CSC 6210 Advanced Multidimensional Search

Jintian DENG

December 12, 2009

1 Problem 1

Problem definition:

Build an external priority search tree[Arg04b] on a set of N points in $O(\frac{N}{B}\log_B N)$I/Os, assuming that the memory size M at least B^2.

Solution:

This problem can be solved easily using distribution sweep[GTVV93]. We use an optimal external sorting, e.g. distribution sort or merge sort, to sort all points into two lists, one sorted by x-axis and the other by y-axis. The list sorted by x is used to locate medians which we will use to distribute the points evenly into B vertical slabs S_i. The list sorted by y is used to perform the sweep from top to bottom.

Every time a point is encountered, we insert it into a list A_i associated with the slab S_i in which the point lies. After the sweep, each A_i is ordered along the y-axis since we take a top-down sweep. So we can easily take the B highest points in each slabs S_i just using $O(1)$I/Os without any sorting. After that we build a B^2-structure[Arg04a] over these B^2 points and zoom in each slabs S_i, using the ordered list A_i and do the procedure recursively down. We stop our recursion when we have $\leq B^2$ points in one slab(Other slabs will also have $\leq B^2$ points since we distribute the points evenly). In this case, we can simply construct a B^2-structure over these points and construct a leaf node of the tree. Since every point is stored in precisely one B^2-structure and each node is associate with one B^2-structure, the height of the tree is $O(\log_B \frac{N}{B^2})$.

Sorting the points requires $O(\frac{N}{B}\log_{\frac{M}{B}} \frac{N}{B})$, which is equal to $O(\frac{N}{B}\log_B N)$ if $M \geq B^2$. In each level, we scan all points once and distribute them to next level. This requires $O(\frac{N}{B})$ each level and $O(\frac{N}{B}\log_B \frac{N}{B^2}) = O(\frac{N}{B}\log_B N)$ in total. For each B^2-structure, we spend $O(B)$I/Os to get the corresponding B^2 points from each list A_i. A B^2-structure can be constructed in $O(\frac{B^2}{B}\log_{\frac{M}{B}} \frac{B^2}{B}) = O(B)$I/Os. Since there are only $\frac{N}{B^2}$ B^2-structures, we spend $O(\frac{N}{B})$ to construct all the B^2-structure. So the time complexity of our algorithm is bounded by $O(\frac{N}{B}\log_B N)$I/Os.

2 Problem 2

Problem definition:

Let N be the number of segments in S. Give an algorithm that finds the lower envelop in $O(\frac{N}{B}\log_B N)$I/Os, assuming that the memory size M is at least B^2.

Solution:

An external priority tree[Arg03] can solve this problem in the required bound. The main idea is performing a vertical plane sweep from left to right and handling insertion and deletion in a buffer tree[Arg03]. When we encounter the beginning of a segment, we insert its y value into the buffer tree, we delete it when we meet its ending. However, there are several subtle issues to be illustrated.

Just before the first deletion, we perform a buffer-emptying process on all nodes on the path from the root to the leftmost leaf using $O(\frac{M}{B}\log_{\frac{M}{B}}\frac{N}{B})$I/Os amortized. Then we can delete the $\alpha M(0 < \alpha \leq 1)$ smallest elements in the tree and store them in the main memory. Let S represents these elements. After this operation, every time an insertion comes, we first check whether its y value is smaller than any element in S. If it is, we insert it into S. We maintain the invariant that S is sorted. Since it's in main menory, this will induce no I/O cost. If S exceeds its capacity after this insertion, we can simply remove the largest element and insert it back into the buffer tree. Every time a deletion comes, we first check whether the element to be deleted is in S. We delete it from S if it's. Otherwise we 'insert' the deletion into the buffer tree. If S becomes empty after this deletion, we perform another buffer-emptying process on all nodes on the path from the root to the leftmost leaf.

We can detect the lower envelop by monitoring the change of the smallest element in S since following our strategy, the smallest *alive* element is alway inside S.

Maintaining S requires no I/Os if S is not empty. When S becomes empty, we actually perform a *deletemin*[Arg03] operation on the buffer tree. The number of *deletemin* operation is bounded by $O(\frac{N}{\alpha M})$ and each deletemin consumes $O(\frac{M}{B}\log_{\frac{M}{B}}\frac{N}{B})$I/Os amortized. Insertion and deletion are handled as in the normal buffer tree. So the time complexity of our algorithm is $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$I/Os. When $M \geq B^2$, $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}) = O(\frac{N}{B}\log_B N)$.

3 Bonus problem

Problem definition:

Consider a structure that can be built in $O(\frac{N}{B}\log_B N)$I/Os (where N is the

number of items indexed), and answer a query in $O(\log_B N)$I/Os. Improve logarithmic rebuilding to obtain a semi-dynamic structure that supports queries still in $O(\log_B^2 N)$I/Os, but insertion in $O(\frac{1}{\sqrt{B}} \log_B^2 N)$I/Os.

solution:

Let V be the set of N items we index and D be the index structure. Logarithmic rebuilding[Ben79] essentially partition the V into $\log_B N$ subset V_i of exponentially increasing size B^i and build a static structure D_i for each of these subsets. In order to achieve our required bound, we can adjust the size of each V_i into $(\sqrt{B})^i$. It turns out that we increase the number of D_i to $O(\log_{\sqrt{B}} N) = O(\log_B N)$. So our queries can still be answered in $O(\log_B^2 N)$ since D can answer a query in $O(\log_B N)$.

An insertion is handled by finding the first structure D_i such that $\sum_{j=1}^{i} |D_i| \leq (\sqrt{B})^i$, discarding all structures D_j, $j \leq i$, and building a new D_i from the objects in these structures using $O(((\sqrt{B})^i/B) \log_B N) = O(B^{i/2-1} \log_B N)$I/Os. Now because of the way D_i was chosen, we know that $\sum_{j=1}^{i-1} |D_i| > (\sqrt{B})^{i-1}$ which means that at least $B^{i/2-1/2}$ objects are moved from lower indexed structures D_j to D_i. If we divide the D_i construction cost between these objects, each of them has to pay $O(\frac{1}{\sqrt{B}} \log_B N)$I/Os. An object can at most move $O(\log_B N)$ times during N insertions since it never move from a higher to a lower index structure. Thus the amortized cost of an insertion is $O(\frac{1}{\sqrt{B}} \log_B^2 N)$I/Os.

References

[Arg03] Lars Arge. The Buffer Tree: A Technique for Designing Batched External Data Structures. *Algorithmica*, 37(1):1–24, 2003.

[Arg04a] Lars Arge. External Geometric Data Structures. In *COCOON*, page 1, 2004.

[Arg04b] Lars Arge. External Memory Geometric Data Structures. *EEF Summer School on Massive Datasets. Springer Verlag*, 2004.

[Ben79] Jon Louis Bentley. Decomposable Searching Problems. *Inf. Process. Lett.*, 8(5):244–251, 1979.

[GTVV93] Michael T. Goodrich, Jyh-Jong Tsay, Darren Erik Vengroff, and Jeffrey Scott Vitter. External Memory Computational Geometry. In *FOCS*, pages 714–723, 1993.