Reporting Overlaps in a Dynamic Interval Set by Filtering Search

Donald Mullis

Digital Equipment Corporation
Workstations and Servers

Abstract

The concept of *filtering search*[2] has been applied to the problem of reporting overlaps in a static set of intervals; this paper describes how it may be applied to the dynamic variation of that problem. The new algorithm retains the $O(\log n + k)$ reporting time and $O(n)$ space of the static algorithm, where k is the output size. We say that insertion and deletion require $O(\log n + k)$ time as well, with k regarded as the number of segments that would be reported by a hypothetical query with the update segment.

1 Review of filtering search

Refer to Chazelle[2] for a thorough explanation of the idea of filtering search. We reformulate his definition slightly here so that it plainly serves for a discrete as well as a continuous domain, and lends itself to dynamization.

We will indicate a semi-open interval with $[a, b)$, where $a, b \in \Re$ and $a < b$. Note that by definition, $[a, b)$ and $[b, c)$ do not intersect. Call an interval with explicitly defined endpoints, such as an element of the input, a *segment*. Call the stored set of n possibly overlapping segments $S = \{[a_i, b_i)\}_{1 \leq i \leq n}$. The task of the static algorithm is to report the segments in S that intersect a query segment q. Call the reported set $S(q)$, and as is customary, $k \equiv |S(q)|$.

$W(S)$ is a *window-list* encoding S. The jth window of the set records a lower bound point L_j. The W_j are stored in a searchable list according to their L_j, and each contains an unsorted list of just those segments of S that intersect $[L_j, L_{j+1})$. Note that interval $[L_j, L_{j+1})$ is not necessarily in S.

Let p be the number of windows in $W(S)$. $W(S)$ initially contains only W_1 with $L_1 \equiv -\infty$, and an implicit W_{p+1} that provides $L_{p+1} \equiv +\infty$. Thus each point of $\Re$ maps to exactly one window.

Since S is static, the L_j may be chosen by sorting all endpoints, then sweeping from low to high, introducing a new window at the point where a newly swept endpoint would otherwise cause the current window W_j to violate a *splitting predicate*

$$\mathcal{P}_{static}(W) \equiv (\delta \min_{x \in W} |S(x)| + \epsilon \geq |W|), \ \delta > 1, \ \epsilon \geq 1$$

where the integer constant parameters δ and ϵ control a time-space tradeoff. Splitting away a new window whenever $\mathcal{P}_{static}(W)$ becomes false evidently ensures that the number of segments linearly searched to report the k intersections of a query *point* with elements of W is $O(k)$, and therefore the reporting time (less the time to find W), is $O(k)$ as well. In his slightly different formulation, Chazelle[2] claims that performance for query with an *interval* is similar, and proves that total storage consumption is $O(n)$.

2 A dynamic variation

The algorithm we introduce here is dynamic in the sense that segment insertions and deletions may be interspersed with queries, with any of the operations requiring time at worst proportional to the number of segments k belonging to S that intersect q, plus $O(\log n)$ time to find some window containing q. The window may be located in time $O(\log n)$ by storing the window set in a threaded balanced tree, or in a skip list [5, 6]. Skip lists simplify the implementation.

As a shorthand, we will say that a *window* W_j intersects a segment if its associated interval $[L_j, L_{j+1})$ intersects the segment. A window's segment set may be partitioned as follows: those segments that each cover the window have an *A-part* in that window, and the rest, that merely intersect, have *B-parts*. Formally, $[a, b)$ intersects W_j in an A-part iff $a \leq L_j$ and $b \geq L_{j+1}$. This implies that every segment is itself partitioned by windows into zero, one, or two B-parts, and zero or more A-parts.

The following *simplification of the splitting predicate* aids our effort to achieve a dynamic algorithm. We set $\delta = 2$ and replace $\min_{x \in W_j} |S(x)|$ with the number of A-parts of the window denoted by $|A_j|$, producing

$$\mathcal{P}(W_j) \equiv (2|A_j| + \epsilon \geq |W_j|), \ \epsilon \geq 1$$

Note that $|A_j|$ is a lower bound on $\min_{x \in W_j} |S(x)|$. We will prove that a window list maintained by our dynamic algorithm (which satisfies $\mathcal{P}$) may be searched in the same order time and consumes the same order space as Chazelle's static window list. The reader may gain a better intuition for all of this by verifying manually that each window in Figure 1 satisfies $\mathcal{P}$.

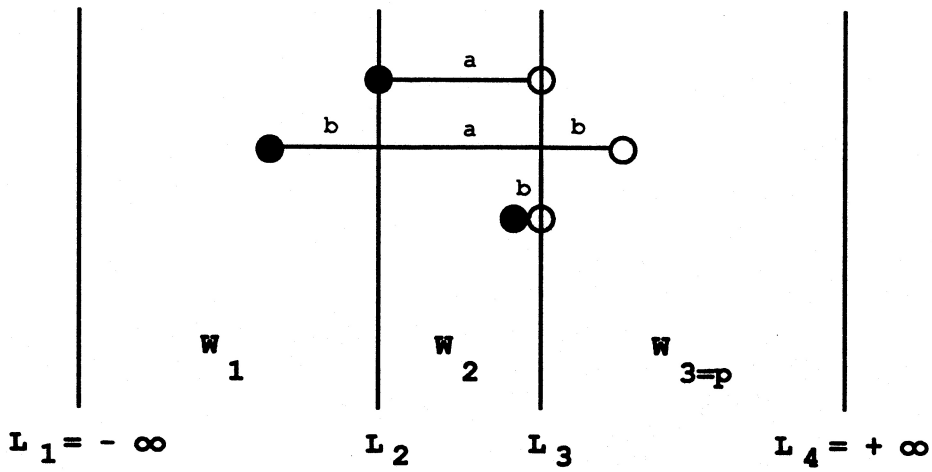

Fig. 1. A window-list that satisfies $\mathcal{P}$, with $\epsilon = 1$.

Some updates require adjacent windows to be merged. This circumstance is detected when $\mathcal{P}$ applied to their union evaluates true. Both insertions and deletions may be handled by the same three-step algorithm:

edit — update segment lists of intersected windows
split — split all windows intersected by q that violate $\mathcal{P}$
merge — merge adjacent window pairs iff at least one of the pair is intersected by q, and their union would satisfy $\mathcal{P}$

A superscript E, S, or M applied to a window or window list denotes its state following **edit**, **split**, or **merge** respectively. Thus,

$$W(S) \xrightarrow{\textbf{edit}} W^E(S') \xrightarrow{\textbf{split}} W^S(S') \xrightarrow{\textbf{merge}} W^M(S')$$

where S' denotes the segment set after its update by **edit**.

It may be worth noting that although the static algorithm produced a deterministic window list from its *a priori* segment set, the dynamic variation may allow many legal window-list encodings of a given segment set.

We shall prove upper bounds on reporting time and storage consumption with an argument based on **merge**, and later prove that the k term in the $O(\log n + k)$ update time is no worse than linear by showing that the number of segments of S examined is at worst proportional to the number intersected by q.

3 $O(\log n + k)$ reporting time, and $O(n)$ storage

The proof requires counting the number of endpoints of S that intersect the interval $[L_j, L_{j+1})$ of each window W_j. We'll call this quantity E_j.

Since each intersection of a segment with a window contains either an A-part or at least one endpoint:

$$|W_j| \leq E_j + |A_j| \tag{1}$$
$$|W_j \cup W_{j+1}| \leq E_j + E_{j+1} + |A_j \cap A_{j+1}| \tag{2}$$

Theorem 1 *Reporting time, after locating some q-intersected window, is* $O(k)$.

Proof: Recall that completion of **merge** requires that *no* possible union of abutting windows would satisfy $\mathcal{P}$:

$$\forall\, W_{1 \leq j < p} : \neg\mathcal{P}(W_j \cup W_{j+1})$$

Expanding $\mathcal{P}$, then substituting inequality (2) into the left-hand side,

$$\begin{aligned} 2\,|A_j \cap A_{j+1}| + \epsilon &< |W_j \cup W_{j+1}| \\ 2\,(|W_j \cup W_{j+1}| - (E_j + E_{j+1})) + \epsilon &< |W_j \cup W_{j+1}| \\ |W_j \cup W_{j+1}| &< 2\,(E_j + E_{j+1}) - \epsilon \\ |W_j| &< 2\,(E_j + E_{j+1}) - \epsilon \end{aligned} \tag{3}$$

Now that we have cardinality of windows on the left of the inequality, and number of endpoints on the right, we sum the right-hand side of inequality (3) over all but the last of the windows that intersect $q = [a, b)$. Call the leftmost and rightmost such windows W_{p_a} and W_{p_b} respectively. Then apply inequality (1) to W_{p_b}, and simplify:

$$\sum_{j=p_a}^{p_b} |W_j| \leq 2 \sum_{j=p_a}^{p_b-1} (E_j + E_{j+1}) - \epsilon\,(p_b - p_a) + |W_{p_b}|$$

$$\leq 2\left(\sum_{j=p_a}^{p_b-1} E_j + \sum_{j=p_a+1}^{p_b} E_j\right) + \left(E_{p_b} + |A_{p_b}|\right)$$

$$\leq 4\sum_{j=p_a}^{p_b} E_j + |A_{p_b}| \qquad (4)$$

In order to proceed, we need to establish an upper bound on the sum of the E_j. Observe that each of the k segments of $W(q)$ gives rise to two endpoints, which may or may not fall within $[L_{p_a}, L_{p_b+1})$; so $\sum_{j=p_a}^{p_b} E_j \leq 2k$. Combining with inequality (4) and noting that $|A_{p_b}| \leq k$ yields:

$$\sum_{j=p_a}^{p_b} |W_j| \leq 9k \quad \square$$

Theorem 2 *Overall storage is $O(n)$.*

Proof: Follows immediately from the $O(n)$ time to report the segments that overlap $[-\infty, +\infty)$, and the observation that all the storage of $W(S)$ would be examined. $\square$

4 $O(\log n + k)$ update time

An update segment q may be added to a window in constant time or, by analogy to the reporting task, deleted in $O(k)$ time, so **edit** consumes not more than $O(\log n + k)$ time.

The windows of $W^E(S)$ examined and possibly divided by **split** contain a total of segments numbering $O(k)$. We will later show that each segment is examined not more than a constant number of times. The two kinds of updates, insertion and deletion, happen to require that the windows containing B-parts of q and A-parts of q respectively be examined and possibly split. Each such window W^E is first tested against the splitting predicate. Upon failure of the predicate after **edit** for either insertion or deletion the window may be regarded as containing an excess of exactly one B-part or alternatively, a deficit of one A-part.

Given below is a procedure for splitting a window in such a state into successors that are each guaranteed to satisfy $\mathcal{P}$. Incidentally, it is this procedure that imposes the constraint that $\delta = 2$ on the algorithm as a whole.

Procedure *SplitWin*:

i. If any endpoint within W^E belongs to a B-part of a segment that half-covers W^E, then a split into two windows at that endpoint suffices. (One loses a B-part, the other gains an A-part.) — case **AB**

ii. Otherwise, set ℓ (resp. r) to the leftmost (rightmost) of right (left) endpoints of W^E.

iii. If $\ell \leq r$, then a split into two windows suffices, at any point between ℓ and r inclusive. (Both lose a B-part.) — case **BB**

iv. If $\ell > r$, then three split-windows are necessary and sufficient; boundaries at r and ℓ suffice. (Proof below.) — case **BAB**

Figure 2, below, may be of help in visualizing the cases.

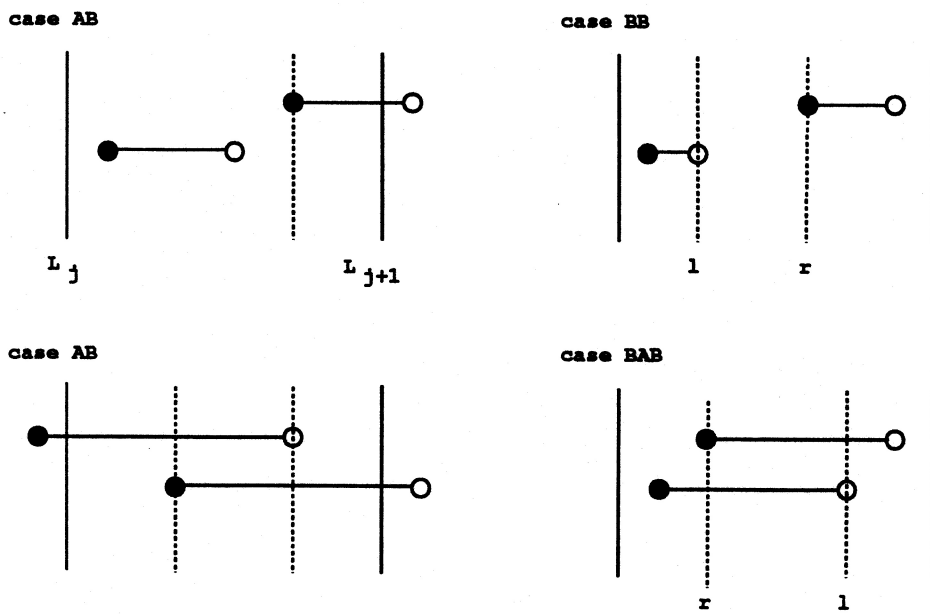

Fig. 2. Examples of the cases of *SplitWin*, with $\epsilon = 1$.

Lemma 1 *SplitWin finds the minimal number of split-windows, which number does not exceed three.*

Proof: Since SplitWin is invoked iff $\neg\mathcal{P}$, at least two split-windows are required. SplitWin produces three split-windows iff $\ell > r$. If $\ell > r$, the largest two split-windows possible that abut neighbors W^E_{j-1} and W^E_{j+1} would span $[L^E_j, r)$ and $[\ell, L^E_{j+1})$ (see case **BAB** of Fig. 2). Making either split-window any larger would either degenerate one of its A-parts into a B-part, or introduce a new B-part. So if $\ell > r$, at least three split-windows are required.

To see that three split-windows suffice, we need only prove that at least one B-part of W^E_j contributes an A-part to the central split-window that spans $[r, \ell)$. This is indeed the case, since r being the rightmost of the left endpoints, and ℓ the leftmost of the right endpoints, *all* segments intersecting the central window would be A-parts. □

Lemma 2 *The* **split** *step requires $O(k)$ time.*

Proof: The algorithm described above for testing and possibly splitting a window requires not more than a constant number of references to each segment of the window (by inspection of *SplitWin*). Only those windows that intersect q are examined, and together they contain $O(k)$ segments. □

Before turning to the **merge** step, we note that upon completion of **split**, $\mathcal{P}$ is satisfied by every window: $\forall\ W^S_j \in W^S(S') :\ \mathcal{P}(W^S_j)$. Storage consumption of $W^S(S')$ may exceed the upper bound of Theorem 2, but not by more than a factor of three, since each W^E_j was split at worst into thirds. So this transient storage requirement is still $O(n)$.

Finally, achieving $O(\log n + k)$ update time requires that **merge** execute in $O(k)$ time. To show that this is possible we first derive upper bounds for window-to-neighbor merge time, and a couple of supporting results.

Lemma 3 *Testing and possibly merging window W^S_j with $W^S_{j\pm1}$ requires $O(|W^S_j|)$ time.*

Proof: The worst case involves the pair of an endmost window intersected by q, and a non-intersected neighbor. Let W^S_j and W_{j+i} respectively identify the windows, where $i = \pm 1$.

Consider that $|A^S_j \cap A_{j+i}| \leq |A^S_j|$, and $|W^S_j \cup W_{j+i}| \geq |W_{j+i}|$. Merging is indicated iff $2\,|A^S_j \cap A_{j+i}| + \epsilon \geq |W^S_j \cup W_{j+i}|$, which would require

$$2\left|A^S_j\right| + \epsilon \geq \left|W_{j+i}\right| \tag{5}$$

Therefore, to trivially reject the possibility of a merge involving an excessively large $|W_{j+i}|$ requires no more knowledge of W_{j+i} than its cardinality, which we may have maintained for each window at no additional cost. The trivial rejection test is inconclusive only if $2|A_j^S| + \epsilon \geq |W_{j+i}|$, in which case an exact test (and possible merge) will require only $O(|W_j^S|)$ steps. □

Corollary 4 *Merging a window W_j^S with a neighbor increases its size to no greater than $3|W_j^S| + \epsilon$.*

Proof: Applying inequality (5),

$$\begin{aligned} \left|W_j^S \cup W_{j+1}^S\right| &\leq \left|W_j^S\right| + \left|W_{j+1}^S\right| \\ &\leq \left|W_j^S\right| + 2\left|A_j^S\right| + \epsilon \\ &\leq 3\left|W_j^S\right| + \epsilon \quad \square \end{aligned}$$

We now observe that if two windows W_j^S and W_{j+i}^S cannot be merged, neither can they be after one of the two has been modified by merging with its other neighbor. Formally,

$$\neg\mathcal{P}\left(W_j^S \cup W_{j+i}^S\right) \Rightarrow \neg\mathcal{P}\left(W_j^S \cup \left(W_{j+i}^S \cup W_{j+2i}^S\right)\right),\ i = \pm 1 \qquad (6)$$

This follows from the observation that merging two windows neither increases the number of A-parts nor decreases the number of segments relative to either original. We omit the details.

Lemma 5 *The* **merge** *step requires $O(k)$ time*

Proof: Invariant (6) implies that a window pair can only require merging if one of the pair intersects q. **Merge** may be implemented by a procedure that steps left-to-right through just the q-intersecting windows. Call the current window W_j^S, and test it for merging with W_{j-1}^S. By Lemma 3, a test and possible merge at the boundary requires $O(|W_j^S|)$ steps. The last q-intersected window is a special case in that it must be tested against neighbors both left and right (W_{j-1}^S and W_{j+1}^S), but Corollary 4 assures us that the cost of the second test and merge is still only $O(|W_j^S|)$. **Split** will have at worst tripled storage consumption for the windows that intersect q, so the total number of segments examined is $O(k)$. □

Theorem 3 *Run time for either insertion or deletion is $O(\log n + k)$.*

Proof: After finding the first window intersected by q, in $O(\log n)$ time, the three steps **edit**, **split**, and **merge** suffice to insert or delete a segment from $W(S)$, and each has been shown to be $O(k)$. □

algorithm	*domain* $[lo,hi)$	*storage*	*update time*	*query time*
naive	$[\Re,\Re)$	$\theta(n)$	$\theta(1)$	$O(n)$
static filtering search[2]	$[\Re,\Re)$	$\theta(n)$	$O(n)$	$O(\log n + k)$
static interval tree	$[\mathbf{U},\Re)$	$O(n+N)$	$O(\log N + \log n)$	$O(\log N + k)$
static segment tree	$[\mathbf{U},\mathbf{U})$	$O(n \log N)$	$O(\log N + \log n)$	$O(\log N + k)$
dynamic interval tree[3]	$[\Re,\Re)$	$\theta(n)$	$O(\log n)$	$O(\log n + k)$
dynamic segment tree[3]	$[\Re,\Re)$	$\theta(n)$	$O(\log n)$	$O(\log n + k)$
dynamic filtering search	$[\Re,\Re)$	$\theta(n)$	$O(\log n + k)$	$O(\log n + k)$

Table 1: Synopsis of overlap reporting algorithms.

5 Related work

The *static interval tree*[3, 4] and *static segment tree*[3] algorithms require a data structure proportional in size to the fixed domain from which endpoints are chosen, but allow a segment bounded by any two of those endpoints to be inserted or deleted at moderate cost. Chazelle's overlap reporting algorithm using filtering search is more restrictive yet, requiring the set of segments to be static. The *dynamic interval tree* and *dynamic segment tree*[3] algorithms, and the filtering search algorithm of this paper, are entirely dynamic.

Table (1) summarizes the essential features of the various algorithms. For all algorithms, n is the number of stored segments. The static tree algorithms require a finite $\mathbf{U} \subset \Re$; let $N = |\mathbf{U}|$. For queries, k is the number of segments reported; for updates k is the number of segments intersected by a newly updated element.

The update times of dynamic interval and dynamic segment tree search do better that of dynamic filtering by the linear term in k; however, this is unimportant for some applications. Consider that some number of queries must be performed (since they are the only way to extract information from the data structure), at cost $O(\log n + k)$ for either algorithm. If the total cost of the queries is proportional to dynamic filtering search's total cost for updates, then query dominates the running time of either algorithm, which will differ by only a constant factor. This is indeed the case for at least one VLSI design-rule checking application[4], in which each segment to be inserted is first used to query those already stored.

6 Applications

Dynamic interval or segment trees are not trivial to implement [3]. Static interval and segment trees are simpler, but require a separate pre-processing step in which the tree is initialized with the finite set of allowable endpoints. Dynamic filtering search suffers from neither of these drawbacks.

Preparata and Shamos[4] describe applications of interval overlap query to problems in design-rule checking of VLSI circuits and concurrent access to databases. Dynamic filtering search may offer solutions that are simpler than the tree-based algorithms described there and have the same computational complexity.

A key idea from filtering search, flattening the data structure in proportion to the volume of output of a query, has been applied to static segment trees in a VLSI application by Bonapace and Lo[1]. Again, a solution based on dynamic filtering search might offer a practical improvement.

7 Acknowledgements

The author would like to thank Stephen Harrison and John Hershberger of Digital's Systems Research Center for their thoughtful comments.

References

[1] Bonapace, C.R. and Lo, C. An $O(n \log m)$ Algorithm for VLSI Design Rule Checking, *Proceedings of the 26th ACM/IEEE Design Automation Conference*, 1989, pp. 503–507.

[2] Chazelle, B. Filtering Search: A New Approach to Query-Answering, *SIAM J. on Comp.*, no. 15, 1986, pp. 703–724.

[3] Mehlhorn, K. *Multi-dimensional Searching and Computational Geometry*, Springer-Verlag, Berlin, 1984.

[4] Preparata, F. and Shamos, M. I. *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.

[5] Pugh, W. Skip Lists: A Probabilistic Alternative to Balanced Trees, *Communications of the ACM*, Vol. 33, no. 6, June 1990, pp. 668–676.

[6] Pugh, W. A Skip List Cookbook, Tech. Rep. CS-TR-2286.1, Dept. of Computer Science, Univ. of Maryland, College Park, MD, July 1989.