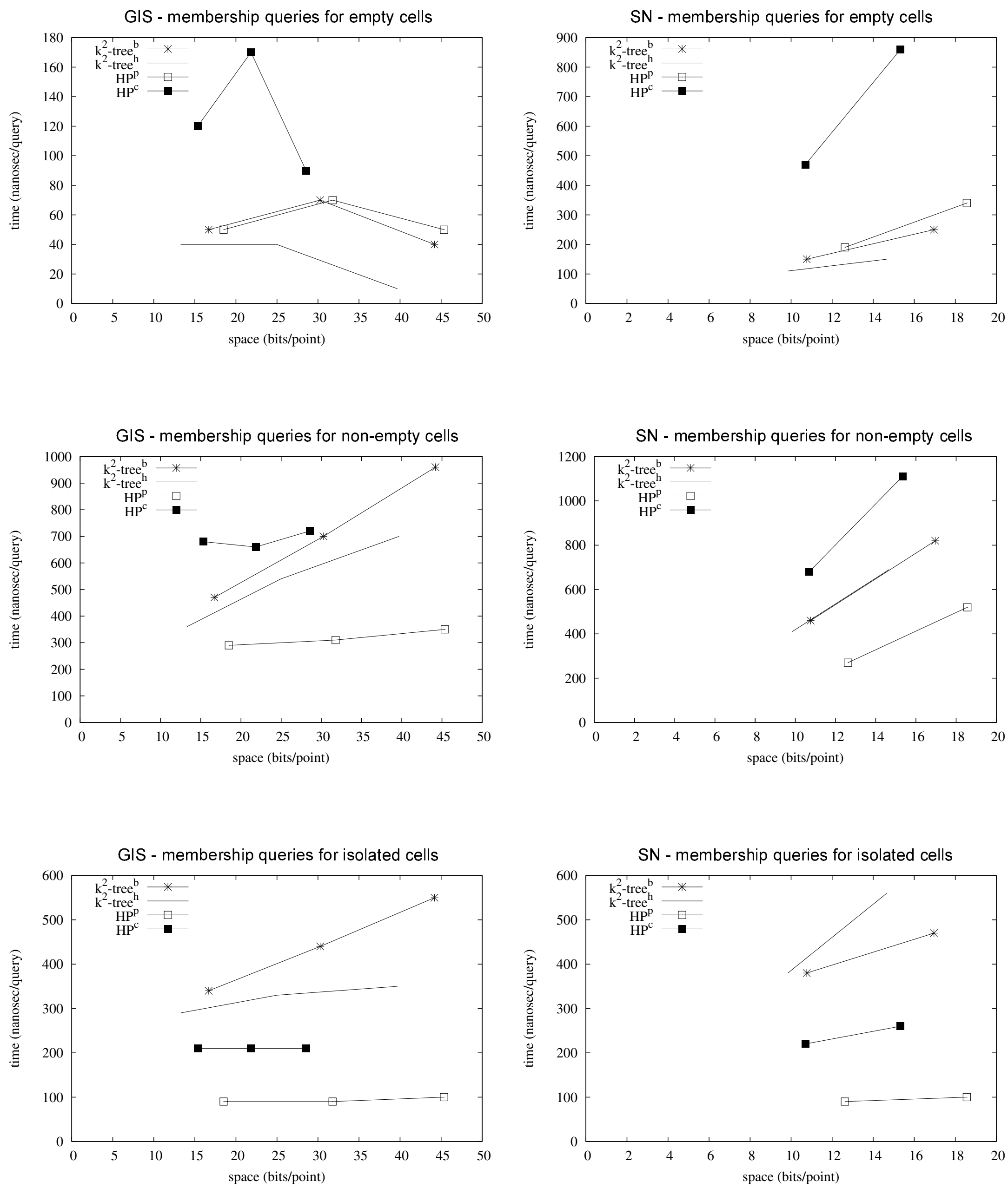

Figure 3: Space and time usages of our HP methods and the k^2-tree variants, for the GIS (left) and SN (right) datasets and each type of membership query: empty cells (top), filled cells (center) and isolated filled cells (bottom). Curves for the GIS datasets show results for Geo-sparse, Geo-med, and Geo-dense; curves for the SN datasets show results for dblp-2011 and enwiki-2013. For example, consider the graph for the GIS dataset and queries on isolated filled cells (bottom left): the middle point on the curve for HP^c is to the left of and below the bend in the curve for k^2-tree^h; this means that, for the Geo-med dataset, the HP^c structure uses less space than the k^2-tree^h structure and also answers faster. (Notice the query type does not affect space usages.)

Acknowledgments

Many thanks to Timothy Chan and Yakov Nekrich for directing us toward Venkat and Mount's paper. The first author is also grateful to the late Ken Sevcik for introducing him to some concepts used in this paper.

This work was partially funded by the Academy of Finland grant 268324 (first author), Conicyt Fondecyt grant 11130377 (second and fifth authors), MINECO grant TIN2013-46801-C4-3-R) and Xunta de Galicia grant GRC2013/053 (third author), and Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F (fourth author).

References

[1] D. Arroyuelo, R. Cánovas, G. Navarro, and K. Sadakane. Succinct trees in practice. In *Proc. ALENEX*, pages 84–97, 2010.

[2] N. Bereczky, A. Duch, K. Németh, and S. Roura. Quad-K-d trees. In *Proc. LATIN*, pages 743–754, 2014.

[3] P. Boldi, M. Rosa, M. Santini, and S. Vigna. Layered label propagation: A multiresolution coordinate-free ordering for compressing social networks. In *Proc. WWW*, pages 587–596, 2011.

[4] P. Boldi and S. Vigna. The WebGraph framework I: Compression techniques. In *Proc. WWW*, pages 595–601, 2004.

[5] N. R. Brisaboa, S. Ladra, and G. Navarro. Compact representation of web graphs with extended functionality. *Info. Sys.*, 39:152–174, 2014.

[6] P. Davoodi and S. S. Rao. Succinct dynamic cardinal trees with constant time operations for small alphabet. In *Proc. TAMC*, pages 195–205, 2011.

[7] G. de Bernardo, S. Álvarez-García, N. R. Brisaboa, G. Navarro, and O. Pedreira. Compact querieable representations of raster data. In *Proc. SPIRE*, pages 96–108, 2013.

[8] F. E. Fich. Constant time operations for words of length w. 1999.

[9] I. Gargantini. An effective way to represent quadtrees. *CACM*, 25:905–910, 1982.

[10] A. Gupta, W.-K. Hon, R. Shah, and J. S. Vitter. Compressed data structures: Dictionaries and data-aware measures. *TCS*, 387:313–331, 2007.

[11] B. Hudson. Succinct representation of well-spaced point clouds. Technical Report abs/0909.3137, CoRR, 2009.

[12] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, IBM Ltd., 1966.

[13] G. Navarro and K. Sadakane. Fully functional static and dynamic succinct trees. *TALG*, 10:16, 2014.

[14] M. Pătraşcu. Succincter. In *Proc. FOCS*, pages 305–313, 2008.

[15] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *JCSS*, 26:362–391, 1983.

[16] P. Venkat and D. M. Mount. A succinct, dynamic data structure for proximity queries on point sets. In *Proc. CCCG*, 2014. To appear.