Dynamic point location in fat hyperrectangles with integer coordinates

John Iacono* iacono@cs.rutgers.edu and Stefan Langerman lfalse@cs.rutgers.edu

Department of Computer Science
Rutgers University – New Brunswick
Piscataway NJ 08854

Abstract

A data structure is presented for point location in a disjoint set of hyperrectangles in arbitrary fixed dimension with integer coordinates less than U. Point location query times are $O(\log\log U)$, in any dimension. The space and construction costs are dependent of the fatness of the hyperrectangles stored in the structure. The fatness of a hyperrectangle x is defined to be the size of the smallest set of hypercubes whose union is x. Given a set of n hyperrectangles with average fatness f the time to construct the data structure is $O(fn\log U\log\log U)$ randomized, and the structure will use $O(fn\log\log U)$ space. Semidynamic and fully dynamic variants of this data structure are also presented.

1 Introduction

Point location is one of the fundamental problems in computational geometry. In a point location problem, we are given a set of disjoint regions in space and are required to, given a query point, identify which region, if any, the point lies in. Point location data structures are evaluated by three factors: The time to build the structure, the size of the structure, and the query time. In the plane, the problem has been solved optimally for a collection of disjoint polygons, with $O(\log n)$ query time and $O(n)$ space and construction cost ([8] using [1] for preprocessing). There has been little success for general higher dimensional objects. Pellegrini [9] presents a data structure that allows queries in d dimensional polyhedra defined by n $(d-1)$-dimensional simplices in time $O(\log n)$, using $O(n^{d-1})$ space and preprocessing. However, in the restriction of the problem to axis-parallel hyperrectangles, several data structures with reasonable space requirements have been proposed. Edelsbrunner, Haring and Hilbert [5] were the first to look at this problem, producing a linear time and space data structure based on the skewer tree that supported queries in a set of n disjoint d dimensional rectangles in time $O(\log^{d-1} n)$. By further restricting the hyperrectangles to those with nonnegative integer coordinates less than U, improvements based on perfect hashing [6] have been made. De Berg, van Kreveld and Snoeyink [3] used linear time and space to support $O((\log\log U)^2)$ queries in two dimensions. In three dimensions and linear space their data structure supported $O(\log^{\epsilon} U)$ queries for any $\epsilon > 0$. By increasing the construction time and space to $O(n\log\log U)$ they, in three dimensions, support $O((\log\log U)^3)$ queries.

In this work we present a static data structure for rectangular point location in arbitrary fixed dimension, where the coordinates of the hyperrectangles are integers less than U. The fatness of a hyperrectangle x, $F(x)$ is defined to be the size of the smallest set of hypercubes whose union is x. Our data structure implements point location in a set of hyperrectangles S, supporting $O(\log\log U)$ queries, and $O(\sum_{x\in S} F(x)\log\log U)$ space and $O(\sum_{x\in S} F(x)\log U\log\log U)$ randomized construction cost. The query times are worst case, and do not depend on fatness. The construction time is randomized because of the use of perfect hashing.

Our data structure is significant because it supports $O(\log\log U)$ queries in any dimension with reasonable space and construction times. This data structure was motivated by the y-fast trie of Willard [13], which can be viewed as supporting one-dimensional point location with $O(\log\log U)$ query time and $O(n)$ space.

Two dynamic variants of our structure that utilize dynamic perfect hasing [4] are presented. As in the strucutre decribed above, both are also used for point location in axis-parallel hyperrectangles with integer coordinates less than U in any fixed dimension. The first is semidynamic, allowing insertion of an object with fatness g in expected amortized time $O(g\log U\log\log U)$. The space required is increased

*Research supported by NSF grant CCR-9732689

to $O(fn \log U)$, where f is the average fatness of the hyperrectangles stored, and n is the current number of items in the structure. The query time remains $O(\log\log U)$. The second is a fully dynamic point location structure, where the time to insert or delete a region of fatness g is expected amortized $O(g)$, and the time to process a point location query is $O(\log U)$. The space required by this structure is $O(fn)$, when n regions of average fatness f are being stored.

The dynamic point location problem for axis-parallel rectangles has been studied before by Smid [10], where the coordinates were not restricted to a range of integers. A fully dynamic method with $O(\log^{(d-1)} n \log\log n)$ query time, and $O(\log^2 n \log\log n)$ insertion and deletion time, while using $O(n)$ space was presented. Smid also showed that if deletion was forbidden, the data structure could be changed so that the $\log\log n$ factors could be removed from the running times. Our insertion-only method has a better query time than Smid's when $n > \log U$, while our fully dynamic method's expected amortized constant insertion and deletion time for fat integer coordinate hyperrectangles in any dimension is advantageous over known methods.

2 Justified hyperrectangles

Our basic data structure does not work on hypercubes or fat hyperrectangles, but rather on objects called justified hyperrectangles, defined below.

Definition 1 *A justified hyperrectangle is defined to be a hyperrectangle formed by the Cartesian product of the closed intervals* $[l_1, r_1] \times \cdots \times [l_d, r_d]$ *if there exists an integer g for which the following two conditions hold:*

1. *For all values of i, for 1 to d, in the binary representation of l_i, the last g digits are 0.*
2. *For all values of i, r_i may be obtained by adding a g digit binary number to l_i.*

Given a justified hyperrectangle s, the g value described above is defined to be $G(s)$ (If multiple values of can be used for g in the above, $G(s)$ is defined to be the largest such value). The vector $(l_1, \ldots, l_d)$ is defined to be the representative of of s, denoted $R(s)$.

Definition 2 *The encoding of a point $p = (p_1, p_2, \ldots p_n)$, $\langle p \rangle$ is defined to be $\Sigma_{i=1}^{d} p_i U^{i-1}$. For fixed d, $\langle p \rangle$ can be computed in time $O(1)$.*

Definition 3 *Given a number i, $H(i)$ is defined to be the set of integers not less than i that are accessed when doing a binary search for i in the region $0 \ldots \lceil \log U \rceil$. Note that $|H(i)|$ is $O(\log\log U)$ and can be computed in time $O(\log\log U)$.*

Definition 4 *Given a point p, $S(p, i)$ is the point p with the last i bits in every dimension zeroed out.*

Fact 5 *Given any point p in a justified hyperrectangle x, $S(p, G(x)) = R(x)$*

Fact 6 *Given any point p in a justified hyperrectangle x, if $i > G(x)$, $S(p, i)$ is not in x.*

Fact 7 *Given any point p in a justified hyperrectangle x, if $i \le G(x)$, $S(p, i)$ is in x.*

3 Static point location in justified hyperrectangles

Theorem 8 *Given a fixed dimension d, a set of n justified non-intersecting hyperrectangles with nonnegative coordinates less than U may be maintained with the following properties: The construction of the structure takes expected time $O(n \log U \log\log U)$. The structure has $O(\log\log U)$ worst-case query time. The data structure takes space $O(n \log\log U)$ where n is the number of hyperrectangles stored in the structure.*

The data structure uses dynamic perfect hashing to maintain a set of encodings of points. Associated with each encoding stored in the hash table is a possibly empty pointer to a justified hyperrectangle. The data structure is based upon the on maintaining the following invariants:

Invariant 9 *For every justified hyperrectangle s in the structure, for every integer $i \in H(G(s))$, $\langle S(s, i) \rangle$ is in the hash table. No other encodings of points are in the hash table.*

Invariant 10 *For every p, if $\langle p \rangle$ is in the hash table, and p is in some justified hyperrectangle s, a link to s is stored in the hash table at location $\langle p \rangle$.*

Assuming these invariants hold, and given facts 5, 6, and 7, the following three facts may be concluded:

Fact 11 *For any point p in justified hyperrectangle s, $\langle S(p, G(s)) \rangle = \langle R(s) \rangle$ is in the hash table, and $S(p, G(s))$ has a link to s.*

Fact 12 *For any point p in justified hyperrectangle s, if $i > G(s)$, and $i \in H(G(s))$, $\langle S(p, i) \rangle$ is in the hash table and will not have a link to s.*

Fact 13 *For any point p in justified hyperrectangle s, if $i < G(s)$, if $\langle S(p, i) \rangle$ is in the hash table, it will have a link to S.*

	Dim-ension	Query	Space	Construction
Disjoint Rectangles [5]	d	$O(\log^{d-1} n)$	$O(n)$	$O(n)$
Polyhedra [9]	d	$O(\log n)$	$O(n^{d-1})$	$O(n^{d-1})$
Rectangular Subdivision [3]	2	$O((\log\log U)^2)$	$O(n)$	$O(n)\dagger$
	3	$O((\log\log U)^3)$	$O(n \log\log U)$	$O(n \log\log U)\dagger$
	3	$O(\log^{\epsilon} U), \epsilon > 0$	$O(n)$	$O(n)\dagger$
Fat Objects [2]	d	$O(\log n)$	$O(n)$	$O(n \log n)$
Fat Hyperrectangles (this paper)	d	$O(\log\log U)$	$O(fn \log\log U)$	$O(fn \log U \log\log U)$ †

Figure 1: Summary of results. n is the number of items in the data structure. U is the maximum integer coordinate. f is the average fatness of the stored hyperrectangles. †indicates an expected runtime.

The three facts above show that by binary searching on i with in the range $0 \ldots \lceil \log U \rceil$, examining the hash table for $S(p,i)$ at each step, it is possible to determine whether p is contained in any justified hyperrectangle, and if it is, which hyperrectangle it lies in. Since each query into the hash table takes $O(1)$ time, and the binary search requires $O(\log\log U)$ queries into the hash table, a point location query can be executed in $O(\log\log U)$ time.

Construction: Given a set S of n justified hyperrectangles with integer coordinates less than U, the construction algorithm proceeds as follows:

Create a dynamic perfect hash table H, with maximum allowed integer U^d.

For each justified hyperrectangle $x = [l_1, r_1] \times [l_2, r_2] \times \cdots [l_d, r_d]$ in S, do the following: Compute $R(x)$, which is the point $(l_1, l_2, \ldots l_d)$. Compute $G(x)$ which is the minimum number of trailing zeroes in the binary representation of any of the scalars that comprise $R(x)$. Calculating the number of trailing zeroes in the binary representation number is a bit operation which is implemented in hardware in modern CPUs (e.g. as the BSR operation on a Pentium Pro [7]) and thus we assume that it can be performed in constant time. Compute $H(R(x))$. For every $h \in H(G(x))$, insert $\langle S(R(x), h) \rangle$ into the hash table. When inserting $\langle R(x) \rangle$, which will be inserted since $R(x) \in H(G(x))$ and $S(R(x), G(x)) = R(x)$, link this entry in the hash table to x. All other entries in the hash table are added without links to any justified hyperrectangle. Note that when two multiple insertions are performed at identical locations in the hash table, only one will insert with a link. Since the size of $H(R(x))$ is $\lceil \log\log U \rceil$, the time to insert $n\lceil \log\log U \rceil$ integers into a dynamic perfect hash table is $O(n \log\log U)$ randomized. The time to compute $G(x)$ for every x is $O(n \log\log U)$ deterministic.

At this point the first invariant is satisfied. The next step is to add links to those encodings of points stored in the hash table that are inside a justified hyperrectangle. Since for every point p, inside a hyperrectangle x, there is an integer i such that $S(p,i) = R(x)$, and since $\langle R(x) \rangle$ is stored in the hash table with a link to x, one can easily determine is a point in a justified hyperrectangle in S. Thus the following is done for every point p without a link whose encoding is stored in the hash table: For every i from 1 to $\lceil \log U \rceil$, search the hash table for $\langle S(p,i) \rangle$. If $\langle S(p,i) \rangle$ is in the hash table with a link to a justified hyperrectangle x, determine if p is in x by comparing their coordinates (in $O(d)$ time, which is $O(1)$ for fixed d). If p is in x, add a link to x from $\langle p \rangle$ to x. This method takes $O(\log U)$ time for each of the $O(n \log\log U)$ points in the hash table, for a total of $O(n \log U \log\log U)$ construction time.

4 Dynamic point location in justified hyperrectangles

Theorem 14 *Given a fixed dimension d, a dynamic set of non-intersecting hyperrectangles with nonnegative coordinates less than U may be maintained with the following properties: Insertion and deletion of a hyperrectangle can be performed in $O(1)$ expected amortized time. Point location queries take $O(\log U)$ time. The data structure takes space $O(n)$ where n is the number of hyperrectangles stored in the structure.*

A data structure for point location in a dynamically changing set of hyperrectangles may be constructed by simply storing $\langle R(s) \rangle$ in the hash table for every justified hyperrectangle s in the data structure. To insert a justified hyperrectangle s, insert in link to s at location $\langle R(s) \rangle$ in the hash table in $O(1)$ expected amortized time. To delete a justified hyperrectangle s, delete the link to s from location $\langle R(s) \rangle$ in the hash table in $O(1)$ expected amortized time. Given a point p, to determine which justified hyperrectangle, if any, it lies in, query the has table at locations $\langle S(p,i) \rangle$ for $i = 0, 1, \ldots \lceil \log U \rceil$, until a link to a hyperrectangle is found. If p lies within this hyperrectangle, then the solution is found, otherwise it can be concluded that p lies outside of all hyperrectangles. Fact 5, Fact 7 and the fact that the hyperrectangles are disjoint guarantees that if p lies in a hyperrectangle, it will be the first one found using this search procedure. The query time is $O(\log n)$. As there is exactly one entry in the hash table for each hyperrectangle in the structure, the space used by this data structure is linear in the number of hyperrectangles currently in the structure.

5 Insertion only method

Theorem 15 *Given a fixed dimension d, a set of n justified non-intersecting hyperrectangles with positive integer coordinates less than U may be maintained, supporting the following operations: Insertion into the structure is performed expected amortized cost $O(\log U \log\log U)$, and point location in worst-case time $O(\log\log U)$. The space used by the structure is $O(n \log U)$.*

The potential method is used. The amortized time of operation i, $\hat{a}_i$, is defined to be the actual time of the operation, a_i, plus the change in potential, $\Phi_i - \Phi_{i-1}$. Summing over a sequence of operations yields: $\sum_{i=1}^{m} a_i = \sum_{i=1}^{m} \hat{a}_i + \Phi_0 - \Phi_m$. Thus, the actual runtime of a sequence of operations is equal to the sum of the amortized time of the operations plus the net loss of potential. If the potential is initially zero, and is always nonnegative, the sum of the amortized costs of the operations is an upper bound on the sum of the actual costs of the operations. For an full discussion of the potental method, and amortized analysis in general, see [11].

This data structure uses dynamic perfect hashing to maintain a set of encodings of points. The data structure is based on maintaining the following invariants:

Invariant 16

For every p, if $\langle p\rangle$ is in the hash table, and p is in some slab s, a link to s is stored in the hash table at location $\langle p\rangle$.

Invariant 17 *For every slab s, for every integer $i \geq G(s)$, $\langle S(s,i)\rangle$ is in the hash table. No other encodings of points are in the hash table.*

Invariant 18 *c units of potential are associated with every encoding of a point in the hash table that does not lie within any currently stored slab. Other encodings of points have no potential. The constant c is the time to execute the mark function described below, not including any recursive subcalls.*

Query: Point location queries are implemented by performing binary search on i in the range $0 \ldots \lceil \log U \rceil$ by examining $\langle S(q,i)\rangle$ in the hash table. If $\langle S(q,i)\rangle$ is in the hash table and is not linked to a slab that contains q, the binary search is continued using a smaller value of i. If $\langle S(q,i)\rangle$ is not in the hash table the binary search is continued using a larger value of i. If $\langle S(q,i)\rangle$ is linked to a slab that contains q then the search has ended successfully. This binary search performs at most $O(\log\log U)$ queries into the hash table at a worst-case cost of $O(1)$ each. Since query does not cause any potential changes, its runtime is worst-case $O(\log\log U)$.

Insert: Insert s is implemented by querying the structure using the method of the previous paragraph to see if any of the points in the set $S(q,i)$, where $i > G(s)$ lies in a slab. $\langle S(q,i)\rangle$ is then inserted in the hash table for all values of i greater than $G(s)$, providing a link to the slab the newly inserted point lies in, if such a slab was found in the previous step.

A link to the newly inserted slab must be made in any points previously in the hash table that lie within the newly inserted slab. $S^{-1}(p,g)$ is defined on any point p for which $S(p,g) = p$ to be the set of points $p + \{0, 2^{g-1}\}^d$. Note that $S(p, g-1) \in S^{-1}(S(p,g), g)$. Due to the Invariant 17 above, it is possible to backtrack from $R(s)$, using S^{-1} to find all points in the hash table inside of s, and link them to s. Calling the function `mark(R(s),s,G(s))`, illustrated below, achieves this objective.

```
procedure mark(point p, slab s, integer g)
{
  link p to s
  for every q in S^{-1}(p,g)
     if (<q> is in the hash table
           and q is in s
           and q != p)
        mark(q,s,g-1);
}
```

Let a denote the number of items in the hash table that had to be linked to s. The actual runtime of the Insert was $O(\log U \log\log U) + ac$. The $O(\log U \log\log U)$ term comes from querying the data structure in $O(\log\log U)$ time for each of the up to $O(\log U)$ items that were being inserted into the hash table. The total potential change is at most $c \log U - ac$, since every one of the up to $\log U$ newly inserted nodes could not be linked to a slab, and thus incur a potential charge of c, while the mark algorithm incurs a loss of c for every node linked. Thus the amortized cost is $O(\log U \log\log U)$. The gain in space is $O(\log U)$.

6 Point location in fat hyperrectangles

In this section, the justified hyperrectangle based point location data structures are generalized to *fat* axis-parallel hyperrectangles. The fatness of a hyperrectangle is defined as follows:

Definition 19 *Given a hyperrectangle* $h = [l_1, r_1] \times \cdots \times [l_d, r_d]$, *the* fatness factor *of* h *is defined to be* $F(h) = \prod_{i=1}^{d} \frac{r_i - l_i}{\min_j (r_j - l_j)}$

Intuitively, if $F(h)$ is close to 1, then the hyperrectangle is almost a hypercube, while if $F(h)$ is large, the hyperrectangle is long and thin for some dimensions. Fatness is a natural property for geometric objects, and a number of algorithms that use this property have been presented.

In order to perform point location amongst fat hyperrectangles, we will decompose each hyperrectangle into a set of justified hyperrectangles, which will be stored into one of the justified hyperrectangle based data structured described above. The fatness factor of a hyperrectangle will only have an multiplicative impact on the insertion, deletion, and construction costs, but not on the query costs.

Let $-x$ denote the positive integer formed by interpreting the $\lceil \log U \rceil$ bit x as a $\lceil \log U \rceil + 1$ bit number, and taking the one's compliment of this number, and interpreting this as a positive integer. For example, if U was 15, and x was 11, $-x$ would be 20 since the one's compliment of 01011 is 10100. Using this property allows one to map any interval into two justified one-dimensional hyperrectangles. For example, in the interval $[3, 11]$, 8 has the largest number of trailing zeros. When you look at the intervals $[-7, -3]$, and $[8, 11]$, both are justified hyperrectangles. Looking at $[-7, -3]$, this is actually $[11000, 11100]$, which is $[24, 28]$, and is a justified hyperrectangle. In general, in can be seen the if z has the greatest number of trailing zeroes in the interval $[x, y]$, $[-(z-1), -x]$ and $[z, y]$ are justified hyperrectangles in one dimension.

Generalizing this idea to n dimensions yields the following construction:

1. For each $i = 1, \ldots, d$, compute z_i, the maximum number of trailing zeroes in the binary representation of any number in $[l_i, r_i]$.

2. Let $G(h) = \min_i z_i$. Let i^* be such that $z_{i^*} = G(h)$.

3. For each $i = 1, \ldots, d$, let $M_i = \{m_1^i, m_2^i, \ldots\}$ be the set of all numbers in $[l_i, r_i]$ whose binary representation has $G(h)$ trailing zeroes. Let $k_i = |M_i|$.

4. The justified hyperrectangles decomposing h will be defined by:
$$\begin{aligned}&\{[-(m_1^1 - 1), -l_1], [m_1^1, m_2^1 - 1], \ldots\\ &\quad [m_{k_1-1}^1, m_{k_1}^1 - 1], [m_{k_1}, r_1]\} \times\\ &\{[-(m_1^2 - 1), -l_2], [m_1^2, m_2^2 - 1], \ldots\\ &\quad [m_{k_2-1}^2, m_{k_2}^2 - 1], [m_{k_2}, r_2]\} \times\\ &\cdots \times \cdots \times \cdots\\ &\{[-(m_1^d - 1), -l_d], [m_1^d, m_2^d - 1], \ldots\\ &\quad [m_{k_d-1}^d, m_{k_d}^d - 1], [m_{k_d}, r_d]\}\end{aligned}$$

Thus, the number of justified hyperrectangles covering h is $\prod_{i=1}^{d}(k_i + 1)$. This product may be bounded by first observing that $2^{G(h)+1} > r_{i^*} - l_{i^*}$. Otherwise, $m_1^{i^*} + 2^{G(h)}$ or $m_1^{i^*} - 2^{G(h)}$ would be in $[l_{i^*}, r_{i^*}]$, and have more than $G(h)$ zeroes. Then:

$$k_i \leq \frac{r_i - l_i}{2^{G(h)}} + 1 < \frac{2(r_i - l_i)}{r_{i^*} - l_{i^*}} + 1 \leq \frac{2(r_i - l_i)}{\min_j (r_j - l_j)} + 1$$

And:

$$\prod_{i=1}^{d}(k_i + 1) \leq \prod_{i=1}^{d}\left(\frac{2(r_i - l_i)}{\min_j (r_j - l_j)} + 2\right) \leq 4^d F(h)$$

7 Summary

Given a set S of n hyperrectangles with integer coordinates less than U, the method of section 6 is used to transform them into a set S' of $O(\sum_{h \in S} F(h))$ justified hyperrectangles with coordinates less than $2^{\lceil \log U \rceil + 1} \leq 4U$. This set S' is given to a justified hyperrectangle based structure for insertion, deletion or construction. Let $T(p)$ denote the set of 2^d points obtained from p by taking the one's compliment of all possible subsets of the d scalars that define p. Point location on p in the set S is carried out by performing point location on all of the elements of the set $T(p)$ in the justified hyperrectangle based data structure storing S'. The justified hyperrectangle construction insures that if p is in hyperrectangle x that was decomposed into a set of justified hyperrectangles X', exactly one point location query on elements of $T(p)$ will succeed, returning one element of X'. Thus working with justified hyperrectangles instead of arbitrary hyperrectangles incurs a multiplicative degradation of the query time by a factor of of $O(2^d)$, which is $O(1)$ for any fixed d.

References

[1] B. Chazelle. Triangulation a simple polygon in linear time. *Discrete Computational Geometry*, 6:485–524, 1991.

[2] M. de Berg. Linear size binary space partitions for uncluttered spaces. Technical Report UU-CS-1998-12, Universiteit Utrecht, 1998. To appear in Algorithmica.

[3] M. de Berg, M. van Kreveld, and J. Snoeyink. Two- and three-dimesional point location in rectangular subdivistions. *Journal of Algorithms*, 18(2):256–277, 1995.

[4] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. M. auf der Heide, H. Rohnert, and R. E. Tarjan. Dynamic perfect hashing: upper and lower bounds. *SIAM J. Comput.*, 23(4):738–761, 1994.

[5] H. Edelsbrunner, G. Haring, and D. Hilbert. Rectangular point location in d dimensions with applications. *Comput. J.*, 29:76–82, 1986.

[6] M. L. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with o(1) worst case access time. *JACM*, 31(3):538–544, 1984.

[7] Intel Coperation. Intel architecture software developer's manual. volume 2: Insruction set reference, 1997.

[8] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12:28–35, 1983.

[9] M. Pellegrini. On point location and motion planning among simplices. *SIAM J. Comput.*, 25(5):1061–1081, 1996.

[10] M. Smid. Dynamic rectangular point location, with an application to the closest pair problem. *Inform. and Comput.*, 117(1):1–9, 1995.

[11] R. E. Tarjan. Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods*, 6(2):306–318, 1985.

[12] P. van Emde Boas. Preserving order in a forest in less than logarithmic time and linear space. *Inform. Process. Lett.*, 6:80–82, 1977.

[13] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\theta(n)$. *Inform. Process. Lett.*, 17(2):81–84, 1983.