A report on

An Optimal Dynamic Interval Stabbing-Max Data Structure

For the partial fulfillment of the course

CS F211 - Data Structures & Algorithms

Under the guidance of

Prof. Sundar S Balasubramaniam

Submitted by:

Rohit Lodha - 2015A7PS040P

Vivek Singhvi - 2015A7PS108P

Team No : 51

Abstract:

The paper on "**An Optimal Dynamic Interval Stabbing-Max Data Structure**" by Pankaj K. Agarwal, Lars Arge and Ke Yi considers the optimal structure for answering queries on **Dynamic *Interval Stabbing-Max Problem***. This paper deals with the one-dimensional version of the problem,i.e, to find an interval with maximum weight containing a given query point. The paper draws its motivation from the algorithmic improvements developed by Kaplan [2] which involves the use of *self-balancing "binary" tree* with secondary data structures. This article improves the deletion speed attained by Kaplan's method by changing the secondary data structures. The data structure is easily compatible and can be adapted for use with external memory with dynamic environment. It can also be extended to higher dimensions. This problem has widespread applications in many areas, including networking and databases. So improving the data structure is a major breakthrough in Networking field.
The main idea of the paper revolves around increasing the fan out of the base tree to reduce its height and using secondary data structures to retrieve and update data quickly and efficiently.

Expository Summary:

The Stabbing-max problem (also known as *rectangle intersection with priorities problem*) is the problem of maintaining a dynamic set of n axis-parallel hyper-rectangles having a weight associated with them such that a rectangle with the maximum weight containing the query point can be found efficiently.
Let there be a set S consisting of n intervals. A *stabbing query* has a point q and asks for an interval with the maximum weight that contains q. The *Interval Stabbing-Max Problem* is to find a data structure that can handle stabbing-max queries efficiently.
The data structure developed in the article updates (insert and delete) and answers the queries in worst case O(log n) time.

Assumptions :

1. The endpoints of all intervals that are in the data structure belong to a fixed set of points.
2. No two intervals have the same endpoint.
3. The range of an interval is its weight.

Prior Work on Stabbing-Max Queries

Trivial Case without deletions

A simple binary search tree taking O(*n*) space and O(log *n*) time for query and insertion.

Interval Tree:

A binary tree sorted on left endpoint of the interval with nodes associated with maximum right endpoint value of the subtree rooted at that node.

Kaplan Structure

BASE TREE (T):

- Based on the interval tree, a self-balancing Binary tree, consisting of O(n) left endpoints of the intervals, with the intervals stored in secondary structures of the nodes of T.
- Each node v is associated with a range σ_v :
  - for a leaf node it is two consecutive endpoints
  - for an internal node it is the union of σ_v of its children.
- Each internal nodes is also associated with a point x_v which is a common boundary point of σ_w and σ_z where w and z are its children.
- An input interval is associated with the highest node v such that $x_v \in s$.
- The subset S_v <= S associated with v is stored in two secondary structure
  - A dynamic height balanced tree sorted by left endpoints of intervals in S_v.
  - A dynamic height balanced tree sorted by right endpoints of intervals in S_v
  - At each of these secondary tree structure, the maximum interval in the subtree rooted at that node is stored.

Dynamic Fractional Cascading:

- The query time is improved by using dynamic fractional cascading.
- By replacing the secondary structure with structures that can answer a max-stabbing query in O(1) time, leading to an O(log n) query bound.
- Let Lv be the set of intervals in US_u, where u is an ancestor of v, whose left endpoints lie in the range associated with the left child of v.
- Similarly, R_v is defined w.r.t. right endpoints.

This improves the query time to O(log n) time as the maximum interval in the secondary structure can be found in O(1) time. But deletion now requires O(log n. log log n) time since an interval O(log log n) time is required to update each of the O(log n) secondary structures on a search path in T.

Proposed Structure

The basic idea is to increase the fan-out of the base tree T to $\log^c n$, reducing its height to O(log n/ log log n).

Data Structures:

Base Tree(T) - N ary trees	S_v - Doubly linked list	Mv Structure- Two Max heaps
Lv Structure – Tournament Tree		

Notations :

v : a node	u : ancestor of v	z and w : children of v	p(v) : parent of v
S : set of all intervals		s : one of the interval $\in$ S	T : Base Tree

Base Tree(T):

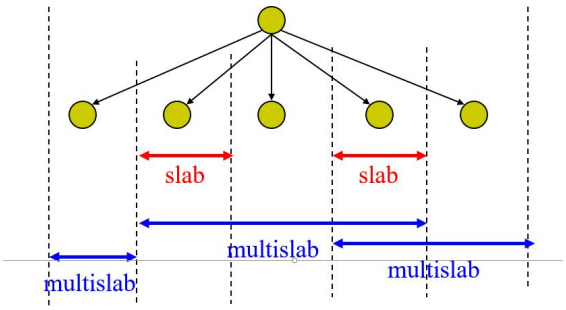
- The tree now consists of n/log n leaves with fanout f = ($\sqrt{\log n}$) with each leaf containing log n consecutive interval endpoints.
- Each node v is associated with a range σ_v :
  - for a leaf node it is two consecutive endpoints
  - for an internal node it is the union of σ_v of its children.
- The range σ_v is divided into f sub-ranges of its children called *slabs*.
  - Boundaries of these subranges are called *slab boundaries.*
  - A continuous range of these slabs is called *multislab*.
- An interval s is associated with v if s crosses at least one slab boundary of v but none of the boundaries at the parent of v.
- If an interval has both endpoints in a leaf z, i.e., it does not cross any slab boundaries, it is stored at z.
- The subset $S_v \subseteq S$ associated with v is stored in a **doubly linked list** at v. Pointers are kept between an interval in S_v and the leaves containing its two endpoints.

Secondary structure:

- Divide each interval in S_v into left(S^l_v), right(S^r_v) and middle(S^m_v) intervals.
  - (S^l_v) will consists of the intervals between the left endpoint and the first boundary.
  - (S^r_v) will contain intervals between the last boundary and the right endpoint.
  - (S^m_v) will be between the first and last boundary.
- Each internal node of T is associated with S^l, S^r and S^m.
  - S^l be the sets of all left (S^l_v) intervals.
  - S^r be the sets of all right (S^r_v) intervals.
  - S^m be the sets of all middle (S^m_v) intervals.
- The subsets of S^l, S^r and S^m are used to build the secondary structure associated with v.

Secondary structure for the intervals associated with the *leaves of T* are not needed as the stabbing-max query or update can be answered by simply scanning the O(log n) intervals in S_z. So these intervals are not considered in the following data structures.

Middle Interval - M_v Structure:



It is made up of two types of max-heaps.

1. Multislab Max-Heap: One Max-heap for each of the multislabs.
   - The multislab max-heap for multislab $\sigma_v[i:j]$ contains all interval in S^m_v that exactly spans $\sigma_v[i:j]$.
   - All multislab heaps use linear space as each interval in S^m_v is stored in exactly one multislab max-heap.
2. Slab Max-Heap: One Max-heap for each of the slabs.

- The slab max-heap for slab σ_{vl} contains the maximum interval from each of the O(log n) multislab max-heaps corresponding to multislabs that span σ_{vl}.
- Each slab max-heap uses $O(\log n)$ space for a total of $O(\log^{3/2} n)$ space.

Overall M_v uses $O(|S_v^m| + \log^{3/2} n)$ space such that a stabbing-max query can be answered in O(log log n) time.The M_v structure can be constructed in $O(|S_v^m| + \log^{3/2} n)$ time and updated in O(log n) time. The overall size of all the M_v structures is $O(n)$ since the number of internal nodes in T is $O(n/\log^{3/2} n)$. Thus, the set S^m can be stored in secondary structures of T so that query and updates can be performed in O(log n) time.

Left Interval - L_v structure:

Similar to Kaplan's structure, it stores some of the intervals that belong to US^l_u, where u are the ancestors of v.

Static version : The following formulae are used

- $\Psi(u,v) = \{s^l \mid s \in S_u$ and s's left endpoint is in $\sigma_v\}$, *i.e,* Ψ(u,v) is the set of intervals associated with u with left endpoint in the slab associated with v.
- $\psi(u,v) = \max \Psi(u,v)$
- let $p^{(0)}(v) = v$ and define $p^{(k)}(v) = p(p^{(k-1)}(v))$ for $k \geq 1$
- Let $\Phi(v) = \text{Uk}\geq 2\ \Psi(p^{(k)}(v),v)$ *,i.e,* $\Phi(v)$ is the set of intervals that belong to a proper ancestor of the parent of v and that have their left endpoints inside σ_v.
- ϕ(v) = max Φ(v).

This L_v structure is used to find the maximum interval in $\Phi(v_1) \cup \cdots \cup \Phi(v_i)$, for any $1 \leq i \leq f$.
Since one of the intervals can be found in many Ψ(v) and $\Phi(v)$, we will store ϕ(v) since the no. of such intervals in equal to the no. of nodes O(n) in T. A tournament tree(L_v) is formed on $\phi(v_1),\ldots, \phi(v_f)$, that is, a binary tree with f leaves whose ith leftmost leaf stores $\phi(v_i)$. Each internal node of L_v stores the maximum of the intervals stored in the leaves of the subtree rooted at v.
Using linear space O(n), the set S^l of left intervals can be stored in secondary structures of T so that stabbing-max queries can be answered in O(log n) time.

Dynamic Version: To make L_v dynamic, we need to maintain the $\phi(v)$ as well as $\psi(u,v)$ intervals during insertions and deletions of left intervals. Two additional linear-size data structures, K_u and H_v, at the nodes of T are created for efficient updating of data:

- for an internal node u, K_u stores the $\psi(u,v)$ intervals for all descendants v of u. It has the same structure as the subtree T_u of T rooted at u. Thus there is a bijection between the nodes of K_u and T_u.
  - The root of K_u stores $\psi(u,u)$ and $\psi(u,v_i)$ is stored in the stored in a child of the node storing $\psi(u,v)$. Hence, it is tournament tree with fanout f.
  - At leaf z, we store a small tournament tree on intervals of sorted $\Psi(u,z)$, in increasing order of left endpoint that lies in σ_z. This tree can be used to maintain the maximum interval of $\psi(u,z)$.

- for an internal or a leaf node $v \in T$, H_v stores the $\psi(u,v)$ intervals for all ancestors u of $p(v)$.
    - It is simply a max-heap on $\psi(u,v)$ for all (proper) ancestors u of the parent of v.

Using linear space O(n), the set S^l of left intervals can be stored in secondary structures of T so that updates and stabbing-max queries can be performed in O(log n) time.

General 1D Structure:

Removing the restriction of fixed endpoints, we will update the base tree T before insertion and after deletion of an interval from the secondary structure.

Deletion: This is done using global rebuilding. When an endpoint is deleted, we mark it as deleted in the respective leaf of T. After n/2 deletions, the structure is completely rebuilt without the deleted endpoints and transfer intervals from old to the new secondary structure. Thus the amortized cost of deletion is O(log n).

Insertion: After an insertion of an endpoint in a leaf z of the weight-balanced tree T, the weight constraint of the nodes on the path from z to the root of T may be violated. To rebalance the tree, we split v along a slab boundary b into two nodes v^0 and v^{00} of roughly equal weight and new slab boundaries are added accordingly. The affected S_v doubly linked list are also updated. Intervals that do not intersect b are moved to $S_{p(v)}$ and rest are distributed in S_v^0 or S_v^{00}. The addition of new boundary result in new M_v, L_v, H_u,and K_u structures.

The amortized cost of a split is $O(\log \log n)$. Since one insertion increases the weights of $O(\log n / \log \log n)$ nodes on a path of T, the amortized cost of an insertion is $O(\log n)$.

	Trivial without deletion	Interval Tree	Kaplan Structure	Kaplan Structure with dynamic fractional cascading	Efficient Structure
Insertion	O(log n)	O(log n)	O(log n)	O(log n)	O(log n)
Deletion	-	O(log n)	O(log n)	O(log n . log log n)	O(log n)
Query	O(log n)	$O(\log^2 n)$	O(log n . log log n)	O(log n)	O(log n)
Space	O(n)	O(n)	O(n)	O(n)	O(n)

Extensions

External memory:

This internal memory interval max-stabbing structure can easily be adapted to external memory by choosing the right base tree parameters. By changing the no. of endpoints containing in leaf of base tree and fan-out of all the tournament trees, this theory can be extended to external memory data structure for storing intervals of size $O(n/B)$ *so that stabbing-max queries can be answered in* $O(\log_B n)$ *I/Os worst-case and updates can be performed in* $O(\log_B n)$ *I/Os amortized.*

Higher Dimensions: This structure can be extended to higher dimension using segment trees adding an extra O(log n) factor to space, update and query time for each dimension.
Thus, there exists an $O(n.\log^{d-1}n)$ size data structure for storing a set of n axis-parallel hyper rectangles in R^d so that stabbing-max queries can be answered in $O(\log^d n)$ time worst-case and updates can be performed in $O(\log^d n)$ time amortized.
Similarly, the external structure can also be extended to higher dimensions.
Thus, *there exists an space external data structure for storing a set of n axis-parallel hyper-rectangles in* R^d *so that stabbing-max queries can be answered in* $O(\log^d_B n)$ *I/Os worst-case and updates can be performed in* $O(\log^d_B n)$ *I/Os amortized.*

Critique:

The paper on *Interval Stabbing-Max Problem for multi-dimensional data structure* takes a comprehensive look at the various aspect of the topic. The data structure is much more efficient than what had been achieved before. The paper discusses previous results as well as provides a new outlook to the problem. It also gives us a dynamic platform to compare all the results. The data structure has been presented in a coherent and logical format by stating the limitations of the previous works as well as scope of improvements. Some of them are increasing the fanout of nodes or using additional secondary data structures for storing the necessary information. Lemmas and Theorems are used wherever necessary to highlight the important functions of the data structure in query optimisation. Extension of structure to higher dimension has been presented by giving an analogy of 1-D case discussed in the article. The effect of space and time complexity on increasing the dimensions and using external memory has been discussed in brief.

The space complexity of the given method is mentioned to be of linear size but the constant factor is quite high due to the presence of many secondary structures**.** The principle of locality of references has been neglected in answering the queries. Many secondary data structures have been switched, which degrades the performance of the system. The presence of many data structure would be even more severe if they are stored in external memory and are large enough to be not stored in main memory resulting in high I/O disk access creating large overheads. Also, the research article focuses only on the theoretical aspects of the problem but fails to cues from the real world. It doesn't describe how the method would be used in networking and database systems. The research paper has been deprecated by Yakov Nekrich et al. [3] who has developed data structure to do the same task in a faster time bound and complexity.

REFERENCES:

[1] P. K. Agarwal, L. Arge, J. Yang, and K. Yi, An Optimal Dynamic Interval Stabbing-Max Data Structure
[2] H. Kaplan, E. Molad, and R. E. Tarjan, Dynamic rectangular intersection with priorities
[3] Yakov Nekrich, A Dynamic Stabbing-Max Data Structure with Sub-Logarithmic Query Time