2005 WSEAS Int. Conf. on REMOTE SENSING, Venice, Italy, November 2-4, 2005 (pp121-125)

2-D Monotone spatial indexing scheme with optimal update time

L. DROSSOS[1], S. SIOUTAS[2], K. TSICHLAS[2] and K. IOANNOU[3]

[1] Technological Institute of Messolongi,
Department of Applied Informatics in Administration and Economics
Technological Institute Campus, 30200, Messolongi
GREECE

[2] Computer Engineering and Informatics department
University of Patras
Building B, University Campus, 26500, Rion, Patras
GREECE

[3] Wireless Telecommunication Laboratory, Department of Electrical and Computer Engineering,
University of Patras,
Rion 26500, Patras
GREECE

Abstract - For monotone generated points on the plane we present the Dynamic Monotone Priority Search Tree (DMoPST) in main / external memory with O(1) update time / O(1) block transfers in worst-case. The external version of the structure above promises efficient applications in transaction time Databases systems.

Key-Words: -Databases, Data structures

1 Overview

For monotone or periodical monotone generation of points in the plane we present a new dynamic version of Priority Search Tree, the Dynamic Monotone Priority Search Tree (DMoPST) in main / external memory with O(1) time / O(1) block transfers in worst-case for update operations. Even if the dynamic generation of points constitutes a special case it is the first time a dynamic version of priority search tree is being presented with optimal update time. The best previous results in main / external memory were presented in [10] / [3] and require O(logn/loglogn) time / O($\log_B n$) I/O's respectively for all operations (three-sided lookup queries, insertions, deletions).

1.1 The Classic Priority Search Tree

We will briefly review the priority search tree of McCreight [6]. Let *S* be a set of *n* points on the plane. We want to store them in a data structure so that the points that lie inside a half infinite strip of the form $[-\infty, b] \times (-\infty, c]$ can be found efficiently.

The priority search tree is a binary search tree of depth *O(logn)* for the *x*-coordinates of the points. The root of the tree contains the point *p* with smallest *y*-coordinate. The left (resp. right) subtree is recursively defined for the set of points in *S-{p}* with *x*-coordinates in the left (resp. right) subtree. From this definition it is easily seen that a point is stored in a node on the search path from the root to the leaf containing its *x*-coordinate.

To answer a range query with a quadrant $[-\infty, b] \times [-\infty, c]$ we find the *O(logn)* nodes in the search path P_b for *b*. Let L_b be the left children of these nodes, that don't lie on the path. For the points of the nodes of P_b *U* L_b we can determine in *O(logn)* time which lie in the query-range. Then for each node of L_b whose point is inside the range we visit its two children and check whether their points lie in the range. We continue recursively as long as we find points in the query-range.

For the correctness of the query algorithm first observe that nodes on the right of the search path have points with *x*-coordinate larger than *b* and therefore lie outside the query-range. The points of P_b may have *x*-coordinate larger than *b* or they may have *y*-coordinate larger than *c*. If any of these is true then they are not reported. The nodes of L_b and their descendants have points with *x*-coordinate smaller than *b* so only their *y*-coordinates need to be tested. For the nodes of L_b whose points lie inside the query-range we need to look further at their descendants. The search proceeds as long as we find points inside the query-range. If a point of a node *u* does not lie inside the query-range then this point has *y*-coordinate larger than *c*. Therefore all points in the subtree rooted at *u* lie outside the query-range and need not to be searched. From the above discussion we can easily bound the query time by *O(logn+k)*, since we need *O(logn)* time to visit the nodes in P_b *U* L_b and *O(k)* time for searching in their subtrees.

1.2 The Fusion Priority Search Tree

The main advantage of the fusion technique ([10]) is that we can decide in time O(1) in which subtree to continue the searching by compressing the k-keys of every B-tree node using w - bit words.

Willard used as a skeleton structure a B-tree whose internal nodes have arity between B/8 and B, with the exception of the root that has arity between 2 and B, and whose leaves store the data and all have the same depth. Each node v stores information about the y_max value of the set of points stored in T_v, it also stores compressed information into a q-heap ([10]) about the y_max values of the respective sons of v, in order to decide in O(1) time to which of them he must continue a further searching.

He also proved that, it is possible to devise an insertion and deletion algorithm for this tree that runs in worst-case time O(h), where h=O(logn/loglogn). Quadrant and Three-sided Queries can be answered in O(h+k) worst-case time where k the size of the answer.

2 The Main Memory DMoPST data structure

We consider the special case of monotone or periodically monotone generated points in the plane.

Definition 1: A dynamic set of n points $S = \{(x_i, y_i), 1 \le i \le n\}$, is monotone iff $\forall i, j : i \le j$, $x_j > x_i$ and $y_j > y_i$ (increasing monotone set) or $y_j < y_i$ (decreasing monotone set).

Definition 2: A set of n points $S = \{(x_i, y_i), 1 \le i \le n\}$, is periodically and uniformly monotone iff we can divide the plane into space slabs of approximately equal size each of which is monotone according to definition 1.

In order to achieve for the above dynamic sets a special version of a priority search tree with linear space and O(1) update time we use the bucketing technique. The essence of the bucketing method is to get the best features of the two different structures by combining them into a two-level structure. The data to be stored is partitioned into buckets B_i, $1 \le i \le k$, (see Fig. 1.a) and the chosen data structure for the representation of each individual bucket is different from the representation of the top-level data structure, representing the O(n/k) collection of buckets (for similar applications of this data structuring paradigm see also [7,5,9]). The points of the Fig. 1.b are generated in a periodical monotone way. Let t_i, $1 \le i \le k$, the ith time period and S_i the corresponding space slab the points of which are going to be stored in the 2-level data structure T_i. During the time the only updateable data structure is the last one (see the T_j 2-level structure of Fig. 1.b). For every t_i, $i \ge 1$, we execute a global reconstruction of a new static priority search tree **STi** ([2,11]) for the points generated between the time periods t_{i-1} and t_i. Considering the same size of each slab due to uniform distribution we can spread out the linear work spent during a global reconstruction on the next

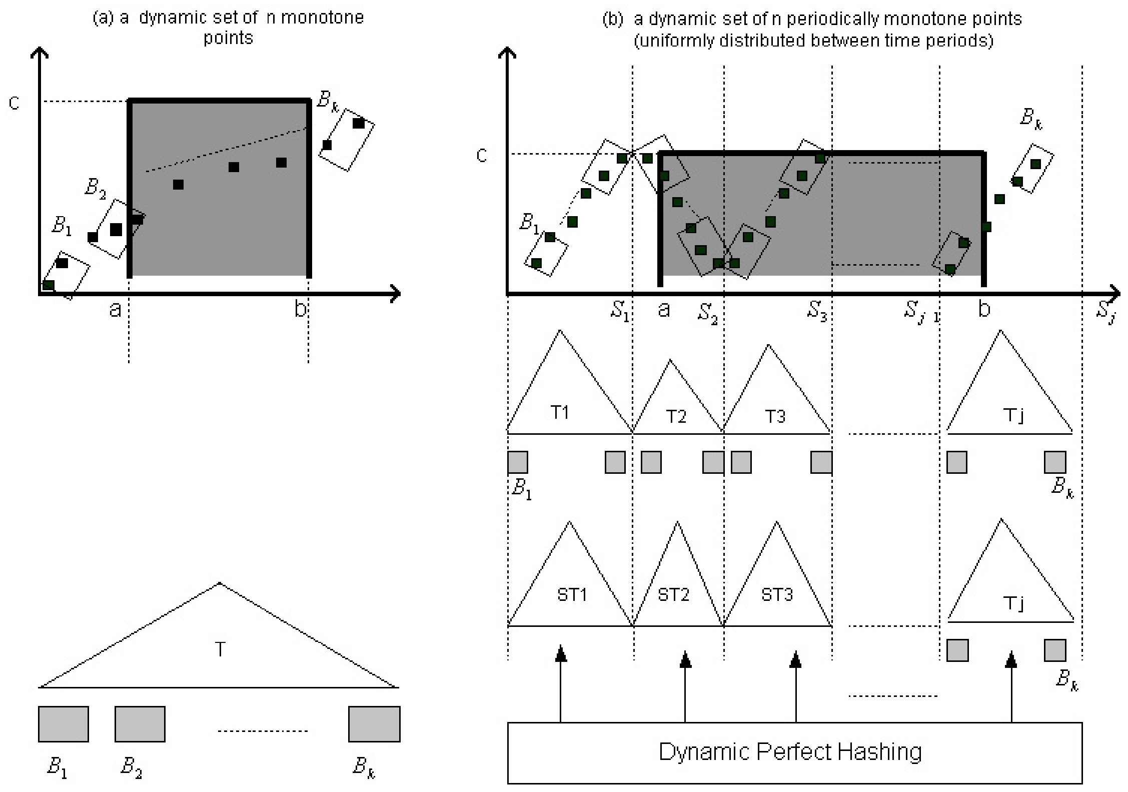
(a) a dynamic set of n monotone points
c
B_1
B_2
B_k
a
b
T
B_1
B_2
B_k
(b) a dynamic set of n periodically monotone points (uniformly distributed between time periods)
c
B_1
B_k
S_1
a
S_2
S_3
S_{j-1}
b
S_j
T1
T2
T3
Tj
B_1
B_k
ST1
ST2
ST3
Tj
B_k
Dynamic Perfect Hashing

Fig 1: Monotone and periodically monotone trajectories of points

updates between t_i and t_{i+1}. The O(1) worst case update cost for the global reconstruction follows.

In the Figure 1 below it is depicted (a) a dynamic set of n monotone generated points and the corresponding 2-level data structure (b) a dynamic set of n periodically monotone generated points, uniformly distributed between time periods, and the corresponding data structures. In each time period t_i the respective 2-level structure T_i is replaced by the static structure ST_{i-1}. In this way it's obvious that the only dynamic or updateable data structure is the last one. Due to the uniform distribution among time periods we can also use a dynamic perfect hash table in order to lookup in O(1) time the respective set of points and as a consequence the corresponding data structure for further searching.

Particularly we use the dynamic perfect hashing strategy of [1] : It's about a randomized algorithm for the dynamic dictionary problem that takes O(1) worst-case time for finding the roots of the corresponding (static or dynamic) data structures (see Fig.1.b) and O(1) amortized expected time for insertions and deletions. It also uses space proportional to the size of the set stored.

We study the two-follow cases:

a) The top-level structure is the classic priority search tree ([6])

a.1) Update time: We will show that making the buckets B_i, $1 \le i \le k$, have size O(log*n*), (k=O(n/logn)) and using as the top-level data structure the Priority Search tree of [4], yields a simpler algorithm.

In addition to the time required to split/fuse buckets, a bucket-rebalancing step may require O(logn) worst-case time to insert/delete a bucket reprentative to/from the top-level tree. The top-level tree is the Priority Search Tree that requires O(logn) worst - case update time. Since the total work to rebalance a bucket is O(logn), we can perform it with O(1) work per update spread over the next O(logn) updates. In other words, if we can permit every bucket to be of size $\Theta(\log \hat{n})$, where $\hat{n}$ the number of current elements, we can guarantee that between rebalancing operation performed in the top-level tree there is no possibility for any other such operation to occur and consequently the incremental spread of work is possible.

In the case of Fig. 9.b we must spent an extra update time overhead for updating the Dynamic Perfect Hash Table, that is a O(1) expected amortized time cost.

a.2) Three-sided Lookup Queries:

a.2.1) Monotone Generated points (Fig. 1.a)

Let the $[a,b]x(-\infty,c]$ query of Fig.1. The required time is $O(\log(n/\log n)+k_1)=O(\log n+k_1)$ for the binary-searching of k_1 bucket's representatives (these representatives belong to the answer) in the top-level Priority Search Tree ([4]) and $O(k_2)$ for the sequential searching in the responding buckets until we reach points with y>c (remember that we gain the benefit of y and x ordering simultaneously). So, the total required time is O(logn+k).

a.2.2) Periodically Monotone Generated points (Fig. 1.b)

The query splits into the follow subqueries (see Fig.1.b): $[a,b]x(-\infty,c]= [a,S_2]x(-\infty,c]+\ldots+ [a,S_{j-1}]x(-\infty,c] + [S_j ,b]x(-\infty,c]$. All except the last one are queries in the appropriate consecutive $ST_2,\ldots,ST_{j-1}$ static structures. The last one is a query in the dynamic T_j structure. So, we pay an extra O(1) worst-case lookup cost in the Dynamic Perfect Hashing Table in order to find the appropriate static structure and $O(k_i)$ cost (k_i<<k and $2\leq i\leq j-1$) for each of the $ST_2,\ldots,ST_{j-1}$ static structures respectively in order to find totally $\sum_{i=2}^{j-1} k_i$ points that satisfy the query. It remains to query the last dynamic structure. Therefore the last subquery requires $O(\log C+k_j)$ time, where C the size of each slab due to the uniform distribution and $k_j + \sum_{i=2}^{j-1} k_i$ =k the total size of the answer. So, in this case the total required time is O(logC+k).

b) The top-level structure is the Fusion priority search tree ([10])

b.1) Update time: Now we make buckets of size O(logn/loglogn) and the O(1) update time follows. In the case of Fig. 9.b we must spent an extra O(1) expected amortized time cost.

b.2) Three-sided Query:

b.2.1) Monotone Generated points (Fig.1.a)

Let the $[a,b]x(-\infty,c]$query of the scheme above The required O(logn/loglogn+k) time follows while we process the query in the same manner as in **a.2.1** case.

b.2.2) Periodically Monotone Generated points (Fig.1.b)

The required O(logC/loglogC+k) time follows while we process the query in the same manner as in **a.2.2** case.

We retain the buckets in the appropriate size using the global rebuilding ([6]) technique. We will use two structures: OLD-MAIN and MAIN. Normally only MAIN exists. When the number of transactions on MAIN exceeds half its initial size, MAIN is made into OLD-MAIN and a construction is initiated to build a new (i.e., rebalanced) MAIN from it. Meanwhile insertions, deletions and queries continue on OLD-MAIN, until the new MAIN under construction can take over. Assuming there were n_0 points when the construction of a new MAIN begun, it was shown that the new structure can take over after at most $(1/3)n_0$ transactions. In the sequel, a transaction will always be an insertion or a deletion.

The data structure described above is complicated in the sense that it uses bucketing combined with the global rebuilding method. We can avoid the global rebuilding using the algorithmic techniques of pebble games ([7]).

Remark: In the case of Monotone Generated points, the top-level structure is a min_priority search tree that means the representative is always the first element (point) of each bucket and this one with the minimum y_coordinate is stored in the root.

In the case of Periodically Monotone Generated points, we store the points of increasing monotone periods into a number of min_priority search trees and the points of decreasing monotone periods into a number of max_priority search trees. The last one structure stores the representatives which are always the last elements (points) of its buckets and the representative with the maximum y_coordinate is always stored in the root.

3. The DMoPST external data structure: A first look.

A first look proposes again a 2-level data structure. That means the data to be stored is partitioned into buckets of $O(\log_B N)$ points and the choosen as a top-level data structure for representating each individual bucket is the external priority search tree of [3]. Since the total work to rebalance a bucket after an insertion takes O($\log_B N$) I/Os, we can perform it with O(1) work per update spread over the next O($\log_B N$) updates. A second more deeply look discovers some dificulties. In [3] the key idea is used to obtain O($\log_B N$) I/Os in worst case when an insertion occurred is to complete the rebuilding phase performing a spliting on each node of the search path using O(1) I/Os. The split can be finalized using O(1) I/Os by constructing incrementally the Y-sets Y(u') and Y(u'') and the query structures $Q_{u'}$ and $Q_{u''}$ of the two new nodes u' and u'' that will be created by the split of node u. That means, we must determine the points to promote for Y(u') and Y(u'') before forming u' and u'', so, the data structure will not be a valid external priority search tree and queries will not be performed in the optimal number of I/Os.

4. Conclusion

In this work we have focused on special cases of 2-D range searching constructing for the first time a new dynamic priority search tree with optimal update time. Our next step for future continuation of this work is the externalization of the DMoPST structure which can promise more efficient applications in transaction time Databases systems ([8]).

Acknowledgements

The authors would like to thank the Operational Program for Educational and Vocational Training II (EPEAEK II) and particularly the Program PYTHAGORAS, for funding the above work.

REFERENCES

[1] Martin Dietzfelbinger, Anna Karlin, Kurt Mehlhorn, Friedhelm Meyer Auf Der Heide, Hans Rohnert, and Robert E. Tarjan, Dynamic Perfect Hashing: Upper and Lower Bounds., SIAM J. Comput. Volume 23, Number 4 pp. 738-761

[2] H. N. Gabow, J. L. Bentley, and R. E. Tarjan, Scaling and related techniques for geometry problems, *in Proceedings, l6th Annual ACM Symp. on Theory of Computing, 1984, pp. 135-143.*

[3] L.Arge, V.Samoladas, J.S.Vitter, "On two-Dimensional Indexability and Optimal Range Indexing", *ACM-PODS '99.*

[4] E. M. McCreight, Priority search trees, *SIAM J. Comput. 14 (1985), pp. 257-276.*

[5] M. Overmars, A O(1) average time update scheme for balanced binary search trees, *Bulletin of the EATCS,* 18:27-29, 1982.

[6] M. Overmars and Jan van Leeuwen, Worst case optimal insertion and deletion methods for decomposable searching problems, *Information Processing Letters*, 12:168-173, 1981.

[7] Raman, R. Eliminating Amortization: On Data Structures with Guaranteed Response Time. *PhD Thesis, University of Rochester, New York, 1992. Computer Science Dept. U. Rochester, Technical Report* TR-439.

[8] B. Salzberg and V. Tsotras, Comparison of Access Methods for Time-Evolving Data, ACM Computing Surveys, Vol. 31, No. 2, June 1999.

[9] Tsakalidis, A. Maintaining order in a generalized linked list, *ACTA Informatica* 21 (1984)

[10] Willard, D., "Applications of the Fusion Tree Method to Computational Geometry and Searching", *ACM-SIAM Symposium on Discrete Algorithms*, 1992.

[11] N. Kitsios, Ch. Makris, S. Sioutas, J. Tsaknakis and B. Vassiliadis, **Geometric Retrieval for Grid Points in RAM model**, Journal of Universal Computer Science (**J.UCS)**, Springer, (to appear).