To appear in *Computational Geometry: Theory and Applications*

Experiments on the Practical I/O Efficiency of Geometric Algorithms: Distribution Sweep vs. Plane Sweep *

Yi-Jen Chiang†

(April 1995; revised June 1997)

Abstract

We present an extensive experimental study comparing the performance of four algorithms for the following *orthogonal segment intersection problem*: given a set of horizontal and vertical line segments in the plane, report all intersecting horizontal-vertical pairs. The problem has important applications in VLSI layout and graphics, which are large-scale in nature. The algorithms under evaluation are our implementations of distribution sweep and three variations of plane sweep. Distribution sweep is specifically designed for the situations in which the problem is too large to be solved in internal memory, and theoretically has optimal I/O cost. Plane sweep is a well-known and powerful technique in computational geometry, and is optimal for this particular problem in terms of internal computation. The three variations of plane sweep differ by the sorting methods (external vs. internal sorting) used in the preprocessing phase and the dynamic data structures (B tree vs. 2-3-4 tree) used in the sweeping phase. We generate the test data by three programs that use a random number generator while producing some interesting properties that are predicted by our theoretical analysis. The sizes of the test data range from 250 thousand segments to 2.5 million segments. The experiments provide detailed quantitative evaluation of the performance of the four algorithms, and the observed behavior of the algorithms is consistent with their theoretical properties. This is, to the best of our knowledge, the first experimental algorithmic study comparing the practical performance between external-memory algorithms and conventional algorithms with large-scale test data.

Keywords: Segment intersection, plane sweep, distribution sweep, B-tree, external-memory algorithm, implementation, experimentation, probabilistic analysis.

---

*An extended abstract of this paper was presented at the *4th Workshop on Algorithms and Data Structures*, Kingston, Ontario, Canada, August, 1995.

†Department of Applied Mathematics and Statistics, SUNY Stony Brook, Stony Brook, NY 11794-3600; yjc@ams.sunysb.edu. This work was conducted when the author was at the Computer Science Department of Brown University. Research supported in part by the National Science Foundation, by the U.S. Army Research Office, and by the Office of Naval Research and the Advanced Research Projects Agency.

1 Introduction

Input/Output (I/O) communication between fast internal memory and slower external memory is the major bottleneck in many large-scale applications. The issue of this I/O bottleneck is becoming more and more important, since problem sizes of applications are getting larger and larger, and technological advances are increasing CPU speeds at an annual rate of 40–60% while disk transfer rates are only increasing by 7–10% annually [32]. Due to this increasing importance, more and more attention has been given to the development of I/O-efficient algorithms in recent years. Most of the developed algorithms, however, are shown to be efficient only *in theory*, and their performance *in practice* is yet to be evaluated. In particular, all such algorithms assume that the internal computation is free compared to the I/O cost, which also has to be justified in practice. In this paper, we establish the practical efficiency of one such algorithm by an extensive experimental study.

Previous Related Work

As mentioned above, most of the previous work on I/O-efficient computation is theoretical. Early work on algorithms for parallel disk systems concentrates largely on fundamental problems such as sorting, matrix multiplication, and FFT [1, 28, 38]. The main focus of this early work is therefore directed at problems that involve permutation at a basic level. Indeed, just the problem of implementing various classes of permutation has been a central theme in external-memory I/O research [1, 14, 15, 17, 38]. Also, a general connection between the comparison-complexity and the I/O-complexity of a given problem is shown in [4].

More recently, external-memory research has moved towards solving graph and geometric problems. Work on graph problems includes transitive closure computations [35], some graph traversal problems [23], and memory management problems for maintaining connectivity information and paths on graphs [19]. Recently, Chiang *et al.* [12] present a collection of new techniques for designing and analyzing I/O-efficient graph algorithms, and apply these techniques to a wide variety of specific problems. Concurrent to the work presented in this paper, Arge [3] considers the problem of manipulating ordered binary-decision diagrams in external memory. For geometric problems, Goodrich *et al.* [24] study a number of problems in computational geometry and develop several paradigms for I/O-optimal geometric computations. Further results in this area have been obtained in [20, 39]. Also, Kanellakis *et al.* [25] and Ramaswamy and Subramanian [31, 34] give efficient data structures for performing range searching in external memory. Concurrent to the work in this paper, a new data structure called *buffer tree* and its applications are given in [2, 5], and an external-memory version of the directed topology tree ([21]) called *topology B-tree* is given in [11].

For excellent examples of experimental work in computational geometry, see Bentley [7, 8, 9, 10]. As for experimental work on I/O-efficient computation, concurrent to our work Vengroff builds an environment called TPIE for programming external-memory algorithms as he proposed earlier in [36], and also Vengroff and Vitter [37] report some benchmarks of TPIE on sorting and matrix multiplication. This work, however, is mainly on providing a programming environment and not on performance comparisons between external-memory algorithms and conventional algorithms. Also worth noting is the work by Ramaswamy and Kanellakis [30], who study the problem of indexing a class hierarchy in Object Oriented Databases. They propose a technique called *class-division*, and show by experiments that in the average case, class-division performs far less I/O's than the *class hierarchy index* technique most popular today. Their experimental setting is somewhat different from ours, however, in that they only run the experiments under a *simulation tool*, and thus they can only measure the *numbers of requested I/O operations*, without being able to measure the *actual* *running times.* (As shown by our experiments, the numbers of requested I/O operations sometimes do not reflect the actual running times.)

Our Results

We present an extensive experimental study comparing the performance of four algorithms for the following *orthogonal segment intersection problem*: given a set of horizontal and vertical line segments in the plane, report all intersecting horizontal-vertical pairs. The problem has important applications in VLSI layout and graphics, which are large-scale in nature. The algorithms under evaluation are our implementations of distribution sweep of Goodrich *et al.* [24] and three variations of plane sweep [29]. Distribution sweep theoretically has optimal I/O cost [24][1]. Plane sweep is a well-known and powerful technique in computational geometry, and is optimal for this particular problem in terms of internal computation [29]. The three variations of plane sweep differ by the sorting methods (external merge sort [1] vs. internal merge sort) used in the preprocessing phase and the dynamic data structures (B tree [6, 13, 16] vs. 2-3-4 tree [16]) used in the sweeping phase. Intuitively, the four algorithms are spread along a design spectrum from an algorithm designed exclusively for external memory (distribution sweep) to one designed exclusively for internal memory using an infinite-size virtual memory assumption. We generate the test data by three programs that use a random number generator while producing some interesting properties that are predicted by our theoretical analysis. The sizes of the test data range from 250 thousand segments to 2.5 million segments. The experiments provide detailed quantitative evaluation of the performance of the four algorithms, and the observed behavior of the algorithms is consistent with their theoretical properties. In particular, our experiments show that while the performance of the three variations of plane sweep depends heavily on the average size of the data structure used during sweeping and can be very poor, the performance of distribution sweep is both steady and efficient. Also, distribution sweep does not require a large amount of main memory to perform well. Moreover, the performance of the algorithms assuming an infinite-size virtual memory is very undesirable, and distribution sweep should always be the choice among the four algorithms whenever there is a potential I/O issue.

The contribution of this work can be summarized as follows:

- We have presented (to the best of our knowledge) the first experimental algorithmic study comparing the practical performance between external-memory algorithms and conventional algorithms with large-scale test data.
- We have generated test data with interesting properties that are predicted by our theoretical analysis. In particular, we give techniques for analyzing the expected number of intersections and the average number of vertical overlaps among vertical segments in the data sets generated, which may be of independent interest.
- We have implemented distribution sweep, three variations of plane sweep and external merge sort under a uniform experimental framework, so that some resource usage can be parameterized and various statistic information related to performance can be obtained and analyzed. The implementations handle all degeneracies and are robust.
- We have presented the first experimental study on the four algorithms for the important orthogonal segment intersection problem with large-scale test data, and established the practical efficiency of distribution sweep.

[1] Concurrent to our work, another theoretically I/O-optimal algorithm for the problem is proposed by Arge using buffer tree [2].

Organization of the Paper

The rest of the paper is organized as follows. In Section 2 we overview the four algorithms under evaluation. Details on the experimental setting are given in Section 3. In Section 4, we summarize our experimental results in nine charts and give a detailed analysis of the performance of the four algorithms. Finally, we conclude the paper in Section 5.

2 The Algorithms Under Evaluation

The four algorithms considered in this paper are our implementations of distribution sweep, denoted `Distribution`, and three variations of plane sweep, denoted `B-Tree`, `234-Tree`, and `234-Tree-Core`, described next. To discuss the I/O cost, let N be the total number of segments in the given input, K the number of intersecting pairs that must be reported, and M and B the numbers of segments that can fit into the main memory and into a page, respectively. Each I/O operation transfers one page of data.

Before discussing the four algorithms under evaluation, we first briefly review the I/O costs of some fundamental operations in external memory: *scanning, sorting*, and *permutation.* To scan through the entire data set once, we need $\lceil N/B \rceil$ I/O operations, since the data set occupies $\lceil N/B \rceil$ pages in disk. The I/O costs for external sorting and permutation are $\Theta(sort(N))$ and $\Theta(\min\{N, sort(N)\})$, respectively, where $sort(N) = \frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B}$ [1]. Notice that for typical values in real I/O system, the $sort(N)$ term is smaller than the N term. For example, if one page holds 10^3 items ($B = 10^3$) and there are 10^4 pages in the main memory ($M/B = 10^4$, true for a typical workstation with 32Mb main memory and 4Kb page size), then $N < sort(N)$ only when $N > B \cdot (\frac{M}{B})^B = 10^{4,003}$. This shows that in external memory permutation has the *same* I/O cost as sorting in practice, although in internal memory permutation needs only $\Theta(N)$ time while sorting needs $\Theta(N \log N)$ time. Equivalently, this is to say that the B term is much larger than the $\log_{\frac{M}{B}} \frac{N}{B}$ term in practice. We will see later, as shown by our experiments, the importance of a factor of $\frac{1}{B}$ in I/O cost in the speed-up of actual performance.

We now describe the four algorithms under evaluation.

2.1 Three Variations of Plane Sweep

The well-known *plane sweep* paradigm [29] is a powerful technique in computational geometry, and is optimal for the orthogonal segment intersection problem in terms of internal computation. The method consists of preprocessing and sweeping phases. In the preprocessing phase, we sort all endpoints by the y-coordinates in non-decreasing order. In the sweeping phase, we (conceptually) use a horizontal sweep line to sweep the plane from bottom to top, and use a dynamic data structure, typically a balanced search tree, to keep the objects currently intersecting the sweep line. Operations (object insertions/deletions or queries) are performed only when some *events* are encountered by the sweep line. In our problem, the objects being maintained are vertical segments, and the events are the endpoints of all segments. When a bottom endpoint of a vertical segment is encountered, it is inserted to the data structure; the segment remains intersecting the sweep line until its top endpoint is encountered, at which time it is deleted from the data structure. When an endpoint of a horizontal segment s is encountered, we search the data structure to find and report all vertical segments intersecting s, i.e., those segments whose x-coordinates are contained in the x-interval spanned by s. The sequence of events during the sweeping process is given by sorting in the preprocessing phase. Using any dynamic balanced tree where insertions, deletions and searches can each be performed in $O(\log N)$ time, plane sweep takes optimal $\Theta(N \log N + K)$ time in terms of internal computation.

Our three variations of plane sweep differ by the sorting methods and the dynamic data structures used. The first variation, `B-Tree`, uses external merge sort [1] and a B tree [6, 13, 16]; this is a direct way to implement plane sweep in secondary memory. In the first phase, the number of I/O operations performed is optimal $\Theta(sort(N)) = \Theta(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ [1]. In the second phase, insertions, deletions and searches in a B-tree can each be performed in $O(\log_B \frac{N}{B})$ I/O operations [6, 13, 16], and the I/O cost for reporting the answers is $\Theta(K/B)$ by carefully "packing" the answers to pages in disk. Thus the overall I/O cost in the second phase is $O(N \log_B \frac{N}{B} + \frac{K}{B})$. The second variation, `234-Tree`, uses external merge sort and a 2-3-4 tree [16] (a balanced tree equivalent to a red-black tree [16]), viewing the internal memory as having an infinite size and letting the virtual memory feature of the OS handle page faults during the second (sweeping) phase. It has the same I/O cost in the first phase and $O(N \log N + \frac{K}{B})$ I/O cost in the second phase, since in the worst case each (virtual) memory access can cause a page fault and requires one I/O operation, and yet by packing the answers to pages the reporting I/O cost is still $\Theta(K/B)$. Finally, the third variation, `234-Tree-Core`, uses internal merge sort and a 2-3-4 tree, letting the OS handle page faults all the time. Again, in the worst case each virtual memory access causes a page fault, therefore the I/O costs in the first and second phases are $O(N \log N)$ and $O(N \log N + \frac{K}{B})$, respectively. Viewing the internal memory as virtually having an infinite size is conceptually the simplest, and is actually the most commonly used strategy today in practice.

2.2 Distribution Sweep

Distribution sweep [24] is an external-memory version of plane sweep based on the subdivision technique used in the "distribution sort" algorithms of [1, 27, 38]. When applied to the orthogonal segment intersection problem, it works as follows.

In the preprocessing phase, we sort the endpoints of all segments into two lists, one by x and the other by y. Again we use external merge sort. The list sorted by x is used to locate the medians which we will use to split the input into $\lfloor \frac{M}{B} \rfloor$ vertical strips. The list sorted by y is used to perform the sweep, moving bottom up. Associated with each strip γ_i is an *active list* A_i that maintains the vertical segments inside γ_i that intersect the horizontal sweep line.

During the sweep, if the bottom endpoint of a vertical segment is encountered, it is inserted into the active list A_i of the strip in which the segment lies, and later it is deleted from A_i when its top endpoint is encountered. When an endpoint of a horizontal segment s is met, we locate the two strips γ_i and γ_j containing the two endpoints of s, $i <= j$, and report all vertical segments currently in the active lists $A_{i+1}, ..., A_{j-1}$. Note that strips $\gamma_{i+1}, ..., \gamma_{j-1}$ are completely spanned horizontally by s. This reports all vertical segments intersecting s except for those lying in γ_i and γ_j, which will be processed in the next level of recursion. After the sweep is complete in the top level, we recursively perform the same procedure to each strip γ_i. The recursive process continues until the size of the subproblem falls below M, at which time we simply solve the problem in main memory.

Based on the above idea, distribution sweep actually uses a "lazy deletion" policy. Each active list A_i can be viewed as a stack, with the topmost page maintained in the internal memory and the remaining pages maintained in disk. Whenever the internal page is full, it is written to an appropriate place in disk corresponding to the same active list. Now, the top endpoints of vertical segments become "no action" events, and the deletions are performed during reporting: in the reporting process, instead of just reporting each vertical segment t in active list A_i, we check the top endpoint of t to see if its deletion time has passed. If so we delete t, otherwise we report t and

Algorithm	First Phase	Second Phase
`Distribution`	$\Theta(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$	$\Theta(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}+\frac{K}{B})$
`B-Tree`	$\Theta(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$	$O(N\log_{B}\frac{N}{B}+\frac{K}{B})$
`234-Tree`	$\Theta(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B})$	$O(N\log N+\frac{K}{B})$
`234-Tree-Core`	$O(N\log N)$	$O(N\log N+\frac{K}{B})$

Table 1 Theoretical worst-case I/O costs of the four algorithms under evaluation.

retain t in A_i. The total I/O cost is optimal $\Theta(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B}+\frac{K}{B})$. Notice that distribution sweep needs two sortings[2] as opposed to just one sorting in plane sweep.

2.3 Theoretical Comparisons

In Table 1 we summarize the theoretical worst-case I/O costs of the four algorithms under evaluation, listed in the order of increasing I/O costs. We make the following *theoretical comparisons* of the algorithms. Observe that `B-Tree` is worse than `Distribution` in the second phase: First, in the log base, B is usually a little bit smaller than M/B, but the fact that this difference only occurs in the log base makes it less important. Secondly and more importantly, N is much larger than N/B — this means that *batched* computation results in a better I/O cost in external-memory computation. Next, `234-Tree` is worse than `B-Tree` in the second phase, with difference (2 vs. B) occurring in the log base. Although the difference only occurs in the log base, $\log_2() - \log_B()$ is much larger than $\log_B() - \log_{\frac{M}{B}}()$, and the multiplication of N to the log terms makes the difference much, much larger. Finally, `234-Tree-Core` is worse than `234-Tree` in the first phase. The two differences respectively in the log base (2 vs. M/B) and in the multiplicative factor (N vs. N/B) are both very large, making the overall I/O cost of `234-Tree-Core` much worse. We will see how those different factors discussed above affect the actual performance of the algorithms in our experiments.

3 Experimental Setting

3.1 Generation and Analysis of the Test Data

We use three programs to generate our test data; all of them use a random number generator that gives a uniform distribution. The programs randomly generate several attributes of a segment such as its length, position, and type (horizontal/vertical), and also maintain certain properties to make the test data interesting. We want to control the following two properties of the test data:

(1) the number of horizontal-vertical intersecting pairs; and

(2) the average number of vertical overlaps among vertical segments, that is, the average number of vertical segments "cut" by the horizontal sweep line l *when* l *is passing through an event.* The average is taken over all sweeping events.

Notice that the number in (1) is the size of the reported answers, and the average number in (2) is exactly the average number of items stored in the data structure when an update/query

[2]Another theoretically I/O-optimal algorithm as mentioned before, using a buffer tree to implement the plane sweep, needs only one sorting and yet the data-structure manipulations [2] are more complicated than distribution sweep.

operation is performed during sweep. In other words, (1) corresponds to the reporting I/O cost of the algorithms, and (2) corresponds to the average tree size in plane sweep (as well as the average size of a collection of all active lists in distribution sweep), indicating whether or not the trees in `234-Tree` and in `234-Tree-Core` can fit in the main memory. Certainly (1) and (2) are two important properties that can affect the performance of the algorithms.

We first address the issue regarding (1). Observe that if we just randomly generate segments with lengths uniformly distributed over $[0, N]$, place them randomly (and uniformly in each dimension) in a square with side length N, and make horizontal and vertical segments equally likely to occur, then the number K of intersections is $\Theta(N^2)$ (obtained by an analysis similar to Theorem 1 below). In this case, any algorithm has $\Omega(\frac{N^2}{B})$ reporting I/O cost, which dominates the searching I/O costs in all four algorithms under evaluation. In fact, the following brute-force algorithm performs equally well: for each segment, check all the other $N-1$ segments for intersections; the I/O cost is $O(N \cdot \frac{N}{B}) = O(\frac{N^2}{B})$. Certainly this kind of test data is undesirable because it makes all four algorithms, as well as the above brute-force algorithm, have the same $\Theta(\frac{N^2}{B})$ I/O cost and thus cannot effectively distinguish the performance of the algorithms.

We try to generate test data such that (1) the number of intersections is small so that the searching I/O cost dominates the reporting cost, and (2) the average number of vertical overlaps varies in a controlled way so that we can observe the effect of the average data-structure size to the running time. Our three data-generation programs are denoted `gen-short`, `gen-long`, and `gen-rect`, and the data sets generated are correspondingly denoted `data-short`, `data-long` and `data-rect`. The programs generate test data with distinct structures regarding the number of intersections and the average number of vertical overlaps. Also, all three programs decide the type (horizontal or vertical) of the current segment being generated by tossing a fair coin, which is simulated by using the random number generator.

Program `gen-short` generates short segments whose lengths are uniformly distributed over $[0, \sqrt{N}]$. The segments are randomly placed in an $N \times N$ square Q. More specifically, for the left endpoints of horizontal segments, the distances to the left and bottom sides of Q are uniformly distributed over $[0, N - \sqrt{N}]$ and over $[0, N]$, respectively. Similarly, for the bottom endpoints of vertical segments, the distances to the left and bottom sides of Q are uniformly distributed over $[0, N]$ and over $[0, N - \sqrt{N}]$, respectively. A sample data set with $N = 36$ is shown in Fig. 1. To simplify the discussion, we assume that the coordinate of the lower-left corner of Q is $(0, 0)$ (in the actual program this coordinate is $(-0.5N, -0.5N)$).

The expected number of horizontal-vertical intersecting pairs in `data-short` is analyzed in the following theorem.

Theorem 1 *Let K be a random variable for the number of horizontal-vertical intersecting pairs in* `data-short`. *Then $E[K] = \frac{1}{16}(N-1)$.*

Proof: See the Appendix. □

Figure 2 shows the actual numbers of intersections with respect to data size N, for all three data sets generated. The observed K values are indeed $\frac{1}{16}N$ for `data-short`.

Now we proceed to analyze for `data-short` the average number of vertical overlaps. Recall that this is the average number of vertical segments "cut" by the horizontal sweep line l *when l is passing through an event*, where the average is taken over all sweeping events. A related problem has been studied in [26]. Intuitively, we would estimate this average number to be proportional to the average length of vertical segments, which is $\Theta(\sqrt{N})$. A rigorous analysis is given next.

Theorem 2 *Let V be a random variable for the number of vertical segments in* `data-short` *cut by the horizontal sweep line l for an event. Then $E[V] = \frac{1}{4}\sqrt{N} + O(1) \sim \frac{1}{4}\sqrt{N}$.*

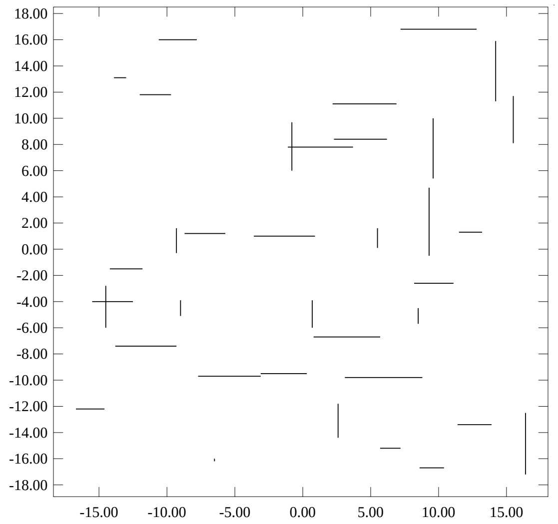


Figure 1: A sample data set of `data-short` with 36 segments.

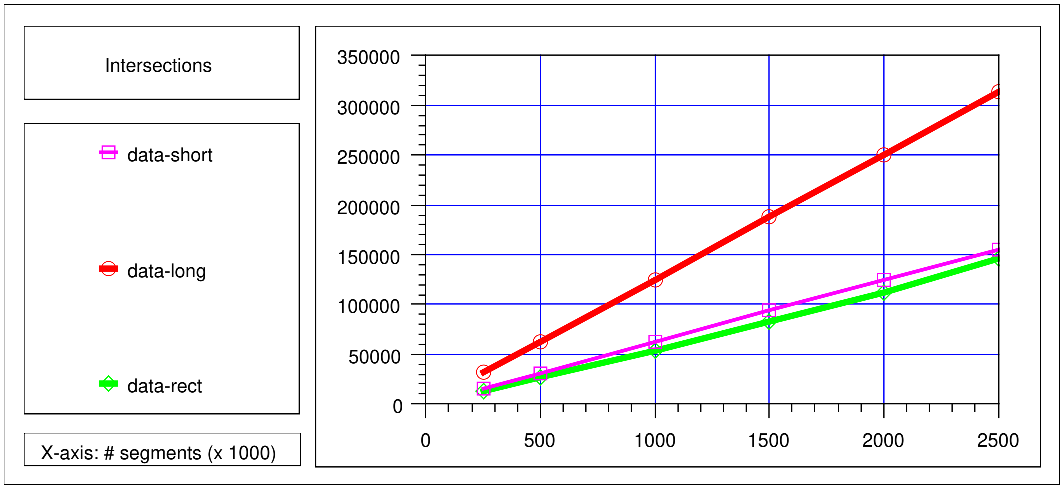


Figure 2: The actual numbers of intersections with respect to the number of segments in all three data sets `data-short`, `data-long` and `data-rect`.

Proof: See the Appendix. □

It is interesting to see that $\frac{1}{4}\sqrt{N}$ can be interpreted as the product of the probability of a segment being vertical ($\frac{1}{2}$) and the average length of vertical segments ($\frac{1}{2}\sqrt{N}$). Figure 3 shows the actual values of the average number of vertical overlaps with respect to data size N, for all three data sets generated. Note that these values are non-integral since we are taking the average over the sweeping events. The values for `data-short` are much smaller than those for the others and are almost invisible in Fig. 3; we thus list for `data-short` the actual values, and also compare them with the analyzed values, in Table 2. It can be seen that the observed values are indeed $\frac{1}{4}\sqrt{N}$.

In the remaining two data sets, we want to increase the average number of vertical overlaps so that we can observe the behavior of the algorithms when the I/O issue is much more severe. At the same time, we still want to keep the number of intersections small.

N : # segments ($\times 10^3$)	250	500	1000	1500	2000	2500
Actual value	125.23	176.74	249.98	306.40	353.58	395.60
$\frac{1}{4}\sqrt{N}$ ($\sim E[V]$)	125	176.78	250	306.19	353.55	395.28

Table 2 The actual and analyzed values of the average number of vertical overlaps in data set `data-short`.

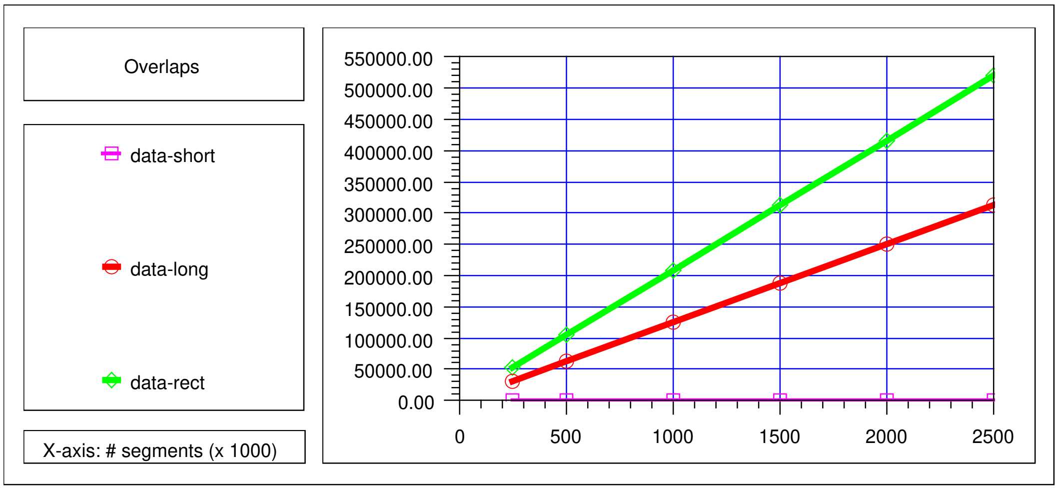


Figure 3: The actual values of the average number of vertical overlaps with respect to the number of segments in all three data sets `data-short`, `data-long` and `data-rect`.

Program `gen-long` generates short as well as long segments. For a horizontal segment, the length is assigned $\sqrt{N}$ (short segment); for a vertical segment, the program tosses a coin, giving length $\sqrt{N}$ (short segment) if the outcome is a head and N (long segment) otherwise. The horizontal and vertical short segments are randomly placed in the $N \times N$ square Q in the same way as described in `gen-short`. As for vertical long segments, the bottom endpoints are placed randomly in an $N \times N$ square Q' whose lower-right corner coincides with the lower-left corner of Q, such that the distances from the bottom endpoints to the left and bottom sides of Q' are both uniformly distributed over $[0, N]$. Notice that all vertical long segments are outside square Q and thus do not introduce any intersections; we create them solely for the purpose of increasing the number of vertical overlaps. Therefore we have about $\frac{1}{2}N$ short horizontal segments, $\frac{1}{4}N$ short vertical segments, and $\frac{1}{4}N$ long vertical segments which cause no intersections. A sample data set with $N = 36$ is shown in Fig. 4. Using similar analysis methods as given before, we have that $E[K] = \frac{1}{8}(N-1)$ and $E[V] = \Theta(N)$; the latter bound shows that $E[V]$ is asymptotically proportional to the length of the *long* vertical segments. The observed K values of `data-long` are indeed $\frac{1}{8}N$ (see Fig. 2), and the observed values of the average number of vertical overlaps are also $\frac{1}{8}N$ (see Fig. 3).

In program `gen-rect`, we generate horizontal and vertical segments with lengths uniformly distributed over $[20, 60]$ and over $[0, 2N]$, respectively. The left endpoints of horizontal segments are randomly placed inside an $80N \times N$ rectangle R (with horizontal side length $80N$), such that the distances to the left and bottom sides of R are uniformly distributed over $[0, 80N]$ and over $[0, N]$, respectively. The vertical segments are placed as follows: in the x-direction, the distance between the left side of R and the *i-th vertical* segment is $(i-1) \times 160$, $i = 1, 2, \cdots$; in the y-direction, the distances from the bottom endpoints to the bottom side of R are uniformly distributed over $[0, N]$. A sample data set with $N = 36$ is shown in Fig. 5. It is easily seen from the construction

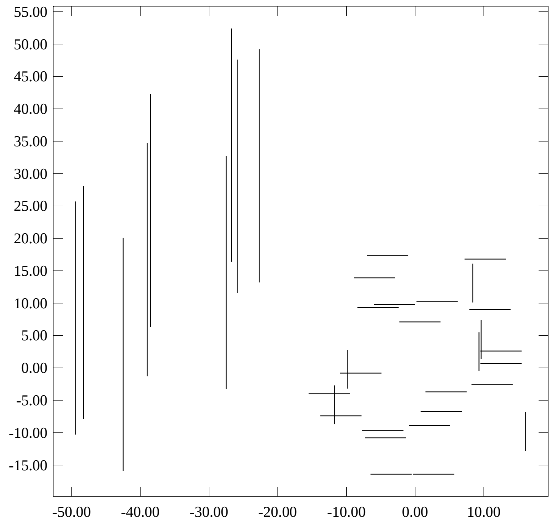


Figure 4: A sample data set of `data-long` with 36 segments. (Remark: the seemingly longest horizontal segment actually consists of two horizontal segments very close to each other.)

that each horizontal segment can intersect at most one vertical segment, so that $K = O(N)$. Using similar analysis methods as given before, we have that $E[K] = \Theta(N)$ and also $E[V] = \Theta(N)$. Again the latter bound shows that $E[V]$ is asymptotically proportional to the average length of vertical segments. From Figures 2 and 3 we see that the actual K values are $\frac{1}{18}N$, and the actual values of the average number of vertical overlaps are $\frac{1}{4.8}N$.

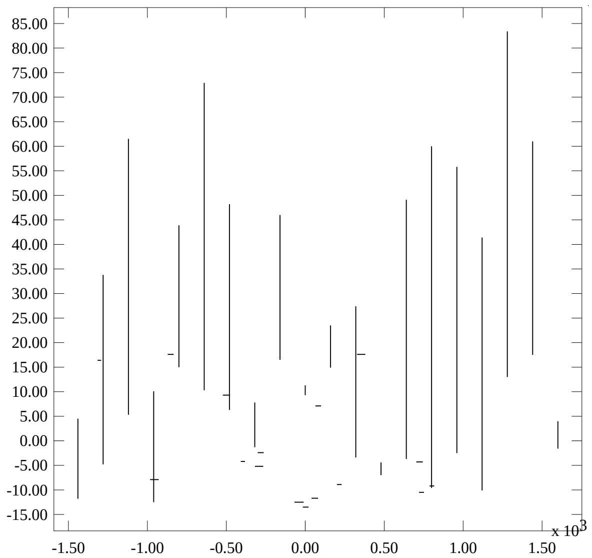


Figure 5: A sample data set of `data-rect` with 36 segments.

We summarize in Table 3 the actual values of the number of intersections and of the average number of vertical overlaps, expressed in terms of N, of all three data sets. Notice that `data-short`,

Data set	Num. intersections	Ave. num. vertical overlaps
`data-short`	$\frac{1}{16}N$	$\frac{1}{4}\sqrt{N}$
`data-long`	$\frac{1}{8}N$	$\frac{1}{8}N$
`data-rect`	$\frac{1}{18}N$	$\frac{1}{4.8}N$

Table 3 Actual values of the number of intersections and of the average number of vertical overlaps, expressed in terms of the data set size N, of the three data sets.

`data-long`, and `data-rect` are listed in the order of increasing vertical overlaps.

3.2 Computing Environment and Performance Measures

We perform the experiments in the Brown Computer Science Department on a Sun Sparc-10 workstation, which is running under Solaris 2.4 and is a multi-user distributed system. The main memory size is 32Mb (of which 26Mb are actually available for applications) and one page is of size 4Kb. Our performance measures are running time, number of I/O operations, and number of page faults. Notice the difference between *I/O operations* and *page faults*: the former means the read and write operations issued by the process, and can be viewed as *explicit* I/O's; the latter means the read and write operations issued by the OS when page faults occur (i.e., when the process accesses some virtual memory addresses not currently mapped to the physical main memory), and can be viewed as *implicit* I/O's.

The running time is our ultimate concern. Unlike previous experimental work, the CPU time does not correctly reflect the performance of the algorithms we want to measure, since our main concern is the amount of time in which the CPU is sleeping waiting for the I/O's and the page faults. The elapsed (wall-clock) time, although includes the CPU time as well as the time for I/O's and page faults, does not correctly reflect the performance either, since in a multi-user system the elapsed time may incorrectly include the running times of other processes. To obtain the correct running time, our solution is to measure the elapsed time while *making sure that no running times of other processes are included.* To do so, we note that there are two classes of processes supported in the system: the *time-sharing* class (usually running under the user mode, with the process priority dynamically adjusted by the system according to the system status), and the *real-time* class (usually running under the super-user mode, with the process priority pre-set and fixed during a pre-specified time period). By using the super-user privilege, we always set (via the `priocntl` command) our running process to be in the real-time class and to have the highest priority during the entire execution. In doing so, our process is put to sleep while waiting for I/O's or page faults, but is guaranteed to be immediately put to run whenever it is runnable. We then measure the elapsed time, which is now the accurate total running time of the process. Also, we use the local disk as the secondary memory, so that the performance is not affected by the network traffics or the network file servers. This resolves the issue of how to measure the running time.

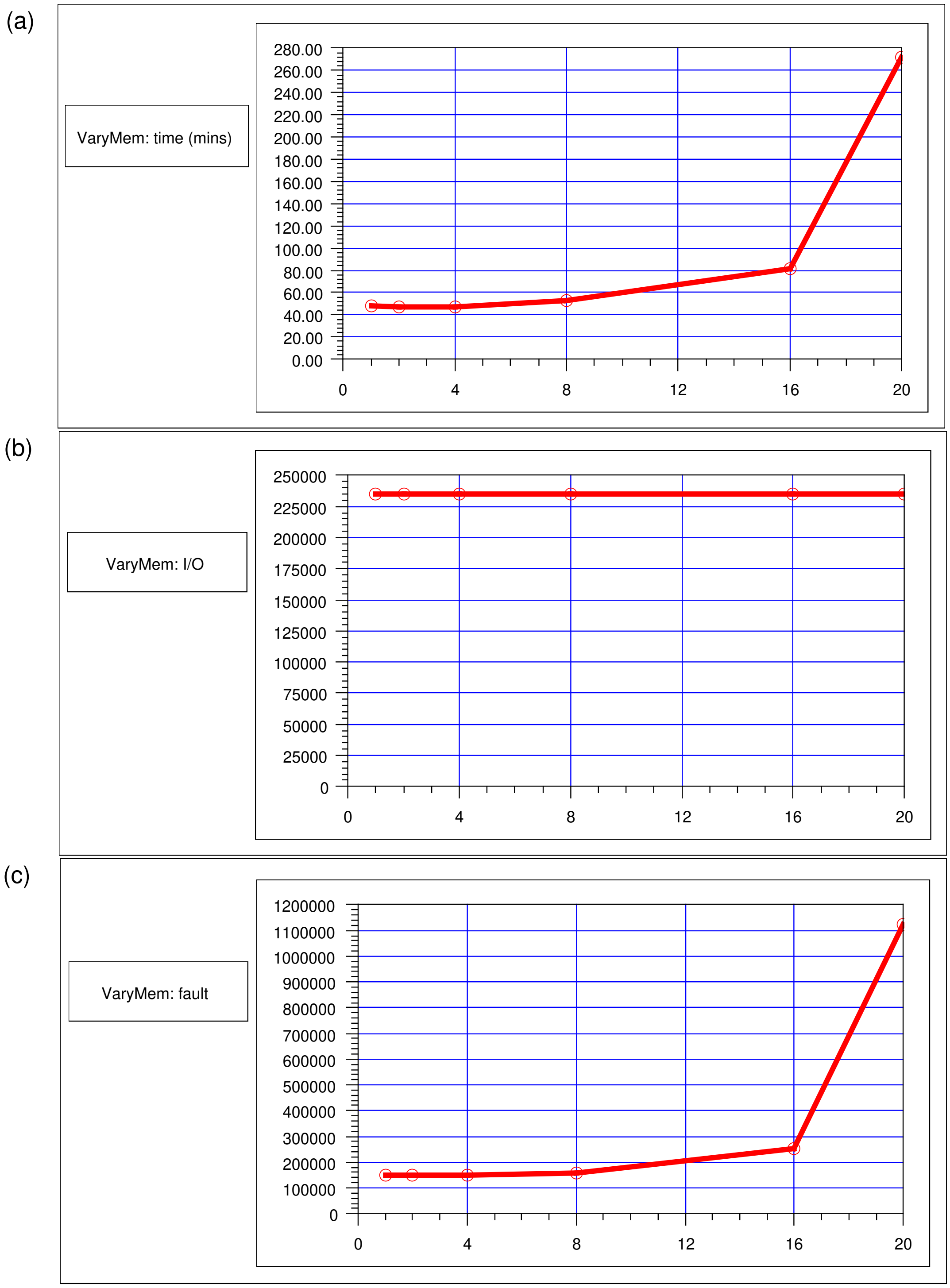
Now we discuss the explicit I/O (read/write operation) and the implicit I/O (page fault). Since the OS maintains a *system buffer cache* in the main memory and uses it to perform *read-ahead* and *delay write*, not all I/O operations requested by a process require physical (actual) I/O's. Thus there are two kinds of explicit I/O's: I/O's *requested* by a process, and I/O's *actually performed.* Similarly, the OS maintains in the main memory a list of disk pages that have been flushed and are waiting to be written out to the disk to reflect the modifications (the so-called *dirty pages*); page faults on these pages do not require physical I/O's (see any standard OS textbook, e.g., [33], for more details). Thus there are also two kinds of page faults, depending on whether or not physical I/O's are needed. Ideally, we would like to measure the number of requested I/O's, the number of performed I/O's, and the number of page faults that require physical I/O's.

We are surprised to find that the system does not fully support performance statistic information. For example, it is claimed that user commands `time` and `timex` give CPU and elapsed times as well as numbers of (performed) I/O's and of page faults (that require physical I/O's), but it turns out that at the time we conduct the experiments in the Brown Computer Science Department, only the information regarding CPU and elapsed times are available[3]. Fortunately, the system provides a `/proc` file system, which maintains for each process in the system a file (with the file name being the process ID) recording various information about the process. By using a system call in our program to retrieve the information in the `/proc` file system during the run time, we are able to obtain the number of page faults that require physical I/O's, and yet the numbers of page reads/writes actually performed by the process are still unavailable. Therefore, for explicit I/O's we only measure the number of requested I/O's: In our implementation, each `read` or `write` system call transfers one page of data, and we keep track of the number of times these two system calls are executed. In summary, we measure the number of requested I/O's for the explicit I/O's, and also the number of page faults that require physical I/O's (referred to as just *page faults* for short in the following) for the implicit I/O's.

We implement the algorithms so that the page size and the amount of main memory used by the programs can be parameterized. More specifically, `page_size` is a parameter throughout all four algorithms, while `MM_used` is a parameter in the external merge sort (used in the preprocessing phase of `Distribution`, `B-Tree` and `234-Tree`; recall from Sections 2.1 and 2.2), as well as a parameter in the sweeping phase of `Distribution`. We remark that by our implementation of B-tree where each node is a page, the main memory used by `B-Tree` during the sweeping phase is at most 8 pages (one for input buffer, one for output buffer, and at most 6 for nodes retained when inserting/deleting an item). Also, in the sweeping phase of `234-Tree` and in both phases of `234-Tree-Core`, since we let the OS handle the page faults assuming an infinite-size virtual memory (recall from Section 2.1), the amount of main memory used depends on the system status in the run time and is not controlled by the `MM_used` parameter.

We set the `page_size` parameter to 4Kb, to be consistent with the system we are using. As for the `MM_used` parameter, it is surprising that the main memory size available for use is typically much smaller than what we thought. When we run `Distribution` on `data-long` of 1.5×10^6 segments with various values of `MM_used` (see Fig. 6), in theory we would expect that using more main memory results in a better performance according to the I/O cost bound $O(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B} + \frac{K}{B})$ (see Table 1), but the experiments show that using 4Mb gives the best performance (average running time 47.64 minutes), and using 20Mb gives a significantly worse performance (average running time 271.97 minutes)! Going from 1Mb to 4Mb, the number of requested I/O's slightly decreases, and the number of page faults increases slightly yet negligibly; the net effect is to make the running time decrease slightly (see Fig. 6). Going from 4Mb to 20Mb, the number of requested I/O's again decreases only slightly, but the number of page faults increases significantly, thus resulting in a much worse performance (see Fig. 6). To investigate, we use the `top` user command to display the information about the processes. We see that the "real" main memory size in the system configuration is only 26Mb rather than 32Mb, and that there are a lot of daemon processes, although sleeping, still having parts of their images occupying the main memory space. Also, the OS seems to put some process-loading constraint so that for large-space processes their images

[3]This was confirmed by the system administrator Peter Galvin [22] and Prof. Tom Doeppner [18] who taught OS in the Department at the time we conducted the experiments.



* X-axis: size of the main memory used (Mb)

Figure 6: Running `Distribution` on data set `data-long` of 1.5×10^6 segments with various sizes of the main memory used: (a) average running times in minutes; (b) exact numbers of requested I/O's; (c) average numbers of page faults requiring physical I/O's (rounded to the nearest integers).

(including their text, data, and stack portions) are never fully loaded into the main memory. The process-loading behavior is decided by the OS and the user has no control over it. In the following, all the algorithms are running with the `MM_used` parameter set to 4Mb. Recall that this parameter has no effect on the sweeping phase of `B-tree` and of `234-Tree`, and also no effect on both phases of `234-Tree-Core`.

4 Analysis of the Experimental Results

Algorithms `Distribution`, `B-Tree`, `234-Tree`, and `234-Tree-Core` have been executed on data sets `data-short`, `data-long`, and `data-rect`, with data sizes ranging from 250 thousand segments to 2.5 million segments. In our implementation, each segment is of 24 bytes, and thus each page holds at most 170 segments ($B = 170$). Also, $M/B = 1024$ (recall that `MM_used` = 4Mb and `page_size` = 4Kb), and each internal node of our B-tree has 73 to 146 children (except that there is no lower bound on the number of children for the root node). While running time and number of page faults depend on the system status in the run time and may differ between runs of the same example, the number of I/O's requested by a process is always the same since our algorithms are deterministic. We run each example three times, and take the average on the running times and on the numbers of page faults. We find that the variation of running times among runs is at most 5%, and more importantly, these differences among runs do not affect the performance ranking of the four algorithms, that is, a faster algorithm always runs faster than a slower algorithm. Figures 7–9 show the corresponding values of (a) average running times, (b) exact numbers of requested I/O's, and (c) average numbers of page faults (rounded to the nearest integers), of the four algorithms running on the three data sets. We refer to Table 1 for the theoretical properties of the four algorithms, and to Table 3 for the properties of the three data sets.

Our experimental results show that while the performance of the three variations of plane sweep depends heavily on the average number of vertical overlaps, the performance of distribution sweep is both steady and efficient. Also, distribution sweep does not require a large amount of main memory to perform well: using 4Mb is enough. We make more detailed observations and analyses as follows:

- `234-Tree-Core` performs the best for small input ($N = 250 \times 10^3$) in all three data sets (see Figs. 7–9), but as input size grows, the performance becomes considerably worse, and up to $N = 10^6$ its running times are already out of comparison. Recall from Table 1 the algorithmic difference between `234-Tree-Core` and `234-Tree`. This shows that internal sorting assuming an infinite-size virtual memory performs much worse than external sorting when I/O becomes an issue. Moreover, the I/O issue is much more severe in sorting than in sweeping, since in sorting all segments are involved, while in sweeping only those segments "cut" by the sweep line are involved in the computation.
- Consider data set `data-short` (see Fig. 7). Excluding `234-Tree-Core`, the performance of the remaining algorithms is the *opposite* to our expectation from Table 1: `234-Tree` always runs the fastest, `B-Tree` the second, and `Distribution` the third. This is because the average number of vertical overlaps is only $\frac{1}{4}\sqrt{N}$ (see Table 3), and thus up to 2.5 million segments the entire data structure of `234-Tree` during sweeping can still fit into the main memory. Also, `Distribution` performs two sortings, while the others perform only one sorting. Notice again that for this data set although I/O is not an issue in sweeping (`234-Tree` performs the best), it is indeed a severe issue in sorting (`234-Tree-Core` performs significantly worse than all the others).

  Also worth noting is that although `B-Tree` performs far more requested I/O's than `Distribution`,

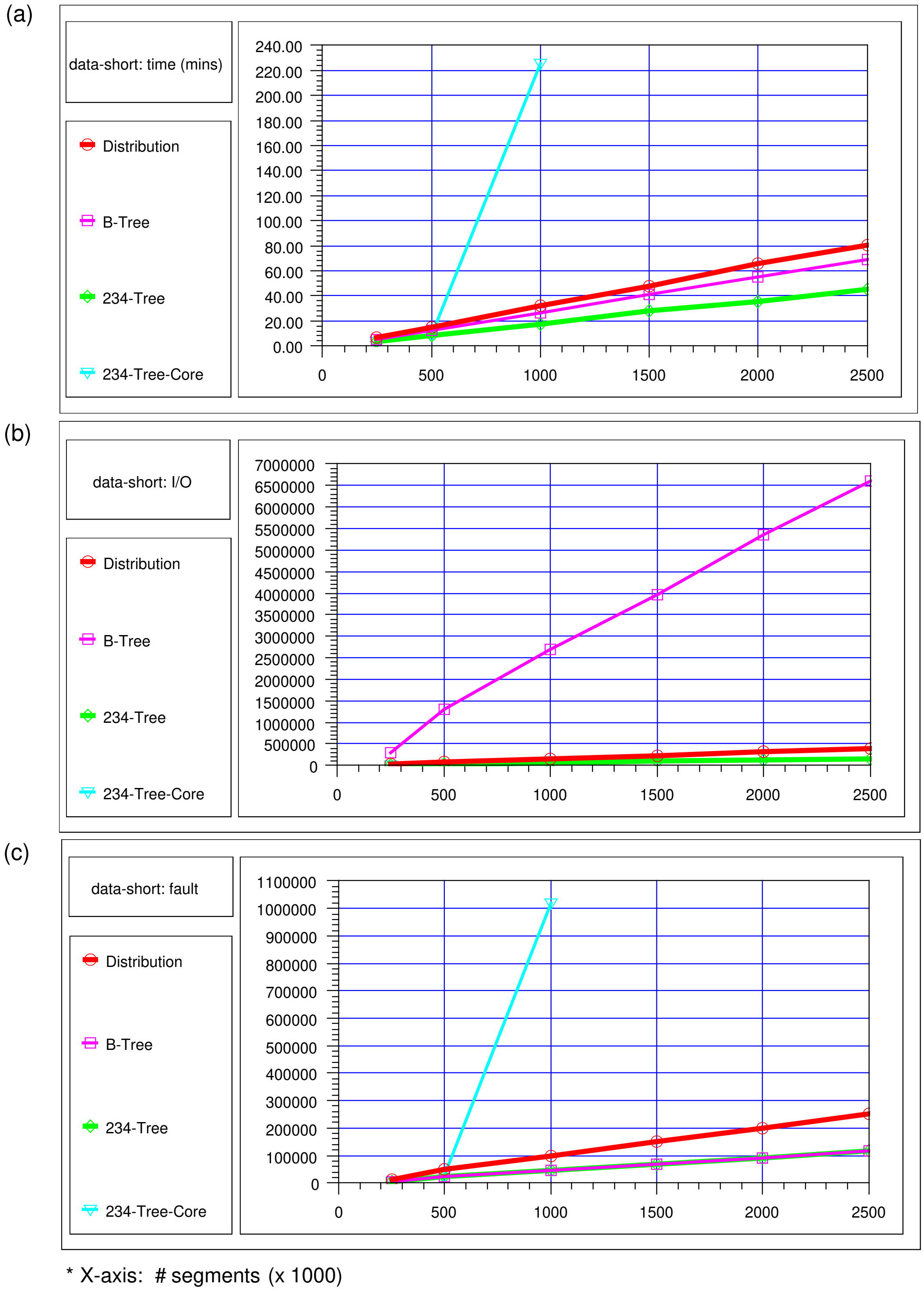


Figure 7: The results for the algorithms running on data set `data-short`: (a) average running times in minutes; (b) exact numbers of requested I/O's; (c) average numbers of page faults requiring physical I/O's (rounded to the nearest integers). We run `234-Tree-Core` only up to $N = 10^6$ since at this point it already takes time much longer than the others even at $N = 2.5 \times 10^6$.

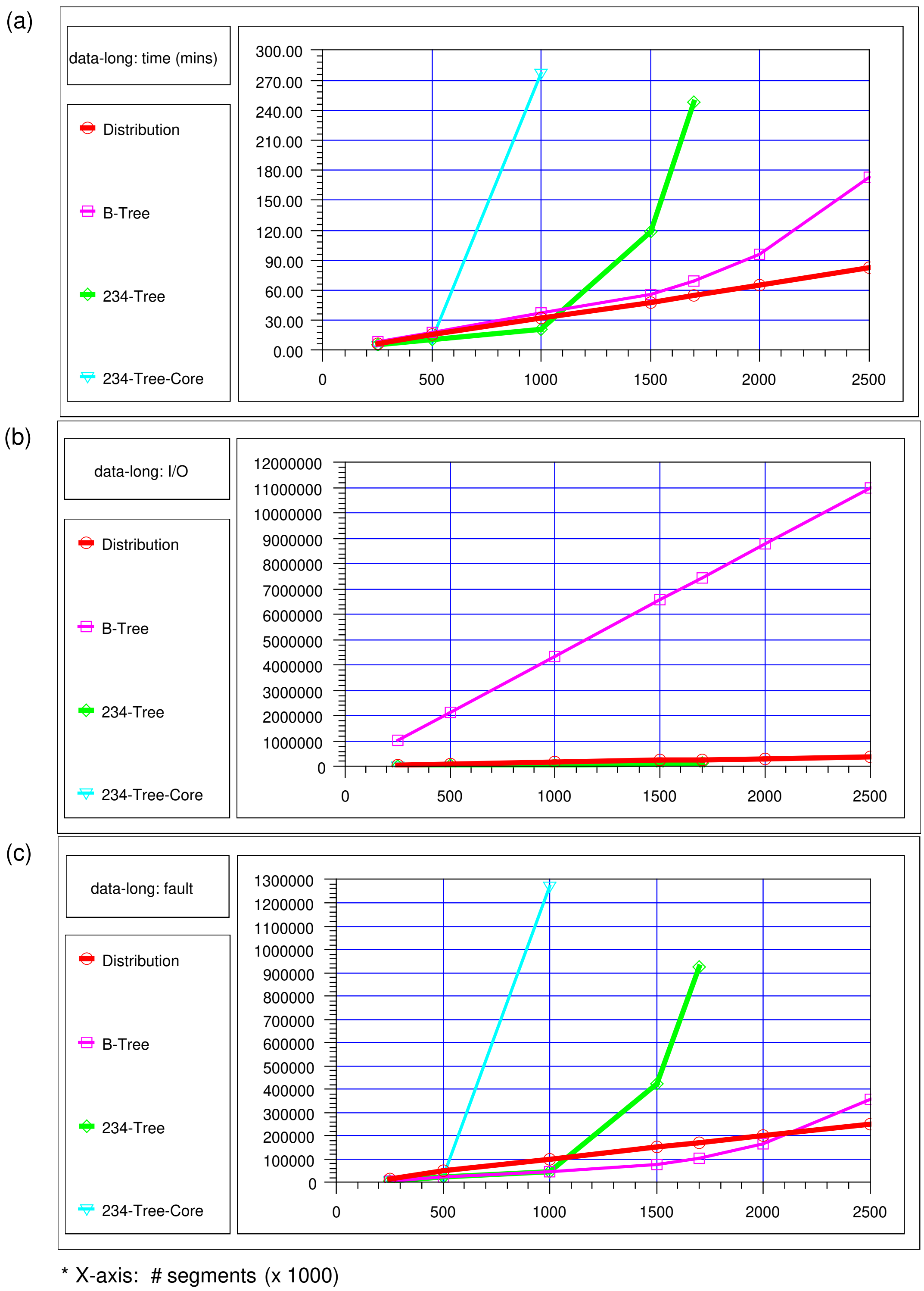


Figure 8: The results for the algorithms running on data set `data-long`: (a) average running times in minutes; (b) exact numbers of requested I/O's; (c) average numbers of page faults requiring physical I/O's (rounded to the nearest integers). We run `234-Tree-Core` only up to $N = 10^6$ and `234-Tree` only up to $N = 1.7 \times 10^6$ since at these points they already take times much longer than the others even at $N = 2.5 \times 10^6$.

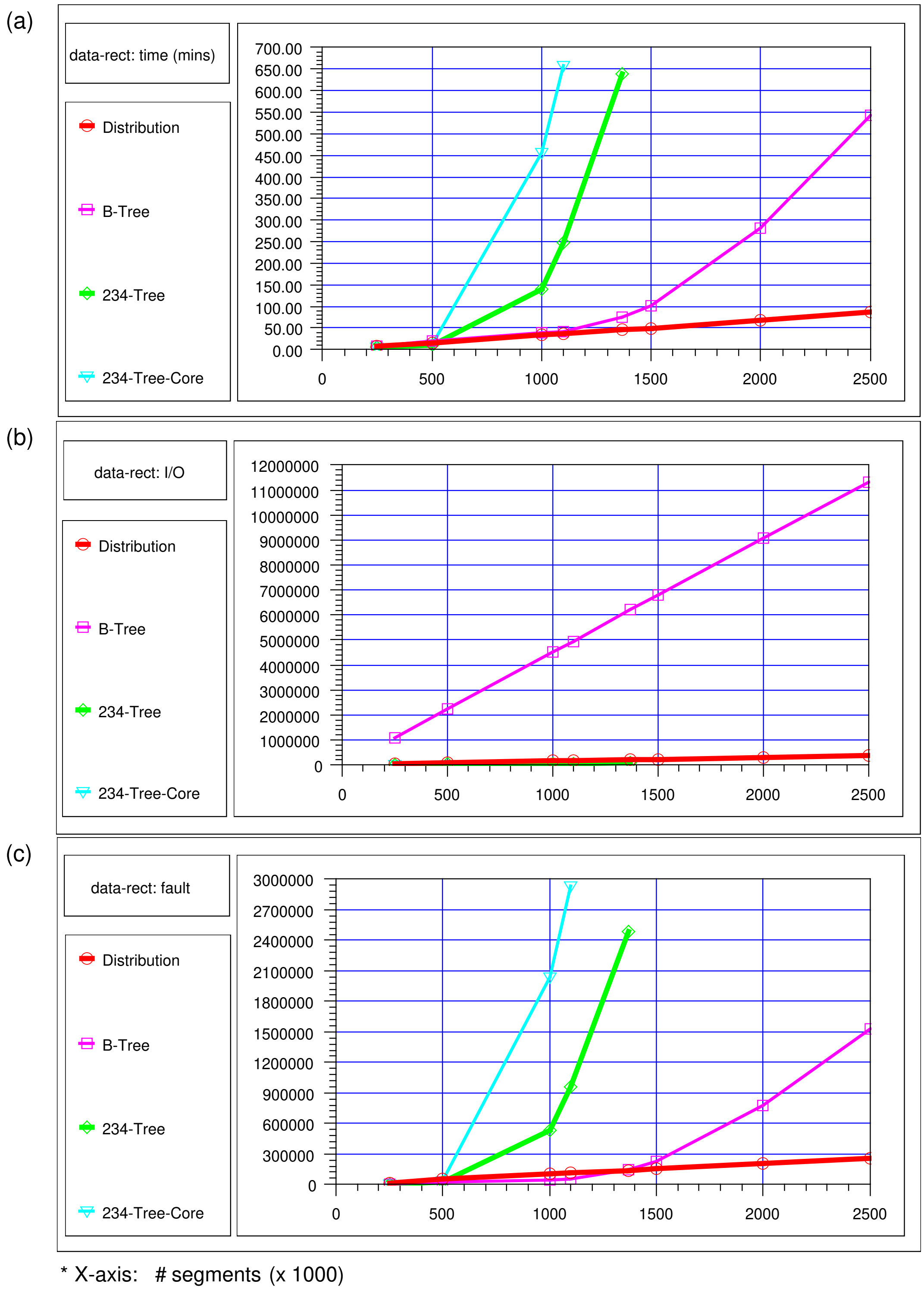


Figure 9: The results for the algorithms running on data set `data-rect`: (a) average running times in minutes; (b) exact numbers of requested I/O's; (c) average numbers of page faults requiring physical I/O's (rounded to the nearest integers). We run `234-Tree-Core` only up to $N = 1.1 \times 10^6$ and `234-Tree` only up to $N = 1.37 \times 10^6$ since at these points they already take times much longer than the others even at $N = 2.5 \times 10^6$.

their running times are comparable, with `B-Tree` a little bit faster (see Fig. 7(a)(b)). This is because not all requested I/O's need physical I/O's: if a read request is issued on a page already residing in the system buffer cache, then no disk read is needed; if two write requests are issued on the same page and the second request occurs before the first request is actually executed, then the first disk write is skipped. It is conceivable that the I/O's requested by `B-Tree` have a better locality so that many of them are cheap. This also shows that *simulations* counting only the numbers of I/O's requested by the algorithms are sometimes *not* enough to reflect the actual running times. Therefore, actual *experiments* running on *real machines* to measure the actual running times are necessary when we want to evaluate the practical I/O performance of the algorithms.

- For data set `data-long`, the I/O issue begins to play an important role (recall from Table 3 that the average number of vertical overlaps is $\frac{1}{8}N$), and the performance of the four algorithms is consistent with their theoretical properties from Table 1: `Distribution` the fastest, `B-Tree` the second, `234-Tree` the third, followed by `234-Tree-Core` (see Fig. 8). More specifically, `Distribution` runs much faster than all the others for $N \geq 1.5 \times 10^6$. The running times and the numbers of requested I/O's of `B-Tree` are still more or less linear, and are always worse than those of `Distribution`. The running times and numbers of page faults for both `234-Tree` and `234-Tree-Core` become out of comparison when the I/O issue shows up: `234-Tree` at $N \geq 1.7 \times 10^6$, and `234-Tree-Core` even earlier, at $N \geq 10^6$. This shows that when I/O becomes an issue, the performance of algorithms assuming an infinite-size virtual memory is very undesirable.
- For data set `data-rec`, the average number of vertical overlaps is $\frac{1}{4.8}N$ (see Table 3), and thus the I/O issue becomes even more important. The performance of the four algorithms is again consistent with their theoretical properties from Table 1, and their running times differ significantly (see Fig. 9). For example, for a moderate data set size $N = 1.37 \times 10^6$ and on average, `Distribution` runs for 45.29 minutes, `B-Tree` runs for 74.54 minutes, but `234-Tree` runs for more than 10.5 hours. Also, for $N = 2.5 \times 10^6$, `Distribution` always runs for less than 1.5 hours, but `B-Tree` always runs for more than 8.5 hours. This shows that the more important the I/O issue is, the more advantage we get from `Distribution`.
- For all three data sets, `234-Tree` and `234-Tree-Core` always have small numbers of requested I/O's (see Figs. 7(b), 8(b), and 9(b)). This is because in the sweeping phase both of them only perform `read` for reading the input once and `write` for writing the output once, and all other I/O activities are page faults caused by the assumption of an infinite-size virtual memory. Also, `Distribution` always has much less requested I/O's than `B-Tree` (see Figs. 7(b), 8(b), and 9(b)). Recall that the I/O cost bounds for `Distribution` and `B-Tree` are $O(\frac{N}{B}\log_{\frac{M}{B}}\frac{N}{B} + \frac{K}{B})$ and $O(N\log_B\frac{N}{B} + \frac{K}{B})$, respectively (see Table 1). With the parameter `MM_used` of the main memory size set to 4Mb, the two logarithmic terms in these bounds are almost the same, and it is the $\frac{1}{B}$ term that makes the difference significant.
- Page faults seem to be a much more effective factor to the running times than requested I/O's (see Figs. 7–9). This is because some requested I/O's, as mentioned before, do not need physical I/O's and are thus much cheaper, but all page faults measured here are those requiring physical I/O's. The large difference between the effectiveness of page faults and of requested I/O's to the running times shows that there are many requested I/O's that are cheap. This again shows that measuring only the numbers of requested I/O's is not enough to reflect the actual running times. We hope that in addition to numbers of requested I/O's and of page faults, the numbers of I/O's requested and *actually performed* can also be available

by a better system support in the future.

Finally, we discuss the question of which algorithm to use in practice when I/O is potentially an issue. First, for the sorting problem, as we have already seen, external sorting should always be the choice over internal sorting assuming an infinite-size virtual memory, regardless the average number of vertical overlaps. This rules out `234-Tree-Core` from competition. We next consider the sweeping problem. Recall that `Distribution` performs significantly better than all the others for `data-long` and `data-rect` (see Figs. 8–9), especially when N gets larger. As for `data-short`, although `Distribution` performs worse than `234-Tree` and `B-Tree`, their running times actually do not differ too much (see Fig. 7). In fact, for $N = 2.5 \times 10^6$, the running times of `Distribution` for all three data sets are always ranging from 80 to 87 minutes, while the running times of the others vary considerably (about 69 minutes to more than 8.5 hours for `B-Tree`, and 45 minutes to much more than 11 hours, possibly 36 hours by linear extrapolation in Fig. 9(a), for `234-Tree`). This shows that the performance of `Distribution` is both steady and efficient. Even in the case where the entire data structure can fit into the main memory, the overhead of `Distribution` is still acceptable. We conclude that `Distribution` should always be the choice among the four algorithms, and that we should choose `234-Tree` only when we know in advance that the average number of vertical overlaps in the data set is small enough so that the data structure is guaranteed to fit into the main memory.

5 Conclusion

We have presented an experimental study comparing the performance of four algorithms for the orthogonal segment intersection problem. The observed behavior of the algorithms shows that while the performance of the three variations of plane sweep depends heavily on the average number of vertical overlaps and can be very poor, the performance of distribution sweep is both steady and efficient. Also, distribution sweep does not require a large amount of main memory to perform well. In addition, it is shown that the performance resulting from the strategy of letting the OS handle page faults and not explicitly considering the I/O cost is very undesirable. We conclude that whenever there is a potential I/O issue, distribution sweep should always be the choice among the four algorithms for the orthogonal segment intersection problem, and for other problems we should always design and use I/O-efficient algorithms to obtain a good performance guarantee.

Appendix

Proof of Theorem 1. Recall that K is a random variable for the number of horizontal-vertical intersecting pairs in `data-short`, and our goal is to compute $E[K]$. We can express K by $K = \sum_{1 \le i,j \le N,\ i \ne j} K_{ij}$, where for $1 \le i, j \le N$, $i \ne j$, K_{ij} is a 0-1 random variable defined by

$$K_{ij} = \begin{cases} 1 & \text{if segment } s_i \text{ is horizontal, segment } s_j \text{ is vertical, and } s_i \cap s_j \neq \emptyset \\ 0 & \text{otherwise.} \end{cases}$$

Clearly, we have

$$\Pr\{K_{ij} = 1\} = \Pr\{s_i \text{ is horizontal and } s_j \text{ is vertical}\} \cdot \Pr\{\text{horizontal } s_i \cap \text{vertical } s_j \neq \emptyset\}.$$

The first term is $\frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$, so let us focus on computing the second term. In the following, s_i is horizontal and s_j is vertical. For s_i and s_j to intersect, the y-coordinate of s_i is contained in the

y-interval spanned by s_j (denoted by $y(s_i) \in I_y(s_j)$), and similarly for the x-dimension. Define P_0 to be the conditional probability $\Pr\{y(s_i) \in I_y(s_j) \mid y(b(s_j)) = t, |s_j| = h\}$, where $y(b(s_j))$ is the y-coordinate of the bottom endpoint of s_j, and $|s_j|$ is the length of s_j. For any $t \in [0, N - \sqrt{N}]$ and $h \in [0, \sqrt{N}]$ we have $P_0 = \Pr\{y(s_i) \in [t, t+h] \mid y(b(s_j)) = t, |s_j| = h\} = \frac{h}{N}$, since $y(s_i)$ is uniformly distributed over $[0, N]$. Let $f(t)$ be the probability density function of $y(b(s_j))$. Since $P_0 = \frac{h}{N}$ is independent of t, we have $\Pr\{y(s_i) \in I_y(s_j) \mid |s_j| = h\} = \int_{-\infty}^{+\infty} P_0 \cdot f(t)\, dt = P_0 \cdot \int_{-\infty}^{+\infty} f(t)\, dt = \frac{h}{N}$. Moreover, $\Pr\{y(s_i) \in I_y(s_j) \mid |s_j| = h, |s_i| = w\} = \Pr\{y(s_i) \in I_y(s_j) \mid |s_j| = h\} = \frac{h}{N}$, since events $\{y(s_i) \in I_y(s_j)\}$ and $\{|s_i| = w\}$ are conditionally independent given $|s_j| = h$. By a similar argument, we have $\Pr\{x(s_j) \in I_x(s_i) \mid |s_j| = h, |s_i| = w\} = \frac{w}{N}$, and therefore $\Pr\{s_i \cap s_j \neq \emptyset \mid |s_j| = h, |s_i| = w\} = \frac{h}{N} \cdot \frac{w}{N}$. Now $|s_j|$ is uniformly distributed over $[0, \sqrt{N}]$, namely,

$$g(h) = \begin{cases} \frac{1}{\sqrt{N}} & \text{if } 0 \leq h \leq \sqrt{N} \\ 0 & \text{otherwise} \end{cases}$$

is the probability density function of $|s_j|$; similarly for $|s_i|$. Thus

$$\Pr\{\text{horizontal } s_i \cap \text{vertical } s_j \neq \emptyset\} = \int_{-\infty}^{+\infty} \int_{-\infty}^{+\infty} \frac{h}{N} \cdot \frac{w}{N} \cdot g(h) \cdot g(w)\, dh\, dw = \frac{1}{4N}.$$

Therefore we have $\Pr\{K_{ij} = 1\} = \frac{1}{16N}$. It follows that $E[K] = N(N-1) \cdot E[K_{ij}] = \frac{1}{16}(N-1)$. □

Proof of Theorem 2. Recall that V is a random variable for the number of vertical segments in `data-short` cut by the horizontal sweep line l for an event, and our goal is to compute $E[V]$. We can express V by $V = V_1 + V_2 + \cdots + V_N$, where for $1 \leq j \leq N$, V_j is a 0-1 random variable defined by

$$V_j = \begin{cases} 1 & \text{if segment } s_j \text{ is vertical and } s_j \cap l \neq \emptyset \\ 0 & \text{otherwise.} \end{cases}$$

Clearly, we have

$$\Pr\{V_j = 1\} = \Pr\{s_j \text{ is vertical}\} \cdot \Pr\{\text{vertical } s_j \cap l \neq \emptyset\}. \tag{1}$$

The first term is $\frac{1}{2}$, so let us compute the second term. In the following s_j is vertical. If $y(l)$ were uniformly distributed over $[0, N]$, then letting l play the role of a horizontal segment s_i and applying the previous method for $E[K]$, we would have $\Pr\{s_j \cap l \neq \emptyset \mid |s_j| = h\} = \Pr\{y(l) \in I_y(s_j) \mid |s_j| = h\} = \frac{h}{N}$, and $\Pr\{V_j = 1\}$ could be computed accordingly. But this is not the case.

Recall that the positions of l depend on the positions of the events. Let the probability density function of $y(l)$ be defined by

$$k(u) = \begin{cases} k_1(u) & \text{if } 0 \leq u < \sqrt{N} \\ k_2(u) & \text{if } \sqrt{N} \leq u \leq N - \sqrt{N} \\ k_3(u) & \text{if } N - \sqrt{N} < u \leq N \\ 0 & \text{else.} \end{cases}$$

If there were only horizontal segments, then we would have $k_1(u) = k_2(u) = k_3(u) = \frac{1}{N}$. Now consider including also the vertical segments. There is no bottom endpoint q with $y(q) \in (N - \sqrt{N}, N]$, so $k_3(u) < \frac{1}{N}$. Also, for a top endpoint q to have $y(q) = u$ for some $u \in [0, \sqrt{N})$ (e.g., $u = \frac{1}{2}\sqrt{N}$), the length of the corresponding vertical segment must not exceed u, while there is no such restriction on a top endpoint with y-coordinate in $[\sqrt{N}, N - \sqrt{N}]$. Therefore we have $k_1(u) < \frac{1}{N}$ and $k_2(u) \geq \frac{1}{N}$. To compute $\Pr\{V_j = 1\}$, we proceed in a different way.

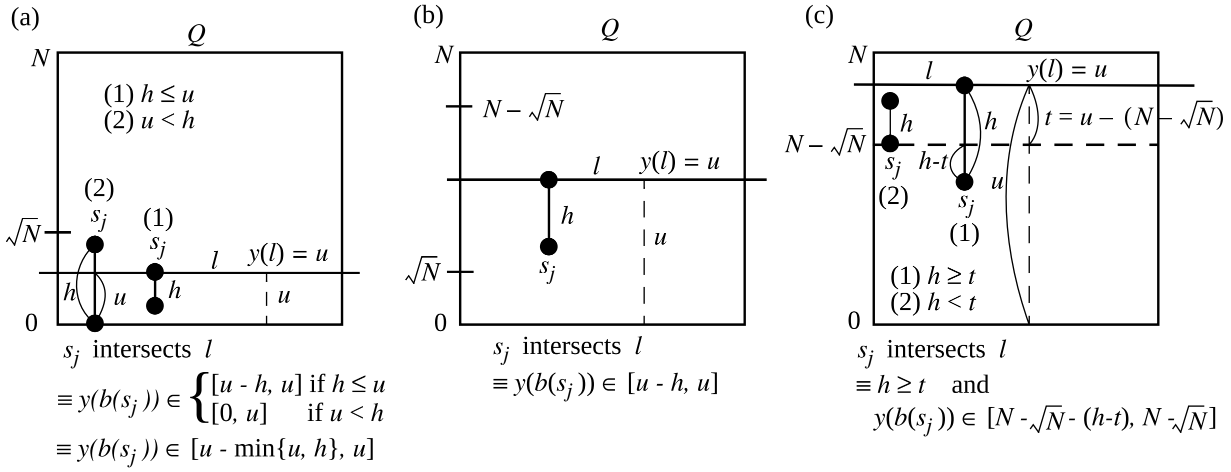


Figure 10: Computing $\Pr\{s_j \cap l \neq \emptyset \mid y(l) = u, |s_j| = h\}$ for `data-short`: (a) $0 \leq u < \sqrt{N}$; (b) $\sqrt{N} \leq u \leq N - \sqrt{N}$; (c) $N - \sqrt{N} < u \leq N$.

Define P to be the conditional probability $\Pr\{s_j \cap l \neq \emptyset \mid y(l) = u, |s_j| = h\}$. By the fact that $y(b(s_j))$ is uniformly distributed over $[0, N - \sqrt{N}]$ and the analysis shown in Fig. 10, we have that $P = \frac{h}{N-\sqrt{N}}$ for $\sqrt{N} \leq u \leq N - \sqrt{N}$ (see Fig. 10(b)) and $P = \frac{\min\{u,h\}}{N-\sqrt{N}}$ for $0 \leq u < \sqrt{N}$ (see Fig. 10(a)). As for $N - \sqrt{N} < u \leq N$ (see Fig. 10(c)), note that $P = \frac{h-t}{N-\sqrt{N}}$ if $h \geq t$ and $P = 0$ if $h < t$, and thus $P = \frac{len}{N-\sqrt{N}}$ where $len = \max\{h - t, 0\} = \max\{h - u + N - \sqrt{N}, 0\}$. In summary, we have

$$P \stackrel{\text{def}}{=} \Pr\{s_j \cap l \neq \emptyset \mid y(l) = u, |s_j| = h\} = \begin{cases} \frac{\min\{u,h\}}{N-\sqrt{N}} \stackrel{\text{def}}{=} P_1 & \text{if } 0 \leq u < \sqrt{N} \\ \frac{h}{N-\sqrt{N}} \stackrel{\text{def}}{=} P_2 & \text{if } \sqrt{N} \leq u \leq N - \sqrt{N} \\ \frac{\max\{h-u+N-\sqrt{N},0\}}{N-\sqrt{N}} \stackrel{\text{def}}{=} P_3 & \text{if } N - \sqrt{N} < u \leq N \\ 0 & \text{else.} \end{cases}$$

Observe that $\min\{u, h\} \leq h$ and that $\max\{h - u + N - \sqrt{N}, 0\} \leq h$ for $u > N - \sqrt{N}$, so we have $P \leq \frac{h}{N-\sqrt{N}}$ for $u \in [0, N]$. Recall that the probability density function of $|s_j|$ is $g(h)$ given before. Let $\widetilde{P}$ denote the probability $\Pr\{\text{vertical } s_j \cap l \neq \emptyset\}$. Then

$$\begin{aligned} \widetilde{P} &= \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} P \cdot k(u)\, g(h)\, du\, dh \\ &\leq \int_0^{\sqrt{N}}\int_0^N \frac{h}{N-\sqrt{N}}\, k(u)\, \frac{1}{\sqrt{N}}\, du\, dh = \int_0^{\sqrt{N}} \frac{h}{N-\sqrt{N}}\, \frac{1}{\sqrt{N}} \left(\int_0^N k(u)\, du\right) dh = \frac{1}{2}\, \frac{1}{\sqrt{N}-1}, \end{aligned}$$

where the last equality follows from the fact that $\int_0^N k(u)du = 1$. Also,

$$\begin{aligned} \widetilde{P} &= \int_{-\infty}^{+\infty}\int_{-\infty}^{+\infty} P \cdot k(u)\, g(h)\, dh\, du \\ &= \int_0^{\sqrt{N}}\int_{-\infty}^{+\infty} P_1 \cdot k_1(u)\, g(h)\, dh\, du + \int_{\sqrt{N}}^{N-\sqrt{N}}\int_{-\infty}^{+\infty} P_2 \cdot k_2(u)\, g(h)\, dh\, du \\ &\quad + \int_{N-\sqrt{N}}^{N}\int_{-\infty}^{+\infty} P_3 \cdot k_3(u)\, g(h)\, dh\, du \end{aligned}$$

$$\geq \int_{\sqrt{N}}^{N-\sqrt{N}} \int_{0}^{\sqrt{N}} \frac{h}{N-\sqrt{N}} \frac{1}{N} \frac{1}{\sqrt{N}} \, dh \, du = \frac{1}{2} \frac{1}{\sqrt{N}-1} \frac{\sqrt{N}-2}{\sqrt{N}},$$

where the inequality follows from that fact that the first and the third additive terms are both non-negative and that $k_2(u) \geq \frac{1}{N}$. Now we have lower and upper bounds for $\widetilde{P}$, and recall from equation (1) that $\Pr\{V_j = 1\} = \frac{1}{2} \cdot \widetilde{P}$. It follows that $E[V] = N \cdot E[V_j] \leq \frac{1}{4}\sqrt{N}(\frac{\sqrt{N}}{\sqrt{N}-1}) = \frac{1}{4}\sqrt{N} + \frac{1}{4} + \frac{1}{4(\sqrt{N}-1)}$ and also $E[V] \geq \frac{1}{4}\sqrt{N}(\frac{\sqrt{N}-2}{\sqrt{N}-1}) = \frac{1}{4}\sqrt{N} - \frac{1}{4} - \frac{1}{4(\sqrt{N}-1)}$, that is, $E[V] = \frac{1}{4}\sqrt{N} + O(1) \sim \frac{1}{4}\sqrt{N}$. $\square$

Acknowledgements

I would like to thank Roberto Tamassia for a lot of stimulating discussions, and Tom Doeppner and Peter Galvin for providing me useful information about the computing systems in the Computer Science Department at Brown University.

References

[1] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.

[2] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 334–345, 1995.

[3] L. Arge. The I/O-complexity of ordered binary-decision diagram manipulation. In *Proc. Int. Symp. on Algorithms and Computation, LNCS 1004*, pages 82–91, 1995.

[4] L. Arge, M. Knudsen, and K. Larsen. A general lower bound on the I/O-complexity of comparison-based algorithms. In *Proc. Workshop on Algorithms and Data Structures, LNCS 709*, pages 83–94, 1993.

[5] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proc. European Symp. Algorithms, LNCS 979*, pages 295–310, 1995.

[6] R. Bayer and E. McCreight. Organization of large ordered indexes. *Acta Inform.*, 1:173–189, 1972.

[7] J. L. Bentley. Experiments on traveling salesman heuristics. In *Proc. 1st ACM-SIAM Sympos. Discrete Algorithms*, pages 91–99, 1990.

[8] J. L. Bentley. K-d trees for semidynamic point sets. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, pages 187–197, 1990.

[9] J. L. Bentley. Tools for experiments on algorithms. In *Proc. CMU 25th Anniversary Symp.*, 1990.

[10] J. L. Bentley. Fast algorithms for geometric traveling salesman problems. *ORSA J. Comput.*, 4(4):387–411, 1992.

[11] P. Callahan, M. T. Goodrich, and K. Ramaiyer. Topology B-trees and their applications. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 381–392, 1995.

[12] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995.

[13] D. Comer. The ubiquitous B-tree. *ACM Comput. Surv.*, 11:121–137, 1979.

[14] T. H. Cormen. *Virtual Memory for Data Parallel Computing.* PhD thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, 1992.

[15] T. H. Cormen. Fast permuting in disk arrays. *Journal of Parallel and Distributed Computing*, 17(1–2):41–57, Jan./Feb. 1993.

[16] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, Cambridge, Mass., 1990.

[17] T. H. Cormen, T. Sundquist, and L. F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. Technical Report PCS-TR94-223, Dartmouth College Dept. of Computer Science, July 1994.

[18] T. Doeppner. Personal communication, 1995.

[19] E. Feuerstein and A. Marchetti-Spaccamela. Memory paging for connectivity and path problems in graphs. In *Proc. Int. Symp. on Algorithms and Comp.*, 1993.

[20] P. G. Franciosa and M. Talamo. Orders, implicit k-sets representation and fast halfplane searching. In *Proc. Workshop on Orders, Algorithms and Applications (ORDAL'94)*, pages 117–127, 1994.

[21] G. N. Frederickson. A data structure for dynamically maintaining rooted trees. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 175–184, 1993.

[22] P. Galvin. Personal communication, 1995.

[23] M. T. Goodrich, M. H. Nodine, and J. S. Vitter. Blocking for external graph searching. In *Proc. ACM SIGACT-SIGMOD-SIGART Symp. on Principles of Database Sys.*, pages 222–232, 1993.

[24] M. T. Goodrich, J.-J. Tsay, D. E. Vengroff, and J. S. Vitter. External-memory computational geometry. In *IEEE Foundations of Comp. Sci.*, pages 714–723, 1993.

[25] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. In *Proc. ACM Symp. on Principles of Database Sys.*, pages 233–243, 1993.

[26] C. M. Kenyon-Mathieu and J. S. Vitter. The maximum size of dynamic data structures. *SIAM J. Comput.*, 20:807–823, 1991.

[27] M. H. Nodine and J. S. Vitter. Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 120–129, 1993.

[28] M. H. Nodine and J. S. Vitter. Paradigms for optimal sorting with multiple disks. In *Proc. of the 26th Hawaii Int. Conf. on Systems Sciences*, 1993.

[29] F. P. Preparata and M. I. Shamos. *Computational Geometry: an Introduction.* Springer-Verlag, New York, NY, 1985.

[30] S. Ramaswamy and P. C. Kanellakis. OODB indexing by class-division. In *Proc. ACM SIGMOD Internat. Conf. on Management of Data*, pages 139–150, 1995.

[31] S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. In *Proc. ACM Symp. on Principles of Database Sys.*, pages 25–35, 1994.

[32] C. Ruemmler and J. Wilkes. An introduction to disk drive modeling. *IEEE Computer*, 27(3):17–28, 1994.

[33] A. Silberschatz, J. Peterson, and P. Galvin. *Operating System Concepts.* Addison-Wesley,

MA, 1991.

[34] S. Subramanian and S. Ramaswamy. The P-range tree: A new data structure for range searching in secondary memory. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 378–387, 1995.

[35] J. D. Ullman and M. Yannakakis. The input/output complexity of transitive closure. *Annals of Mathematics and Artificial Intelligence*, 3:331–360, 1991.

[36] D. E. Vengroff. A transparent parallel I/O environment. In *Proc. 1994 DAGS Symposium on Parallel Computation*, 1994.

[37] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proc. IEEE Symp. on Parallel and Distributed Computing*, 1995.

[38] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two-level memories. *Algorithmica*, 12(2-3):110–147, 1994.

[39] B. Zhu. Further computational geometry in secondary memory. In *Proc. Int. Symp. on Algorithms and Computation*, 1994.