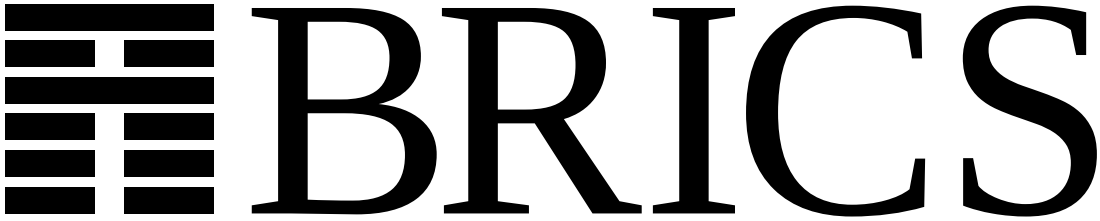


Basic Research in Computer Science

BRICS DS-96-3 L. Arge: Efficient External-Memory Data Structures and Applications

Efficient External-Memory Data Structures and Applications

Lars Arge

BRICS Dissertation Series **DS-96-3**

ISSN 1396-7002 **August 1996**

See back inner page for a list of recent publications in the BRICS Dissertation Series. Copies may be obtained by contacting:

BRICS
Department of Computer Science
University of Aarhus
Ny Munkegade, building 540
DK - 8000 Aarhus C
Denmark

Telephone: +45 8942 3360
Telefax: +45 8942 3255
Internet: BRICS@brics.dk

BRICS publications are in general accessible through WWW and anonymous FTP:

`http://www.brics.dk/`
`ftp ftp.brics.dk (cd pub/BRICS)`

Efficient External-Memory Data Structures and Applications

Lars Arge

Ph.D. Dissertation

Department of Computer Science
University of Aarhus
Denmark

Efficient External-Memory Data Structures and Applications

A Dissertation
Presented to the Faculty of Science
of the University of Aarhus
in Partial Fulfillment of the Requirements for the
Ph.D. Degree

by
Lars Arge
February 1996

Abstract

In this thesis we study the Input/Output (I/O) complexity of large-scale problems arising e.g. in the areas of database systems, geographic information systems, VLSI design systems and computer graphics, and design I/O-efficient algorithms for them.

A general theme in our work is to design I/O-efficient algorithms through the design of I/O-efficient data structures. One of our philosophies is to try to isolate all the I/O specific parts of an algorithm in the data structures, that is, to try to design I/O algorithms from internal memory algorithms by exchanging the data structures used in internal memory with their external memory counterparts. The results in the thesis include a technique for transforming an internal memory tree data structure into an external data structure which can be used in a *batched* dynamic setting, that is, a setting where we for example do not require that the result of a search operation is returned immediately. Using this technique we develop batched dynamic external versions of the (one-dimensional) range-tree and the segment-tree and we develop an external priority queue. Following our general philosophy we show how these structures can be used in standard internal memory sorting algorithms and algorithms for problems involving geometric objects. The latter has applications to VLSI design. Using the priority queue we improve upon known I/O algorithms for fundamental graph problems, and develop new efficient algorithms for the graph-related problem of ordered binary-decision diagram manipulation. Ordered binary-decision diagrams are the state-of-the-art data structure for boolean function manipulation and they are extensively used in large-scale applications like logic circuit verification.

Combining our batched dynamic segment tree with the novel technique of external-memory fractional cascading we develop I/O-efficient algorithms for a large number of geometric problems involving line segments in the plane, with applications to geographic informations systems. Such systems frequently handle huge amounts of spatial data and thus they require good use of external-memory techniques.

We also manage to use the ideas in the batched dynamic segment tree to develop "on-line" external data structures for a special case of two-dimensional range searching with applications to databases and constraint logic programming. We develop an on-line external version of the segment tree, which improves upon the previously best known such structure, and an optimal on-line external version of the interval tree. The last result settles an important open problem in databases and I/O algorithms. In order to develop these structure we use a novel balancing technique for search trees which can be regarded as weight-balancing of B-trees.

Finally, we develop a technique for transforming internal memory lower bounds to lower bounds in the I/O model, and we prove that our new ordered binary-decision diagram manipulation algorithms are asymptotically optimal among a class of algorithms that include all know manipulation algorithms.

To my parents
For their love, encouragement and support

Acknowledgments

A single conversation with a wise man is better than ten years of study
Chinese Proverb

The other day a colleague reminded me that I would soon have been studying for ten years at University of Aarhus. At first the thought was pretty frightful, but then I remembered all the wise and friendly women and men I had met during these years—they all deserve thanks.

My advisor Erik Meineche Schmidt deserves special thanks. His enthusiastic teaching style made me interested in doing research. He believed in my research capabilities even though I didn't spend all my time as an undergraduate reading books and studying for exams, and at the times when I didn't believe in myself he convinced me that it was perfectly normal. He pointed me in the right direction and then he entrusted me the freedom that made my study such an enjoyable and interesting time. Also, I have had a lot of fun with him drinking beer and discussing politics. In that respect I see a lot of myself in Erik.

Very special thanks go to Peter Bro Miltersen. He has taught me a lot about how to do research and many times he has helped me in formulating my results in a correct and precise way. Even though we have only known each other for half of the ten years, we have had at least ten years worth of fun together. I was especially happy when he married my friend through many years, Nina Bargisen. Now I get to play with their lovely daughter Emma—she is a lot of fun! It also seems that I will finally co-author a paper with him, something I'm very enthusiastic about. Thanks to the whole family.

Thanks also go to my friends and co-authors Mikael (Knud) Knudsen and Kirsten Larsen for making the writing of my first paper such a great experience. Also my friend and "personal programmer" Christian Lynbech deserves a lot of thanks. I would never have been able to write this thesis without his fantastic knowledge about LaTeX and Emacs and his willingness to tell me about the hacks at short notice. Finishing this thesis I hope to spend a lot more time making him loose in badminton and listening to Danish pop-music with him.

At the department a lot of other friends and colleagues contributed to all these fun years. In the algorithm group I'm very grateful to Sven Skyum for teaching me computational geometry and for listening to my problems, and to Dany Breslauer, Gerth S. Brodal, Devdatt Dubhashi, Gudmund S. Frandsen and Thore Husfeldt for lots of inspiring discussions. Outside the group thanks especially go to Ole Caprani, Allan Cheng, Bettina B. Clausen, Nils Klarlund, Mogens Nielsen, Michael I. Schwartzbach and Peter Ørbæk. Also the always helpful secretaries Susanne Brøndberg, Janne Damgaard, Marianne Iversen, Karen Møller, and Charlotte Nielsen deserve lots of thanks, and so does the rest of the support staff.

One thing that have made the ten years especially joyful is all the undergrads I have been teaching computer science over the years. Not only did they help in "keeping me young", but it was also a lot of fun. I don't think I would have survived without them. I won't try to

write all their names here, as I'm afraid to forget half of them. Thanks all of you!

However, one of them—Liv Hornekær—deserves very special thanks. I don't know if I would have been able to complete the work in this thesis if she hadn't been there. I'm especially happy that she joined me while I was at Duke University, and for the time we have spent on hiking and camping at exotic places around the world. Many hugs to her!

I would also like to thank all my other friends for helping me keep life in perspective by constantly reminding me that life is more than external-memory algorithms. All my roommates over the years have been indispensable in that regard, and Lone V. Andersen, Morten Brun, Claus Lund Jensen, Bettina Nøhr, Mette Terp, and Dorte Markussen have been especially persistent. Martha Vogdrup and all her various roommates also deserve special credit in that regard. Martha is always ready for a quick party and I hope we will have a lot of those in the years to come. I'm also very grateful for my years in the Student Council (Studenterrådet) which matured me a lot, and thus these years are a major reason why I managed successfully to finish this thesis.

During my Ph.D. study I was fortunate to visit Jeffrey S. Vitter at Duke University for eight months. The stay at Duke meant everything to me! Not only was it incredibly inspiring to be around other researchers interested in external-memory algorithms, but in general the inspiring atmosphere at Duke made me work hard and have fun while doing so. Working with Jeff taught me a lot and we obtained a lot of nice results. I learned a lot from the change in environment both scientifically and personally, and I would like to thank Jeff and Duke University for giving me the opportunity to do so. During my stay at Duke I made a lot of other friends who made my stay interesting and fun. I would especially like to thank P. Krishnan, for being such a great officemate, Pankaj Agarwal, Rakesh Barve, Cathie Caimano, Eddie and Jeannie Grove, T. M. Murali, and Darren Vengroff. The eight months passed way too quickly!

I would also like to thank my thesis evaluation committee—Joan Boyar, Mogens Nielsen, and Roberto Tamassia—for their nice comments and suggestions.

Also my brother Thomas and my sister Maria deserve thanks. However, I'm by far most grateful to my parents for their love, encouragement and support, which gave me all the opportunities.

Lars Arge
Århus, February/August 1996

The work in this thesis was financially supported by University of Aarhus, Aarhus University Research Foundation, the ESPRIT II Basic Research Actions Program of the EC under contract No. 7141 (project ALCOM II), the Danish Research Academy and BRICS (Acronym for Basic Research in Computer Science, a Center of the Danish National Research Foundation).

Contents

Chapter 1

Introduction

The difference in speed between modern CPU and disk technologies is analogous to the difference in speed in sharpening a pencil by using a sharpener on one's desk or by taking an airplane to the other side of the world and using a sharpener on someone else's desk

D. Cormer[1]

Traditionally when designing computer programs people have focused on the minimization of the internal computation time and ignored the time spent on Input/Output (I/O). Theoretically one of the most commonly used machine models when designing algorithms is the Random Access Machine (RAM) (see e.g. [6, 125]). One main feature of the RAM model is that its memory consists of an (infinite) array, and that any entry in the array can be accessed at the same (constant) cost. Also in practice most programmers conceptually write programs on a machine model like the RAM. In an UNIX environment for example the programmer thinks of the machine as consisting of a processor and a huge ("infinite") memory where the contents of each memory cell can be accessed at the same cost (Figure 1.1). The task of moving data in and out of the limited main memory is then entrusted to the operating system. However, in practice there is a huge difference in access time of fast internal memory and slower external memory such as disks. While typical access time of main memory is measured in nano seconds, a typical access time of a disk is on the order of milli seconds [45]. So roughly speaking there is a factor of a million in difference in the access time of internal and external memory, and therefore the assumption that every memory cell can be accessed at the same cost is questionable, to say the least!

In many modern large-scale applications the communication between internal and external memory, and not the internal computation time, is actually the bottleneck in the computation. Examples of large-scale applications can e.g. be found in database systems [79, 110], spatial databases and geographic information systems (GIS) [52, 70, 84, 114, 124], VLSI verification [17], constraint logic programming [78, 79], computer graphics and virtual reality systems [60, 114], computational biology [135], physics and geophysics [47, 129] and in meteorology [47]. The amount of data manipulated in such applications is too large to fit in main memory and must reside on disk, hence the I/O communication can become a very severe bottleneck. A good example is NASA's EOS project GIS system [52], which is expected to manipulate petabytes (thousands of terabytes, or millions of gigabytes) of data!

The effect of the I/O bottleneck is getting more pronounced as internal computation gets faster, and especially as parallel computing gains popularity [107]. Currently, technological

[1] Cited in [34]

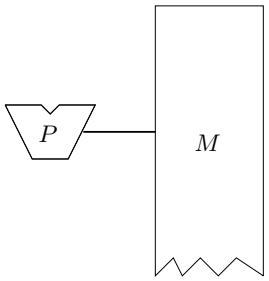


Figure 1.1: A RAM-like machine model.

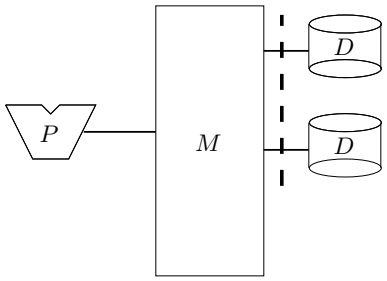


Figure 1.2: A more realistic model.

advances are increasing CPU speeds at an annual rate of 40–60% while disk transfer rates are only increasing by 7–10% annually [113]. Internal memory sizes are also increasing, but not nearly fast enough to meet the needs of important large-scale applications.

Modern operating systems try to minimize the effect of the I/O bottleneck by using sophisticated paging and prefetching strategies in order to assure that data is present in internal memory when it is accessed. However, these strategies are general purpose in nature and therefore they cannot take full advantage of the properties of a specific problem. Instead we could hope to design more efficient algorithms by explicitly considering the I/O communication when designing algorithms for specific problems. This could e.g. be done by designing algorithms for a model where the memory system consists of a main memory of limited size and a number of external memory devices (Figure 1.2), and where the memory access time depends on the type of memory accessed. Algorithms designed for such a model are often called *external-memory (or I/O) algorithms*.

In this thesis we study the I/O complexity of problems from several of the above mentioned areas and design efficient external-memory algorithms for them.

1.1 The Parallel Disk Model

Accurately modeling memory and disk systems is a complex task [113]. The primary feature of disks that we want to model is their extremely long access time relative to that of internal memory. In order to amortize the access time over a large amount of data, typical disks read or write large blocks of contiguous data at once. Therefore we use a theoretical model defined in [5] with the following parameters:

$$\begin{aligned} N &= \text{number of elements in the problem instance;} \\ M &= \text{number of elements that can fit into internal memory;} \\ B &= \text{number of elements per disk block;} \end{aligned}$$

where $M < N$ and $1 \leq B \leq M/2$.

In order to study the performance of external-memory algorithms, we use the standard notion of I/O complexity [5]. We define an *Input/Output (I/O) operation* to be the process of simultaneously reading or writing a block of B contiguous data elements to or from the disk. As I/O communication is our primary concern, we define the I/O complexity of an algorithm simply to be the number of I/Os it performs. Thus the I/O complexity of reading all of the input data is equal to N/B. Internal computation is free in the model. Depending on the

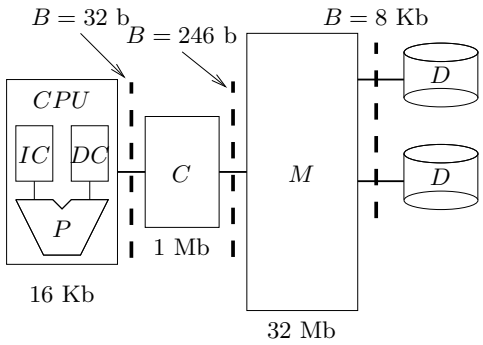


Figure 1.3: A "real" machine with typical memory and block sizes [45].

size of the data elements, typical values for workstations and file servers in production today are on the order of $M = 10^6$ or 10^7 and $B = 10^3$. Large-scale problem instances can be in the range $N = 10^{10}$ to $N = 10^{12}$.

An increasingly popular approach to further increase the throughput of the I/O system is to use a number of disks in parallel [63, 65, 133]. Several authors have considered an extension of the above model with a parameter D denoting the number of disks in the system [19, 98, 99, 100, 133]. In *the parallel disk model* [133] one can read or write one block from each of the D disks simultaneously in one I/O. The number of disks D range up to 10^2 in current disk arrays.

Our I/O model corresponds to the one shown in Figure 1.2, where we only count the number of blocks of B elements moved across the dashed line. Of course the model is mainly designed for theoretical considerations and is thus very simple in comparison with a real system. For example one cannot always ignore internal computation time and one could try to model more accurately the fact that (in single user systems at least) reading a block from disk in most cases decreases the cost of reading the block succeeding it. Also today the memory of a real machine is typically made up of not only two but several levels of memory (e.g. on-chip data and instruction cache, secondary cache, main memory and disk) between which data are moved in blocks (Figure 1.3). The memory in such a hierarchy gets larger and slower the further away from the processor one gets, but as the access time of the disk is extremely large compared to that of all the other memories in the hierarchy one can in most practical situations restrict the attention to the two level case. Thus the parallel disk model captures the most essential parameters of the I/O systems in use today, and theoretical results in the model can help to gain valuable insight. This is supported by experimental results which show that implementing algorithms designed for the model can lead to significant runtime improvements in practice [40, 41, 126, 129].

Finally, it should be mentioned that several authors have considered extended theoretical models that try to model the hierarchical nature of the memory of real machines [1, 2, 3, 4, 7, 77, 116, 131, 132, 134], but such models quickly become theoretically very complicated due to the large number of parameters. Therefore only very basic problems like sorting have been considered in these models.

1.2 Outline of the Thesis

The rest of this thesis consists of two parts. The first part contains a survey of important results in the area of I/O algorithms and I/O complexity, with special emphasis on the authors results. The second part contains the authors results in the form of five papers. These papers are:

[Buffer] *The Buffer Tree: A New Technique for Optimal I/O-Algorithms*

[GIS] *External-Memory Algorithms for Processing Line Segments in Geographic Information Systems* (co-authored with Vengroff and Vitter)

[Interval] *Optimal Dynamic Interval Management in External Memory* (co-authored with Vitter)

[OBDD] *The I/O-Complexity of Ordered Binary-Decision Diagram Manipulation*

[LowB] *A General Lower Bound on the I/O-Complexity of Comparison-based Algorithms* (co-authored with Knudsen and Larsen)

Extended abstract versions of the papers have appeared in [11, 15, 16, 12, 13]. In the survey we refer to the papers as indicated above.

The survey part of the thesis is divided into four chapters. In Chapter 2 we discuss basic paradigms for designing I/O-efficient algorithms and I/O complexity of fundamental problems. This leads to a survey of external-memory results in computational geometry (Chapter 3) and in graph algorithms (Chapter 4). Finally, Chapter 5 contains conclusions.

Throughout the first part we give a high level presentation of ideas and results without giving too many details. The reader is referred to the papers for details. We shall restrict the discussion to the one-disk model ($D = 1$). Many of the results can be modified to work in the D-disk model, however, for at least one prominent example this is not true [GIS].

Part I

Survey of Important I/O Results

Chapter 2

Fundamental Problems and Paradigms

640K ought to be enough for anybody
B. Gates[1]

In this chapter we first consider basic paradigms for designing I/O-efficient algorithms and then address the question of lower bounds in the I/O model. Early work on I/O complexity concentrated on sorting and sorting related problems. Initial theoretical work was done by Floyd [59] and by Hong and Kung [72] who studied matrix transposition and fast Fourier transformation in restricted I/O models. The general I/O model was introduced by Aggarwal and Vitter [5] and the notion of parallel disks was introduced by Vitter and Shriver [133]. The latter papers also deal with fundamental problems such as permutation, sorting and matrix transposition. The problem of implementing various classes of permutations has been addressed in [47, 48, 50]. More recently researchers have moved on to more specialized problems in the computational geometry [11, 15, 34, 40, 67, 74, 79, 110, 121, 130, 137], graph [12, 40, 42, 97] and string areas [44, 56, 57].

As already mentioned the number of I/O operations needed to read the entire input is N/B and for convenience we call this quotient n. We use the term *scanning* to describe the fundamental primitive of reading (or writing) all elements in a set stored contiguously in external memory by reading (or writing) the blocks of the set in a sequential manner in $O(n)$ I/Os. Furthermore, we say that an algorithm uses a linear number of I/O operations if it uses $O(n)$ I/Os. Similarly, we introduce $m = M/B$ which is the number of blocks that fits in internal memory. Aggarwal and Vitter [5] showed that the number of I/O operations needed to sort N elements is $\Omega(n \log_m n)$, which is then the external-memory equivalent of the well-known $\Omega(N \log_2 N)$ internal-memory bound.[2] Taking a closer look at the bound for typical values of B and M reveals that because of the large base of the logarithm, $\log_m n$ is less than 3 or 4 for all realistic values of N and m. Thus in practice the important term is the B-term in the denominator of the $O(n \log_m n) = O(\frac{N}{B} \log_m n)$ bound. As typical values of B are measured in thousands, an improvement from an $\Omega(N)$ (which we will see is the worst case I/O performance of many internal-memory algorithms) to the sorting bound $O(n \log_m n)$ can be really significant in practice.

[1] In 1981.

[2] We define $\log_m n = \max\{1, \log n / \log m\}$.

In Section 2.1 we describe the basic paradigms for designing efficient I/O algorithms. Section 2.2 then contains a further discussion of the I/O complexity of fundamental problems like sorting and permuting.

2.1 Basic Paradigms for Designing I/O-efficient Algorithms

Originally Aggarwal and Vitter [5] presented two basic "paradigms" for designing I/O-efficient algorithms; the *merging* and the *distribution* paradigms. In Section 2.1.1 and 2.1.2 we demonstrate the main idea in these paradigms by showing how to sort N elements in the optimal number of I/Os. Another important paradigm is to construct I/O-efficient versions of commonly used data structures. This enables the transformation of efficient internal-memory algorithms to efficient I/O-algorithms by exchanging the data structures used in the internal algorithms with the external data structures. This approach has the extra benefit of isolating the I/O-specific parts of an algorithm in the data structures. We call the paradigm the *data structuring* paradigm and in Section 2.1.3 we illustrate it by way of the so called *buffer tree* designed in [Buffer]. As demonstrated in Chapter 3 and 4, I/O-efficient data structures turn out to be a very powerful tool in the development of efficient I/O algorithms.

2.1.1 Merge Sort

External merge sort is a generalization of internal merge sort. First N/M $(= n/m)$ sorted "runs" are formed by repeatedly filling up the internal memory, sorting the elements, and writing them back to disk. This requires $O(n)$ I/Os. Next m runs are continually merged together into a longer sorted run, until we end up with one sorted run containing all the elements—Figure 2.1. The crucial property is that we can merge m runs together in a linear number of I/Os. To do so we simply load a block from each of the runs and collect and output the B smallest elements. We continue this process until we have processed all elements in all runs, loading a new block from a run every time a block becomes empty. Since there are $O(\log_m n/m) = O(\log_m n)$ levels in the merge process, and since we only use $O(n)$ I/O operations on each level, we obtain the optimal $O(n \log_m n)$ algorithm.

One example of the use of the merging paradigm can be found in [LowB] where we develop an optimal algorithm for the problem of removing duplicates from a multiset, an important problem in e.g. relational database systems where the amount of data manipulated indeed can get very large.

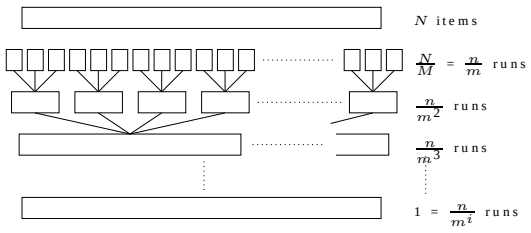


Figure 2.1: Merge sort.

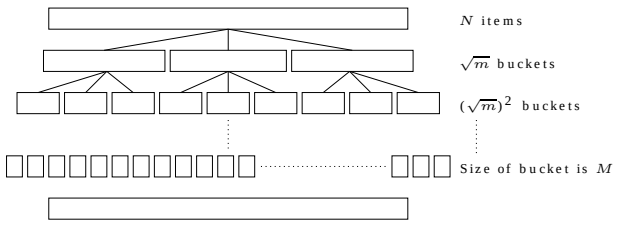


Figure 2.2: Distribution sort.

2.1.2 Distribution Sort

External distribution sort is in a sense the reverse of merge sort and the external-memory equivalent of quick sort. Like in merge sort the distribution sort algorithm consists of a number of levels each using a linear number of I/Os. However, instead of repeatedly merging m run together, we repeatedly distribute the elements in a "bucket" into m smaller "buckets" of approximately equal size. All elements in the first of these smaller buckets are smaller than all the elements in the second bucket and so on. The process continues until the elements in a bucket fit in internal memory, in which case the bucket is sorted using an internal-memory sorting algorithm. The sorted sequence is then obtained by appending the small sorted buckets.

Like m-way merge, m-way distribution can also be performed in a linear number of I/Os, by just keeping a block in internal memory for each of the buckets we are distributing elements into. However, for "technical" reasons the distribution is only done $\sqrt{m}$-way—see Figure 2.2. Otherwise one cannot produce the partition elements needed to distribute the elements in a bucket in a linear number of I/Os [5]. Actually the partition elements are produced using a rather complicated algorithm based on the linear time internal-memory selection algorithm [29]. However, even with the smaller distribution factor the distribution sort algorithm still runs in the asymptotically optimal $O(n \log_{\sqrt{m}} n) = O(n \log_m n)$ I/O operations.

The distribution sort paradigm is definitely the most powerful of the two paradigms. As a first application we can mention the optimal algorithm developed in [LowB] for the problem of finding the mode—the most frequently occurring element—of a multiset.

Both the distribution sort and merge sort algorithms demonstrate two of the most fundamental and useful features of the I/O-model, which is used repeatedly when designing I/O algorithms. First the fact that we can do m-way merging or distribution in a linear number of I/O operations, and secondly that we can solve a complicated problem in a linear number of I/Os *if* it fits in internal memory. In the two algorithms the sorting of a memory load is an example of the last feature, which is also connected with what is normally referred to as "locality of reference"—one should try to work on data in chunks of the block (or internal memory) size, and do as much work as possible on data once it is loaded into internal memory.

2.1.3 Buffer Tree Sort

We can sort N elements in internal memory in $O(N \log_2 N)$ time using what one could call *tree sort* or *priority queue sort*, using a search tree or a priority queue in the obvious way. The standard search tree structure for external memory is the B-tree [21, 51, 82]. On this structure insert, delete, deletemin and search operations can be performed in $O(\log_B n)$ I/Os. Thus using this structure in the standard algorithms results in $O(N \log_B n)$ I/O sorting algorithms. This result is non optimal as the logarithm is base B rather than m, but more importantly it is a factor of B away from optimal because of the N rather than n in front of the logarithm. As mentioned above this factor can be very significant in practice. The inefficiency is a consequence of the fact that the B-tree is designed to be used in an "on-line" setting, where queries should be answered immediately and within a good worst-case number of I/Os, and thus updates and queries are handled on an individual basis. This way one is not able to take full advantage of the large internal memory. Actually, using a decision tree like argument as in [79], one can easily show that the search bound is indeed optimal in such an "on-

line" setting. This fact seems to have been the main reason for many researchers [5, 42, 67] to develop I/O algorithms, and special techniques for designing I/O algorithms, instead of trying to develop and use external data structures.

However in "off-line" environments where we are only interested in the overall I/O use of a series of operations on the involved data structures, and where we are willing to relax the demands on the search operations, we could hope to develop data structures on which a series of N operations could be performed in $O(n \log_m n)$ I/Os in total. This is precisely what is done in [Buffer]. Below we sketch the basic tree structure developed using what is called the *buffer tree* technique. The structure can be used in the normal tree sort algorithm, and we also sketch how the structure can be used to develop an I/O-efficient external priority queue. In later sections we will discuss extensions and applications of these structures as well as other structures developed using the general technique.

Basically the buffer tree is a tree structure with fan-out $\Theta(m)$ built on top of n leaves, each containing B of the elements stored in the structure. Thus the tree has height $O(\log_m n)$—refer to Figure 2.3. A *buffer* of size $\Theta(m)$ blocks is assigned to each internal node and operations on the structure are done in a "lazy" manner. On insertion of an element we do not right away search all the way down the tree to find the place among the leaves to insert the element. Instead, we wait until we have collected a block of insertions, and then we insert this block in the buffer of the root (which is stored on disk). When a buffer "runs full" its element are "pushed" one level down to buffers on the next level. We call this a *buffer-emptying process* and it is basically performed as a distribution step, i.e. by loading the elements into internal memory, sorting them, and writing them back to the appropriate buffers on the next level. If the buffer of any of the nodes on the next level becomes full the buffer-emptying process is applied recursively. The main point is now that we can perform the buffer-emptying process in $O(m)$ I/Os, because the elements in a buffer fit in memory and because the fan-out of the structure is $\Theta(m)$. The latter means that at most $O(m)$ I/Os is used on distributing non-full blocks. As $\Theta(m)$ blocks are pushed one level down the tree in a buffer-emptying process, every *block* is touched at most a constant number of times on each of the $O(\log_m n)$ levels of the tree. Thus it is easy to show, using an amortization argument, that a series of N insertions can be done in $O(n \log_m n)$ I/Os. This corresponds to performing one insertion in $O(\frac{\log_m n}{B})$ I/Os *amortized*. Deletions can be handled using similar ideas. We also need to consider rebalancing of the tree, that is, what to do when an insert or delete reaches a leaf. In [Buffer] we show that by using an (a, b)-tree [73] as the basic tree structure, rebalancing can also be handled in the mentioned I/O bound.

In order to use the structure in a sorting algorithm we need an operation that reports all

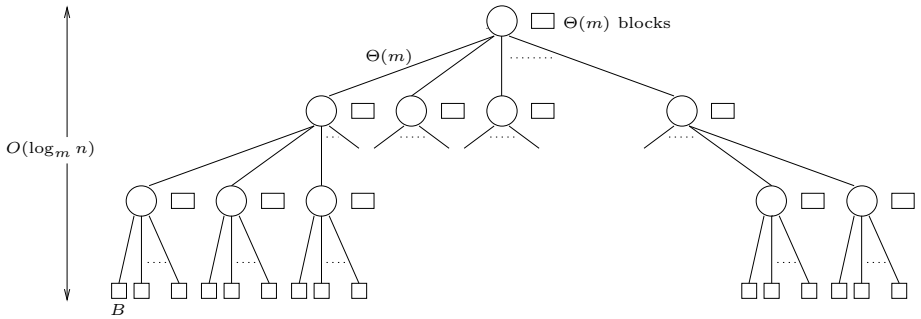


Figure 2.3: Buffer tree.

the elements in the tree in sorted order. To do so we first empty all buffers in the tree. As the number of internal nodes is $O(n/m)$ this can easily be done in $O(n)$ I/Os. After this all the elements are in the leaves and can be reported in sorted order using a simple scan.

Using the buffer idea we have obtained a structure with the operations needed to sort N elements optimally with *precisely* the same tree sort algorithm as can be used in internal memory. This means that we have reached our goal of hiding the I/O-specific parts of the algorithm in the data structure. In [Buffer] we show how to use the buffer tree structure to develop an external-memory priority queue. Normally, we can use a search-tree structure to implement a priority queue because we know that the smallest element in a search-tree is in the leftmost leaf. The same strategy can be used to implement an external priority queue based on the buffer tree. However, in the buffer tree we cannot be sure that the smallest element is in the leftmost leaf, since there can be smaller elements in the buffers of the nodes on the leftmost path. There is, however, a simple strategy for performing a deletemin operation in the desired amortized I/O bound. When we want to perform a deletemin operation, we simply do a buffer-emptying process on all nodes on the path from the root to the leftmost leaf. This requires $O(m \cdot \log_m n)$ I/Os. Now we can be sure not only that the leftmost leaf consists of the B smallest elements, but also that the $\Theta(m \cdot B) = \Theta(M)$ smallest elements in the tree are in the $\Theta(m)$ leftmost leaves. Thus we can delete these elements and keep them in internal memory. In this way we can answer the next $\Theta(M)$ deletemin operations without performing any I/Os. Of course we then have to check insertions against the minimal elements in internal memory, but that can also be done without doing any I/Os. Thus our deletemin operation uses $O(\frac{\log_m n}{B})$ I/Os amortized. Apart the ability to sort optimally with yet another well-known algorithm, the external priority queue will come in handy later on, especially in Chapter 4 where we discuss external-memory graph algorithms.

2.2 Lower Bounds and Complexity of Fundamental Problems

As already mentioned Aggarwal and Vitter [5] proved that the number of I/O operations needed to sort N elements is $\Omega(n \log_m n)$. Furthermore, they showed that the number of I/Os needed to rearrange N elements according to a given permutation is $\Omega(\min\{N, n \log_m n\})$. For simplicity one normally writes $\mathrm{sort}(N)$ and $\mathrm{perm}(N)$ for $n \log_m n$ and $\min\{N, n \log_m n\}$, respectively, suppressing m (and thus B) and only mentioning the main parameter N.

The basic idea in the proof of the $\mathrm{perm}(N)$ permutation lower bound is to count how many permutations can be generated with a given number of I/O operations and compare this number to $N!$ As permutation is a special case of sorting the lower bound obtained this way also applies to sorting. As already discussed $\log_m n$ is very small in practice, which means that the sorting term in the permutation bound will be smaller than N in all realistic cases. In this case $\mathrm{perm}(N) = \mathrm{sort}(N)$ and the problem of permutation is as hard as the more general problem of sorting. Thus the dominant component of sorting in this case is the routing of the elements, and not the determination of their order. For extremely small values of M and B the permutation bound becomes N, and the optimal permutation algorithm is equivalent to moving the elements one at a time using one I/O on each element. However, assuming the *comparison model* in internal memory and using an adversary argument, Aggarwal and Vitter proved that the $\Omega(n \log_m n)$ lower bound still holds for sorting in this case. Thus for small values of M and B the specific knowledge of what goes where makes the problem of permuting easier than sorting. That one in most cases needs to sort to perform a given permutation

is one of the important features that distinguishes the I/O model from an internal memory model. As we will discuss in Chapter 4 this becomes very prominent when considering graph problems.

2.2.1 Transforming Lower Bounds

In [LowB] we assume the comparison model in internal memory for *all* values of M and B. This allows us to design a general method for transforming an I/O algorithm into an internal-memory algorithm, such that the number of comparisons used to solve a given problem instance is bounded by some function of the number of I/Os used to solve the same problem instance. Using this general construction internal-memory comparison lower bounds can be transformed into I/O lower bounds. More precisely we prove that given an I/O algorithm that solves a given problem using $I/O(\overline{x})$ I/Os on the input instance $\overline{x}$, there exists an ordinary internal-memory comparison algorithm that uses no more than $n \log B + I/O(\overline{x}) \cdot T_{\text{merge}}(M - B, B)$ comparisons on $\overline{x}$. $T_{\text{merge}}(s, t)$ denotes the number of comparisons needed to merge two sorted lists of size s and t.

The main idea in the proof is the following. We define what we call an I/O-tree to be an ordinary decision-tree extended with I/O-nodes. We define the tree in such a way that removal of the I/O-nodes results in an ordinary comparison tree solving the same problem. The proof then corresponds to constructing an algorithm that transforms a given I/O-tree to another I/O-tree, such that the number of comparisons on every path from the root to a leaf is bounded by a function of the number of I/O-nodes on the corresponding path in the original I/O-tree. The main idea in the transformation is to establish the total order of the elements in internal memory after every I/O-operation. Thus we modify the original I/O-tree by replacing the comparisons between I/O-nodes with a tree that performs all comparisons needed to obtain a total ordering of the elements in internal memory. We apply this transformation inductively from the topmost "comparison-subtree" and downwards. The main property we use to obtain the total order after each I/O is that we already know the total order of the $M - B$ elements in main memory which are unaffected by the I/O. Thus we can obtain the total order by sorting the B new elements and merging them into the old elements. The construction results in a new I/O-tree which is equivalent to the old one. Furthermore, the number of comparison-nodes on a path in the new tree is bounded by the number of I/O-nodes on the path multiplied with the number of comparisons required to do the merge. Removing the I/O-nodes now results in a comparison-tree with the desired relation to the original I/O-tree.

Using the transformation technique we immediately obtain an alternative proof of the sorting I/O lower bound in the comparison I/O model as well as lower bounds on a number of set problems. We also obtain lower bounds on the duplicate removal and mode multiset problems mentioned previously, which show that the algorithms developed in [LowB] using the merge and distribution paradigms are asymptotically optimal.

2.3 Bibliographic Notes and Summary of our Contributions

The merge and distribution paradigms were introduced by Aggarwal and Vitter [5]. In [Buffer] we introduce the data structuring paradigm. We develop the buffer tree and the external priority queue and show how they can be used in the standard internal-memory algorithms for sorting. This follows our general philosophy of trying to "hide" the I/O-specific part of an algorithm in the data structures. The obtained sorting algorithms are optimal and are also the first algorithms that do not require all elements to be present by the start of the algorithms.

Aggarwal and Vitter [5] proved $\Omega(n \log_m n)$ and $\Omega(\min\{N, n \log_m n\})$ I/O lower bounds on the problems of sorting and permuting, respectively. In the sorting bound they assumed the comparison model in internal memory when B and M are extremely small. In [LowB] we assume the comparison model for all values of B and M and prove a general connection between the comparison complexity and the I/O complexity of a problem. We use this connection to obtain lower bounds on sorting as well as on a number of other set problems. We also prove lower bounds on two multisets problems. We use the distribution and merge paradigms to develop non-trivial algorithms for these problems in order to show that the lower bounds are tight.

Chapter 3

Computational Geometry Problems

Where there is matter, there is geometry
J. Kepler

In Chapter 1 we gave examples of large-scale applications almost all of which involve manipulation of geometric data. In this section we consider external-memory algorithms for computational geometry problems, and in order to do so we define two additional parameters:

$$\begin{aligned} K &= \text{number of queries in the problem instance;} \\ T &= \text{number of elements in the problem solution.} \end{aligned}$$

In analogy with the definition of n and m we define $k = K/B$ and $t = T/B$ to be respectively the number of query blocks and number of solution element blocks.

In internal memory one can prove what might be called sorting lower bounds $O(N \log_2 N + T)$ on a large number of important computational geometry problems. The corresponding bound $O(n \log_m n + t)$ can be obtained for the external versions of the problems either by redoing standard proofs [14, 67], or by using the conversion result in [LowB] as discussed in Section 2.2.1. Note however that the latter is only interesting for problems that can be solved using comparisons only.

Computational geometry problems in external memory were first considered by Goodrich et al. [67], who developed a number of techniques for designing I/O-efficient algorithms for such problems. They used their techniques to develop I/O algorithms for a large number of important problems. In internal memory the *plane-sweep* paradigm is a very powerful technique for designing computational geometry algorithms, and in [67] an external-memory version of this technique called *distribution sweeping* is developed. As the name suggests the technique relies on the distribution paradigm. In [67] distribution sweeping is used to design optimal algorithms for a large number of problems including orthogonal line segment intersection, all nearest neighbors, pairwise rectangle intersection and batched range queries. In [Buffer] we show how the data structuring paradigm can also be use on computational geometry problems. We show how data structures based on the buffer tree can be used in the standard internal-memory plane-sweep algorithm for a number of the mentioned problems, namely orthogonal line segment intersection, pairwise rectangle intersection and batched range queries. In [67] two techniques called *batched construction of persistent B-trees* and *batched filtering* are also discussed, and external-memory algorithms for convex-hull construction in

two and three dimensions are developed using the distribution paradigm. Some external computational geometry results are also reported in [62, 137]. In [GIS] we design efficient I/O algorithms for a large number of problems involving line segments in the plane by combining the ideas of distribution sweeping, batched filtering, buffer trees and a new technique, which can be regarded as an external-memory version of *fractional cascading*. Most of these problems have important applications in GIS systems, which provide a rich source of large-scale problems.

A number of researchers have considered the design of worst-case efficient external-memory "on-line" data structures, mainly for the range searching problem. While B-trees [21, 51, 82] efficiently support range searching in one dimension they are inefficient in higher dimensions. In [27, 74, 79, 110, 121, 130] data structures for (special cases of) two and three dimensional range searching are developed. In [Interval] we develop an optimal on-line data structure for the equally important problem of dynamic interval management. This problem is a special case of two-dimensional range searching with applications in database systems. Range searching is also considered in [104, 119, 120] where the problem of maintaining range trees in external memory is considered. However, the model used in this work is different from ours. In [34] an external on-line version of the topology tree is developed and this structure is used to obtain structures for a number of dynamic problems, including approximate nearest neighbor searching and closest pair maintenance.

We divide our discussion of computational geometry results in three main parts; in Section 3.1 we discuss distribution sweeping and external plane-sweeping using buffer tree structures through the solutions to the orthogonal line segment intersection problem. In Section 3.2 we move on to the more complex line segment problems from [GIS]. Finally, we in Section 3.3 survey results on "on-line" range searching in the I/O model, including the results in [Interval].

3.1 The Orthogonal Line Segment Intersection Problem

The orthogonal line segment intersection problem is that of reporting all intersecting orthogonal pairs in a set of N orthogonal line segment in the plane. The problem is important in computer graphics and VLSI design systems. In internal memory a simple optimal solution to the problem based on the plane-sweep paradigm works as follows (refer to Figure 3.1). A sweep with a horizontal line is made from top to bottom, and when the top endpoint of a vertical segment is reached the segment is inserted in a balanced search tree ordered according to x coordinate. The segment is removed again when its bottom endpoint is reached. When a horizontal segment is reached a range query is made on the search tree in order to report all the vertical segments intersecting the segment. As inserts and deletes can be performed in $O(\log_2 N)$ time and range querying in $O(\log_2 N + T')$ time, where T' is the number of reported segments, we obtain the optimal $O(N \log_2 N + T)$ solution. As discussed in Section 2.1.3 a simple natural external-memory modification of the plane-sweep algorithm would be to use a B-tree as the tree data structure, but this would lead to an $O(N \log_B n + t)$ I/O solution, and we are looking for an $O(n \log_m n + t)$ I/O solution. In the next two sections we discuss I/O-optimal solutions to the problem using the distribution sweeping and buffer tree techniques.

3.1.1 Distribution Sweeping

Distribution sweeping [67] is a powerful technique obtained by combining the distribution and the plane-sweep paradigms. Let us briefly sketch how it works in general. To solve

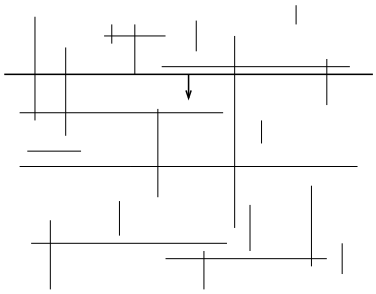


Figure 3.1: Solution to the orthogonal line segment intersection problem using plane-sweep.

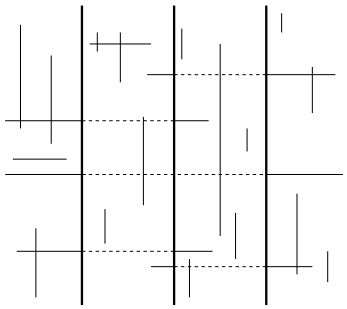


Figure 3.2: Solution to the orthogonal line segment intersection problem using distribution sweeping.

a given problem we divide the plane into m vertical *slabs*, each of which contains $\Theta(n/m)$ input objects, for example points or line segment endpoints. We then perform a vertical top to bottom sweep over all the slabs in order to locate components of the solution that involve interaction between objects in different slabs or objects (such as line segments) that completely span one or more slabs. The choice of m slabs ensures that one block of data from each slab fits in main memory. To find components of the solution involving interaction between objects residing in the same slab, the problem is then solved recursively in each slab. The recursion stops after $O(\log_m n/m) = O(\log_m n)$ levels when the subproblems are small enough to fit in internal memory. In order to get an $O(n \log_m n)$ algorithm we need to be able to perform one sweep in $O(n)$ I/Os.

Using the general technique to solve the orthogonal line segment intersection problem results in the following algorithm (refer to Figure 3.2): We divide the plane into m slabs and sweep from top to bottom. When a top endpoint of a vertical segment is encountered, we insert the segment in an active list (a stack where we keep the last block in internal memory) associated with the slab containing the segment. When a horizontal segment is encountered we scan through all the active lists associated with the slabs it completely spans. During this scan we know that every vertical segment in a list is either intersected by the horizontal segment, or will not be intersected by any of the following horizontal segments and can therefore be removed from the list. The process finds all intersections except those between vertical segments and horizontal segments (or portions of horizontal segments) that do not completely span vertical slabs (the solid part of the horizontal segments in Figure 3.2). These are found after distributing the segments to the slabs, when the problem is solved recursively for each slab. A horizontal segment may be distributed to two slabs, namely the slabs containing its endpoints, but it will at most be represented twice on each level of the recursion. It is easy to realize that if T' is the number of intersections reported, one sweep can be performed in $O(n + t')$ I/Os—every vertical segment is only touched twice where an intersection is not discovered, namely when it is distributed to an ative list and when it is removed again. Also blocks can be used efficiently because of the distribution factor of m. Thus by the general discussion of distribution sweeping we report all intersections in the optimal $O(n \log_m n + t)$ I/O operations.

In addition to the large number of problems solved in [67] using distribution sweeping, Chiang in [40, 41] performed a number of experiments in order to analyze the practical efficiency of the above algorithm. It turned out that when the test instances got just moderately

large the algorithm significantly outperformed various internal-memory algorithms as well as the B-tree version of the plane-sweep algorithm.

3.1.2 Rangesearch Operation on the Buffer tree

In [Buffer] we extend the basic buffer tree with a rangesearch operation, and thus we have the operations needed to solve the orthogonal line segment intersection problem with the plane-sweep solution described previously. Basically a rangesearch operation on the buffer tree is done in the same way as insertions and deletions. When we want to perform a rangesearch we create a special element which is pushed down the tree in a lazy way during buffer-emptying processes, just as all other elements. However, we now have to modify the buffer-emptying process. The basic idea in the modification is the following. When we meet a rangesearch element in a buffer-emptying process, instructing us to report elements in the tree between x_1 and x_2, we first determine whether x_1 and x_2 are contained in the same subtree among the subtrees rooted at the children of the node in question. If this is the case we just insert the rangesearch element in the corresponding buffer. Otherwise we "split" the element in two—one for x_1 and one for x_2—and report the elements in those subtrees where *all* elements are contained in the interval $[x_1, x_2]$. The splitting only occurs once and after that the rangesearch element is treated like inserts and deletes in buffer-emptying processes, except that we report the elements in the sub-trees for which all elements are contained in the interval. In [Buffer] we show how we can report all elements in a subtree (now containing other rangesearch elements) in a linear number of I/Os. Using the normal amortization argument it then follows that a rangesearch operation requires $O(\frac{\log_m n}{B} + t')$ I/Os amortized. Here $t' \cdot B$ is the number of elements reported by the operation.

Note that the above procedure means that the rangesearch operation gets batched in the sense that we do not obtain the result of query immediately. Actually parts of the result will be reported at different times as the element is pushed down the tree. However, this suffices in the plane-sweep algorithm in question, since the updates performed on the data structure do not depend on the results of the queries. This is the crucial property that has to be fulfilled in order to used the buffer tree structure. Actually, in the plane-sweep algorithm the entire sequence of updates and queries on the data structure is known in advance, and the only requirement on the queries is that they must all eventually be answered. In general such problems are known as *batched dynamic problems* [55].

To summerize, the buffer tree, extended with a rangesearch operation, can be used in the normal internal-memory plane-sweep algorithm for the orthogonal segment intersection problem, and doing so we obtain an optimal $O(n \log_m n + t)$ I/O solution to the problem.

3.2 Problems on General Line Segments in the Plane

Having discussed how the distribution paradigm (in form of distribution sweeping) and the data structuring paradigm (using the buffer tree) can be used to solve the relatively simple problem of orthogonal line segment intersection, we now turn to more complicated problems involving general line segments in the plane.

As mentioned previously GIS systems are a rich source of important problems that require good use of external-memory techniques. As most GIS systems at some level store data as layers of maps, and as one map typically is stored as a collection of line segments, important GIS operations involve such line segments in the plane. As an illustration, the computation of

new scenes or maps from existing information—also called map overlaying—is an important GIS operation. Some existing software packages are completely based on this operation [9, 10, 106, 124]. Given two thematic maps, that is, maps with e.g. indications of lakes, roads, or pollution levels, the problem is to compute a new map in which the thematic attributes of each location is a function of the thematic attributes of the corresponding locations in the two input maps. For example, the input maps could be a map of land utilization (farmland, forest, residential, lake), and a map of pollution levels. The map overlay operation could then be used to produce a new map of agricultural land where the degree of pollution is above a certain level. One of the main problems in map overlaying is "line-breaking", which can be abstracted as the red-blue line segment intersection problem, that is, the problem of computing intersections between segments in two internally non-intersecting sets.

In [GIS] we develop I/O-efficient algorithms for a large number of geometric problems involving line segments in the plane, with applications to GIS systems. In particular we address region decomposition problems such as trapezoid decomposition and triangulation, and line segment intersection problems such as the red-blue segment intersection problem. However, in order to do so we have to develop new techniques since distribution sweeping and buffer trees are inadequate for solving this type of problems. In Section 3.2.1 through 3.2.4 we sketch the problems encountered and the solutions derived in [GIS].

3.2.1 The Endpoint Dominance Problem

In this section we consider the *endpoint dominance problem* (EPD) defined in [GIS] as follows: Given N non-intersecting line segments in the plane, find the segment directly above each endpoint of each segment. Even though EPD seems as a rather simple problem, it is a powerful tool for solving other important problems. In [GIS] we show that if EPD can be solved in $O(\text{sort}(N))$ I/Os then the trapezoid decomposition of N non-intersecting segments, as well as a triangulation of a simple polygon with N vertices, can also be computed in $O(\text{sort}(N))$ I/Os.

We also show that EPD can be used to sort non-intersecting segments in the plane. A segment $\overline{AB}$ in the plane is above another segment $\overline{CD}$ if we can intersect both $\overline{AB}$ and $\overline{CD}$ with the same vertical line l, such that the intersection between l and $\overline{AB}$ is above the intersection between l and $\overline{CD}$. Two segments are incomparable if they cannot be intersected with the same vertical line. The problem of sorting N non-intersecting segments is to extend the partial order defined in this way to a total order. As we shall discuss later this problem is important in the solution of line segment intersection problems. Figure 3.3 demonstrates

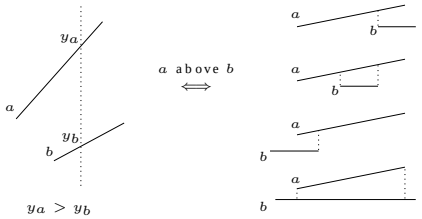


Figure 3.3: Comparing segments. Two segments can be related in four different ways.

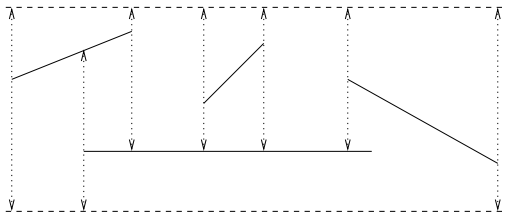


Figure 3.4: Algorithm for the segment sorting problem.

that if two segments are comparable then it is sufficient to consider vertical lines through the four endpoints to obtain their relation. Thus to sort N segments we add two "extreme" segments as indicated in Figure 3.4, and use EPD twice to find for each endpoint the segments immediately above and below it. Using this information we create a planar s,t-graph where nodes correspond to segments and where the relations between the segments define the edges. Then the sorted order is obtained by topologically sorting this graph in $O(\text{sort}(N))$ I/Os using an algorithm developed in [42]. Again this means that if EPD can be solved in $O(\text{sort}(N))$ I/Os then we can sort N segments in the same number of I/Os.

In internal memory EPD can be solved optimally with a simple plane-sweep algorithm. We sweep the plane from left to right with a vertical line, inserting a segment in a search tree when its left endpoint is reached and removing it again when the right endpoint is reached. For every endpoint we encounter, we also perform a search in the tree to identify the segment immediately above the point. One might think that it is equally easy to solve EPD in external memory using distribution sweeping or buffer trees. Unfortunately, this is not the case.

One important property of the plane-sweep algorithm is that only segments that actually cross the sweep-line are stored in the search tree at any given time during the sweep. This means that all segments in the tree are comparable and that we easily can compute their order. However, if we try to store the segments in a buffer tree during the sweep, is can also contain "old" segments which do not cross the sweep-line. This means that we can end up in a situation where we try to compare two incomparable segments. In general the buffer tree only works if we know a total order on the elements inserted in it, or if we can compare all pair of elements. Thus we cannot directly use the buffer tree in the plane-sweep algorithm. We could try to compute a total order on the segments before solving EPD, but as discussed above, the solution to EPD is one of the major steps towards finding such an order so this seems infeasible. We have not been able to design another algorithm for solving EPD using the buffer tree.

For similar reasons using distribution sweeping seems infeasible as well. Recall from Section 3.1.1 that in distribution sweeping we need to perform one sweep in a linear number of I/Os to obtain an efficient solution. Normally this is accomplished by sorting the objects by y coordinate in a preprocessing phase. This e.g. allows one to sweep over the objects in y order without sorting on each level of recursion, because as the objects are distributed to recursive subproblems their y ordering is retained. In the orthogonal line segment intersection case we presorted the segments by endpoint in order to sweep across them in endpoint y order. In order to use distribution sweeping to solve EPD it seems that we need to sort the *segments* and not the endpoints in a preprocessing step.

It should be mentioned that if we try to solve the other problems discussed in [GIS] with distribution sweeping or the buffer tree, we encounter problems similar to the ones described above.

3.2.2 The External Segment Tree

As attempts to solve EPD optimally using the buffer tree or distribution sweeping fail we are led to other approaches. The solution derived in [GIS] is based on an external version of the segment tree developed in [Buffer]. In this section we describe this structure.

The segment tree [23, 108] is a well-known dynamic data structure used to store a set of segments in one dimension, such that given a query point all segments containing the point can be found efficiently. Such queries are normally called *stabbing queries*. In internal memory

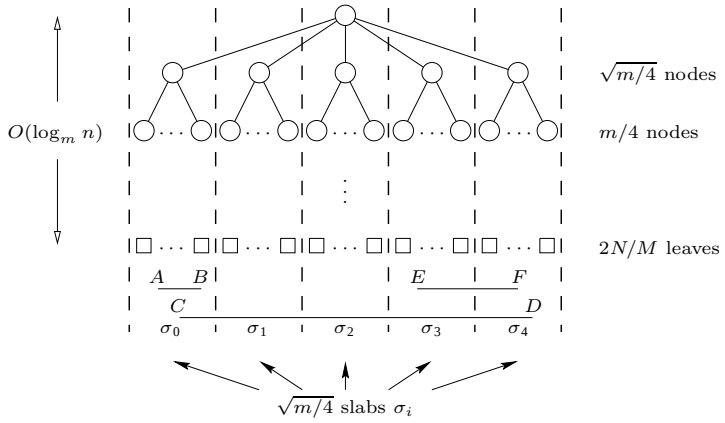


Figure 3.5: An external-memory segment tree based on a buffer tree over a set of N segments, three of which, $\overline{AB}$, $\overline{CD}$, and $\overline{EF}$, are shown.

a segment tree consists of a binary *base tree* which stores the endpoints of the segments, and a given segment is stored in the secondary structures of up to two nodes on each level of the tree. More precisely a segment is stored in all nodes v where it contains the interval consisting of all endpoints below v, but not the interval associated with $parent(v)$. A stabbing query can be answered efficiently on such a structure, simply by searching down the tree for the query value and reporting all the segments stored in each node encountered.

When we want to "externalize" the segment tree and obtain a structure with height $O(\log_m n)$, we need to increase the fan-out of the nodes in the base tree. This creates a number of problems when we want to store segments space-efficiently in secondary structures such that queries can be answered efficiently. The idea behind our approach is therefore to make the nodes have fan-out $\Theta(\sqrt{m})$ instead of the normal $\Theta(m)$. This smaller branching factor at most doubles the height of the tree, but it allows us to efficiently store segments in a number of secondary structures of each node. As we shall see in Section 3.3 ([Interval]) this simple idea turns out to be very powerful in general when designing external data structures from normal data structures that use secondary structures.

An external-memory segment tree based on the approach in [Buffer] is shown in Figure 3.5. The tree has branching factor $\sqrt{m/4}$, and is perfectly balanced over the endpoints of the segments it represents. Each leaf represents $M/2$ consecutive endpoints. The first level of the tree partitions the data into $\sqrt{m/4}$ intervals σ_i—for illustrative reasons we call them *slabs*—separated by dotted lines in Figure 3.5. *Multislabs* are defined as contiguous ranges of slabs, such as for example $[\sigma_1, \sigma_4]$. There are $m/8 - \sqrt{m}/4$ multislabs. The key point is that the number of multislabs is a quadratic function of the branching factor. Thus by choosing the branching factor to be $\Theta(\sqrt{m})$ rather than $\Theta(m)$ we have room in internal memory for a constant number of blocks for each of the $\Theta(m)$ multislabs. Segments such as $\overline{CD}$ in Figure 3.5 that spans at least one slab completely are called *long segments*. A copy of each long segment is stored in the largest multislab it spans. Thus, $\overline{CD}$ is stored in $[\sigma_1, \sigma_3]$. All segments that are not long are called *short segments*. They are not stored in any multislab, but are passed down to lower levels of the tree where they may span recursively defined slabs and be stored. $\overline{AB}$ and $\overline{EF}$ are examples of short segments. The portions of long segments that do not completely span slabs are treated as small segments. There are at most two such synthetically generated short segments for each long segment, hence total space utilization is

$O(n \log_m n)$ blocks. To answer a stabbing query we simply proceed down a path in the tree searching for the query value, and in each node encountered we report all the long segments associated with each of the multislabs that span the query value.

Because of the size of the nodes and auxiliary multislab data, the external segment tree is inefficient for answering single queries. However, using the buffer technique it can be used in a batched dynamic environments. In [Buffer] we show how to add buffers to the structure and perform a buffer-emptying process in $O(m + t')$ I/Os, such that in a batched dynamic environment inserting (and deleting) N segments in the tree and performing K queries requires $O((n+k) \log_m n + t)$ I/Os in total. Using the normal internal-memory plane-sweep algorithm this leads to an optimal algorithm for the pairwise rectangle intersection problem. The problem, which was also solved optimally in [67] using distribution sweeping, is very important in e.g. VLSI design rule checking [26].

3.2.3 External-Memory Fractional Cascading

It is possible to come close to solving EPD by first constructing an external-memory segment tree over the projections of the segments onto the x-axis and then performing stabbing queries at the x coordinates of the endpoints of the segments. However, what we want is the single segment directly above each query point in the y dimension, as opposed to all segments it stabs. This segment could be found if we were able to compute the segment directly above a query point among the segments stored in a given node of the external segment tree. We call such a segment a dominating segment. Then we could examine each node on the path from the root to the leaf containing a query, and in each such node find the dominating segment and compare it to the segment found to be closest to the query so far. When the leaf is reached we would then know the "global" dominating segment.

However, there are a number of problems that have to be dealt with in order to find the dominating segment of a query among the segments stored in a node. The main problems are that the dominating segment could be stored in a number of multislab lists, namely in all lists containing segments that contain the query, and that a lot of segments can be stored in a multislab list. Both of these facts seem to suggest that we need lots of I/Os to find the dominating segment. However, as we are looking for an $O(n \log_m n)$ solution and as the tree has $O(\log_m n)$ levels, we are only allowed to use a linear number of I/Os to find the positions of *all* the N query points among the segments stored in one level of the tree. This gives us less than one I/O per query point per node!

Fortunately, we are in [GIS] able to modify the external segment tree and the query algorithm to overcome these difficulties. As a first modification we strengthen the definition of the external segment tree and require that the segments in the multislab lists are sorted (according to the definition discussed in Section 3.2.1). We call the modified structure an *extended external segment tree*. Note that all pairs of segments in the same multislab list can be compared just by comparing the order of their endpoints on one of the boundaries, and that a multislab list thus can be sorted using a standard sorting algorithm. In [GIS] it is shown how to build an extended external segment tree on N non-intersecting segments in $O(\text{sort}(N))$ I/Os. The construction is basically done using distribution sweeping. An important consequence of the modification of the structure is that we now can sort all the segments stored in a node efficiently using the merging paradigm.

The sorting of the multislab lists makes it easier to search for the dominating segment in a given multislab list but it may still require a lot of I/Os. We also need to be able to look

for the dominating segment in many of the multislabs lists. We overcome these problems using *batched filtering* [67] and a new technique similar to what in internal memory is called *fractional cascading* [20, 38, 39, 123]. The idea in batched filtering is to process all the queries at the same time and level by level, such that the dominating segments in nodes on one level of the structure are found for all the queries, before continuing to consider nodes on the next level. In internal memory the idea in fractional cascading is that instead of e.g. searching for the same element individually in S sorted lists containing N elements each, each of the lists are in a preprocessing step augmented with sample elements from the other lists in a controlled way, and with "bridges" between different occurrences of the same element in different lists. These bridges obviate the need for full searches in each of the lists. To perform a search one only searches in one of the lists and uses the bridges to find the correct position in the other lists. This results in a $O(\log_2 N + S)$ time algorithm instead of an $O(S \log_2 N)$ time algorithm.

In the implementation of what could be called *external fractional cascading*, we do not explicitly build bridges but we still use the idea of augmenting some lists with elements from other lists in order to avoid searching in all of the lists. The construction is rather technical, but the general idea is the following. First we use a preprocessing step (like in fractional cascading) to sample a set of segments from each slab in each node and augment the multislab lists of the corresponding child with these segments. The set of sample segments for a slab consists of every $(2\sqrt{m/4}\,)$th segment that spans it in the sorted sequence of segments. The sampling is done in $O(\text{sort}(N))$ I/Os using the distribution paradigm. Having preprocessed the structure we filter the N queries through it. In order to do so in the optimal number of I/Os the filtering is done in a rather untraditional way—from the leaves towards the root. First the query points are sorted and distributed to the leaves to which they belong. Then for each leaf we find the dominating segment among the segments stored in the leaf for all query points assigned to the leaf. This can be done efficiently using an internal memory algorithm, because all the segments stored in a leaf fit in internal memory. This is also the reason for the untraditional search direction—we cannot in the same way efficiently find the dominating segments among the segments stored in the root of the tree, because more than a memory load of segments can be stored there. Finally, the queries in each node are sorted according to dominating segment, and we end up in a situation as indicated in Figure 3.6b). Next we go through $O(\log_m n)$ filtering steps, one for each level of the tree. Each filtering step begins with a set of queries at a given level, partitioned by the nodes on that level and ordered within the nodes by the order of the dominating segments on that level. This corresponds to the output of the leaf processing. The step should produce a similar configuration on the

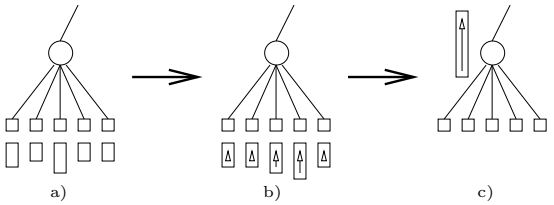


Figure 3.6: Filtering queries through the structure. An arrow in a list indicate that it is sorted.

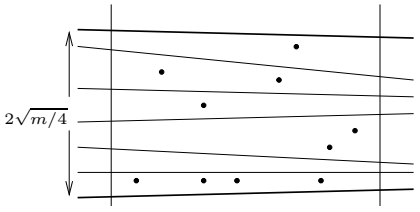


Figure 3.7: All queries between sampled segments (indicated by fat lines) must appear together in the list of queries for the slab.

next level up the tree. For one node this is indicated in Figure 3.6c). To perform the step on a node we "merge" the sorted list of queries associated with its children and the segments in the multislab list. The key property that allows us to I/O-efficiently find the dominating segments among the segments stored in the node, and to sort the queries accordingly, is that the list of queries associated with a child of the node cannot be too unsorted relative to their dominating segment in the node. As indicated in Figure 3.7 this is a result of the preprocessing phase—we are sure that all queries between two sampled segments appear together in the list of queries. This allows us to process all queries between two sampled segments efficiently and use the distribution paradigm to distribute them to one of $2\sqrt{m/4}$ lists—one for each of the segments between the sampled segments. The key property is that we are able to do this for all the slabs simultaneously using at most $2\sqrt{m/4} \cdot \sqrt{m/4} = m/2$ blocks of internal memory.

In [GIS] it is shown that we can do a filtering step in $O(n)$ I/Os, and thus we solve EPD in $O(n \log_m n)$ I/O operations. Note the similarity with fractional cascading where we search in one list and then use the bridges to avoid searching in the other list. We find dominating segments (efficiently) in the leaves and sort the queries accordingly only this ones, and then the sampled and augmented segments obliviates the need for search in the other lists.

3.2.4 Algorithms obtained using the Endpoint Dominance Problem

The $O(\text{sort}(N))$ solution to EPD has several almost immediate consequences as discussed in [GIS]. As already mentioned it leads to an $O(\text{sort}(N))$ algorithm for segment sorting. Similarly it leads to an algorithm for trapezoid decomposition of a set of segments [92, 108] with the same I/O bound, as the core of this problem precisely is to find for each segment endpoint the segment immediately above it. The ability to compute a trapezoid decomposition of a simple polygon also leads to an $O(\text{sort}(N))$ polygon triangulation algorithm using a slightly modified version of an internal-memory algorithm [61]. Furthermore, if one takes a closer look at the algorithm for EPD one realizes that it works in general with K query points, that are not necessarily endpoints of the segments. Therefore the result also leads to an $O((n+k) \log_m n)$ I/O solution to the batched planar point location problem.

But maybe the most important results obtained from the solution to EPD are the results on reporting line segment intersections. As already mentioned, the red-blue line segment intersection problem is of special interest because it is an abstraction of the important map-overlay problem in GIS Systems. Using the ability to sort non-intersecting segments we solve the problem in $O(n \log_m n + t)$ I/O operations [GIS]. The solution is based on distribution sweeping with the new idea of a branching factor of $\Theta(\sqrt{m})$ instead of $\Theta(m)$, and the segment sorting algorithm is used twice in a preprocessing step to sort two sets consisting of the segments of one color and the endpoints of segments of the other color. The general segment intersection problem however cannot be solved using this method. The reason is that we cannot sort intersecting segments. Nevertheless it is shown in [GIS] how external segment trees can be used to establish enough order on the segments to make distribution sweeping possible. The general idea in the algorithm is to build an extended external segment tree on all the segments, and during this process to eliminate *on the fly* any inconsistencies that arise because of intersecting segments. This is done using a number of the external priority queue developed in [Buffer], and it leads to a solution for the general problem that integrates all the elements of the red-blue algorithm into one algorithm; extended external segment trees, distribution sweeping, distribution, merging and external priority queues. The algorithm use $O((n+t) \log_m n)$ I/O operations.

3.3 External-Memory On-line Range Searching

We now turn our attention to external data structures for range searching where we want the answer to a query right away. Even though "the rules of the game" is a lot different when designing such on-line data structures compared to the off-line ones we have considered so far, it turns out that we can use our previous ideas to design efficient structures also under this constraint.

As discussed in Section 2.1.3, the B-tree [21, 51, 82] is the standard external-memory search tree data structure on which insert, delete and search operations can be performed in $O(\log_B n)$ I/Os. It is also easy to perform a rangesearch operation on a B-tree in $O(\log_B n + t)$ I/O operations, and as mentioned in Section 2.1.3 this is optimal when one wants an answer to a query right away. But while the B-tree efficiently supports external dynamic one-dimensional range searching, it is inefficient for handling more general problems like two-dimensional or higher-dimensional range searching. The problem of two-dimensional range searching both in main and external memory has been the subject of much research, and many elegant data structures like the range tree [24], the priority search tree [89], the segment tree [23], and the interval tree [53, 54] have been proposed for internal-memory two-dimensional range searching and its special cases. These structures are not efficient when mapped to external memory. However, the practical need for I/O support has led to the development of a large number of external data structures, which have good average-case behavior for common problems but fail to be efficient in the worst case sense [68, 69, 86, 95, 103, 111, 114, 115, 117].

Recently some progress has been made on the construction of external two-dimensional range searching structures with good worst-case performance. In Figure 3.8 the different special cases of general two-dimensional range searching are shown. As discussed in [79] it is easy to realize that the problem of answering stabbing queries on a set of interval (segments) reduces to the simplest of these queries called **diagonal corner queries**; If an interval $[x, y]$ is viewed as the point (x, y) in the plane, a stabbing query with q reduces to a diagonal corner query with corner (q, q) on the line $x = y$—refer to Figure 3.8. In [Interval] we develop an optimal external-memory data structure for this problem. The structure uses optimal $O(n)$ space, and updates, insertions and deletions of points (intervals), can be performed in $O(\log_B n)$ I/Os while diagonal corner queries (or stabbing queries) require $O(\log_B n+t)$ I/Os. We discuss the structure in Section 3.3.1.

In [110] a technique called *path caching* for transforming an efficient internal memory data structure into an I/O efficient one is developed. Using this technique on the priority search tree [89] results in a structure that can be used to answer general **two-sided** two-dimensional queries, which are slightly more general than diagonal corner queries—refer again

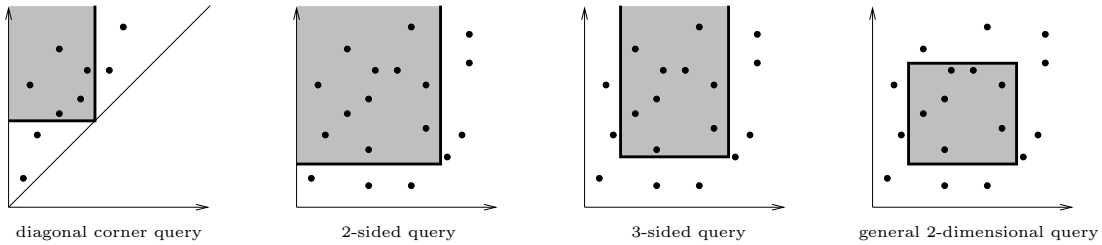


Figure 3.8: Different types of two-dimensional queries.

to Figure 3.8. This structure answers queries in the optimal $O(\log_B n + t)$ I/Os and supports updates in amortized $O(\log_B n)$ I/Os, but uses slightly non-linear space $O(n \log_2 \log_2 B)$.

In internal memory the priority search tree actually answers slightly more general queries than two-sided queries, namely **three-sided** two-dimensional queries (Figure 3.8), in optimal query and update time using linear space. A number of attempts have been made to externalize this structure [27, 74, 110]. The structure in [74] uses linear space but answers queries in $O(\log_2 N + t)$ I/Os. The structure in [27] also uses linear space but answers queries in $O(\log_B n + T)$ I/Os. In both papers a number of non-optimal dynamic versions of the structures are also developed. The structure in [110] was developed using path caching and answers queries in the optimal number of I/Os $O(\log_B n + t)$ but uses slightly non-linear $O(n \log_2 B \log_2 \log_2 B)$ space. This structure supports updates in $O(\log_B n (\log_2 B)^2)$ I/Os amortized. In [121] another attempt is made on designing a structure for answering three sided queries and a dynamic data structure called the *p-range tree* is developed. The structure uses linear space, answers queries in $O(\log_B n + t + IL^*(B))$ I/Os and supports updates in $O(\log_B n + (\log_B n)^2/B)$ I/Os amortized. The symbol $IL^*(\cdot)$ denotes the iterated $\log^*$ function, that is, the number of times one must apply $\log^*$ to get below 2. The p-range tree can be extended to answer **general two-dimensional** queries (or four-sided queries—refer to Figure 3.8) in the same number of I/Os using $O\left(n \log_2 N/(\log_2 \log_B n)\right)$ blocks of space. Dynamic versions of the structure, as well as tradeoffs between I/O and space use, are also discussed in [121].

Very recently static structures for the various forms of **three-dimensional** range searching which all answer queries in $O((\log_2 \log_2 \log_B n) \log_B n + t)$ I/Os are developed in [130].

3.3.1 External-Memory Interval Management

In [79] the *dynamic interval management* problem is considered. This problem is crucial for indexing constraints in constraint databases and in temporal databases [78, 79, 109]. Dynamic interval management is the problem of dynamically maintaining a set of intervals, such that all intersections between the intervals and a query interval can be reported. As discussed in [79] the key component of external dynamic interval management is to support stabbing queries, which is the same as answering diagonal corner queries.

In [Interval] we develop an optimal external-memory data structure for the stabbing query (or diagonal corner query) problem. The structure is an external-memory version of the interval tree [53, 54] and it uses ideas similar to the ones in Section 3.2.2. As mentioned the structure uses optimal $O(n)$ space and updates can be performed in $O(\log_B n)$ I/Os while stabbing queries require $O(\log_B n + t)$ I/Os. This leads to an optimal solution to the interval management problem. Unlike other external range search structures the update I/O bounds for the data structure are worst-case. The worst-case bounds are obtained using a balancing technique for balanced trees which can be regarded as weight-balancing of B-trees. The tree can be used to remove amortization from other external-memory as well as internal-memory data structures.

In the next section we sketch the main idea in the *external interval tree* in the case where the segments stored in the structure all have endpoints in a fixed set of points (of size $O(N)$). In Section 3.3.1.2 we then discuss how the weight-balanced B-tree is used to remove the fixed-endpoint-set assumption.

3.3.1.1 External-Memory Interval Tree

In internal memory an interval tree (as a segment tree) consists of a binary tree that stores the endpoints of the segments, and where the segments are stored in secondary structures of the internal nodes of the tree [53]. As in the segment tree an interval is associated with each node, consisting of all the endpoints below the node, but unlike in a segment tree a segment is only stored in the secondary structures of one node. More precisely a segment is stored in the root r if it contains the "boundary" between the two intervals associated with the children of r. If it does not contain this boundary it is recursively stored in the subtree rooted in the child on the same side of the boundary as the segment. The set of segments stored in r is stored in two structures: a search tree sorted according to left endpoint of the segments and one sorted according to right endpoint. To preform a stabbing query with x one only needs to report segments in r that contain x and recurse. If x is contained in the interval of the left child this is done by traversing the segments stored in the tree sorted according to left endpoint, from the segments with smallest left endpoint towards the ones with largest left endpoint, until one meets a segment that does not contain x. All segments after this segment in the sorted order will not contain x.

In order to "externalize" the interval tree we use the ideas from Section 3.2.2 ([Buffer] and [GIS]). As base tree we use a tree with fan-out $\sqrt{B}$. As previously the interval associated with a node v is divided into $\sqrt{B}$ slabs (intervals) defined by the children of v, which again results in $O(B)$ multislabs. In analogy with internal memory, a segment is stored in the secondary structures of the highest node where it crosses one or more of the slab boundaries. Each internal node v contains $O(B)$ secondary structures; $O(B)$ multislab lists implemented as B-trees and two B-trees for each of the $\sqrt{B}$ slab-boundaries. The two B-trees for a boundary b_i is called the right and left *slab list*, respectively. The left slab list for b_i contains segments with left endpoint between b_i and b_{i-1} and it is sorted according to left endpoint. Similarly, the right slab list contains segments with right endpoint between b_i and b_{i+1} and it is sorted according to right endpoint.

A segment is stored in two or three structures—two slab lists and possibly a multislab list. As an example, segment s in Figure 3.9 will be stored in the left slab list of b_2, in the right slab list of b_4, and in the multislab list corresponding to these two slab boundaries. Note the similarity between the slab lists and the sorted lists of segments in the nodes of an internal-memory interval tree. As in the internal case s is stored in two sorted lists—one for each of the endpoints of s. This represents the part of s to the left of the leftmost boundary contained in s, and the part to the right of the rightmost boundary contained in s (the parts that are stored recursively in a segment tree). Unlike the internal case, we also need to store/represent the part of s between the two extreme boundaries. This is done using the multislab lists.

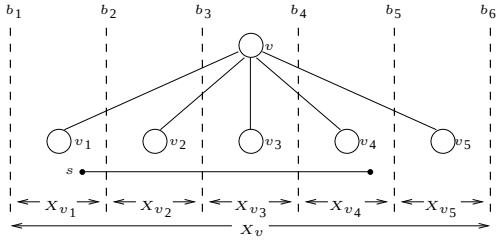


Figure 3.9: A node in the base tree.

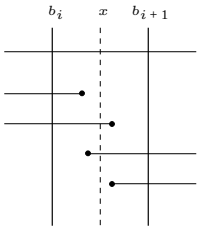


Figure 3.10: Reporting segments.

In order to obtain optimal bounds we need one more structure. This structure is called the *underflow structure* and contains segments from multislab lists containing $o(B)$ segments. Thus we only explicitly store a multislab list if it contains $\Omega(B)$ segments. The underflow structure always contains less than B^2 segments and is implemented using a structure called a corner structure [79]. This structure allows insertion and deletion of segments in a constant number of I/Os, and it can be used to answer stabbing queries in $O(t+1)$ I/Os (when the number of segments stored is $O(B^2)$).

The linear *space* bound basically follows from the fact that each segment is only stored in a constant number of secondary structures. Note however, that if we did not store segments belonging to a multislab having $o(B)$ segments in the underflow structure, we could have $\Omega(B)$ sparsely utilized blocks in each node, which would result in a non-linear space bound.

To perform a stabbing *query* with x we search down the structure for the leaf containing x, and in each internal node we report the relevant segment. Let v be one such node and assume that x falls between slab boundary b_i and slab boundary b_{i+1}—refer to Figure 3.10. To report the relevant segments we first report the segments in all multislab lists that contain segments crossing b_i and b_{i+1}, and perform a stabbing query with x on the underflow structure and report the result. Then in analogy with the internal memory case we report segments from b_i's right slab list from the largest towards the smallest (according to right endpoint) until we meet a segment that does not contain x, and segments from b_{i+1}'s left slab list from the smallest towards the largest. It is easy to realize that this process will indeed report all the relevant segments. The query algorithm uses an optimal number of I/O operations because in each of the $O(\log_B n)$ nodes on the search path we only use $O(1)$ I/Os that are not "paid for" by reportings (blocks read that contain $\Theta(B)$ output segments). The underflow structure is again crucial because it means that all blocks loaded in order to report segments in the relevant multislab lists contain $\Theta(B)$ segments.

It is equally easy to do an *update*. We search down the base tree using $O(\log_B n)$ I/Os until we find the first node where the segment in question crosses one or more slab boundaries. Then we update the two relevant slab lists and the relevant multislab list (if any) in $O(\log_B n)$ I/Os each. Of course we also have to take care of moving segments between the underflow structure and the multislab lists in some cases, but as shown in [Interval] this can also be handled efficiently.

3.3.1.2 Weight-balanced B-trees

In the previous section we assumed that the segments stored in the external interval tree have endpoints in a fixed set. Put another way we ignored rebalancing of the base tree. If we want to remove this assumption we will have to use a dynamic search tree as base tree. To perform an update we would then first insert or delete the new endpoints from the base tree and perform the necessary rebalancing, and then insert the segment as described. To preserve the $O(\log_B n)$ update bound we have to be able to perform the rebalancing in the same number of I/Os.

There are many possible choices for a dynamic base tree — height-balanced, degree-balanced, weight-balanced, and so on. The different choices of dynamic tree lead to different balancing methods, but normally (and also in this case) when search trees are augmented with secondary structures it turns out that a rebalancing operation requires work proportional to the number of elements in the subtree rooted at the node being rebalanced—this is normally called the *weight* of the node. In internal memory a natural choice of dynamic base tree is

the BB[α]-tree [96], because in this structure a node with weight w can only be involved in a rebalancing operation for every $\Omega(w)$ updates that access (go through) the node [30, 90]. This leads to an $O(1)$ amortized bound on performing a rebalancing operation and thus to an $O(\log_B n)$ amortized rebalance bound. Unfortunately BB[α]-trees are not very suitable for implementation in external memory. The natural candidate however, the B-tree, does not have the property that a node of weight w can only be involved in a rebalance operation for every $\Omega(w)$ updates that access the node. In [Interval] a variant of B-trees, called weight-balanced B-trees, that possesses this property is developed. Basically a weight-balanced B-tree is a B-tree with some extra constraints, namely constraints on the weight of a node as a function of the level it is on in the tree. In a normal B-tree, nodes can have subtrees of very varying weight. Actually the ratio between the largest subtree weight possible and the smallest is exponential in the height of the tree the node is root in. In the weight-balanced B-tree this ratio is a small constant factor.

The development of the weight-balanced B-tree leads to optimal *amortized* I/O bounds on the external interval tree. However, in [Interval] we show how we can spread the rebalancing work on a node over a number of updates in an easy way, and thus remove the amortization from the bounds. Actually, it turns out that the weight-balanced B-tree can be used as a simple alternative to existing complicated ways of removing amortization from internal-memory data structures. For example it can be used in the structure described in [136] for adding range restriction capabilities to dynamic data structures, and by fixing B to a constant in the external interval tree one obtains an internal-memory interval tree with worst-case update bounds. Even though it is not reported in the literature it seems possible to use the techniques in [136] to remove the amortization from the internal-memory interval tree, but our method seems much simpler.

The main property of the weight-balance B-tree that allows relatively easy removal of amortization, is that rebalancing is done by splitting of nodes instead of rotations as in the BB[α]-tree case. Intuitively splitting a node v into v' and v'' only affects the secondary structures of v, and for example not the secondary structures of parent(v) because the set of endpoints below this node is not affected by the split. This allows one to continue to use the "old" secondary structures of v after the split, and lazily build the structure of v' and v'' over the updates that pass them before one of them needs to be split again. If rebalancing is done using rotations, several nodes are affected by a rebalance operation, and it is not so obvious how to continue to answer queries correctly while building the necessary structures lazily. Also, it is necessary to prevent other rebalancing operations (rotations) involving any of these node (not only the node that was initially out of balance) to take place before the rebuilding is complete.

3.4 Bibliographic Notes and Summery of our Contributions

Goodrich et al. [67] designed efficient external-memory algorithms for a large number of important problems, mainly using the distribution sweeping technique and the distribution paradigm. In [Buffer] we use the buffer tree technique to develop buffered versions of one-dimensional range trees and of segment trees. Using the data structuring paradigm, the standard plane-sweep algorithms then lead to optimal algorithms for the orthogonal line segment intersection problem, the batched range searching problem and the rectangle intersecting problem. Apart from the fact that the normal internal-memory solutions are used, the power of these solutions is that all I/O specific parts are hidden in the data structures. Furthermore, we believe that our algorithms are of practical interest due to small constants in the asymptotic bounds.

In [GIS] we then develop I/O-efficient algorithms for a number of problems involving (general) line segments in the plane. These algorithms have important applications in GIS systems where huge amounts of data are frequently handled. We introduce the endpoint dominance problem and design an I/O-efficient algorithm for it by combining several of the known techniques, especially the segment tree from [Buffer], with a new technique which can be regarded as a version of fractional cascading for external memory. The solution leads to an algorithm for the batched planar point location problem. A solution to this problem is also given in [67] but it only works when the planar subdivision is monotone whereas our solution works for general subdivisions. The endpoint dominance algorithm also leads to new I/O-efficient algorithms for trapezoid decomposition, triangulation of simple polygons, segment sorting, and red-blue and general line segment intersection. It remains open if one can solve the general line segment intersection problem in $O(n \log_m n + t)$ I/Os (it should however be noted that in internal memory a simple $O((N + T) \log_2 N)$ time plane-sweep solution [25] was also known long before the complicated optimal algorithm [37]). It also remains open if one can triangulate a simple polygon in a linear number of I/Os, e.g. if the vertices of the polygon are given in the sorted order they appear around the perimeter.

In [Interval] we develop external on-line data structures for the stabbing query problem. As discussed in [79] an optimal structure for this problem leads to an optimal solution to the interval management problem. The segment tree structure solves the stabbing query problem in internal-memory and some attempts have been made to externalizing this structure in an on-line setting [28, 110]. They all use $O(n \log_2 n)$ blocks of external memory and the best of them [110] is static and answers queries in the optimal $O(\log_B n+t)$ I/Os. In [Interval] we use our ideas from [Buffer] (Section 3.2.2) to develop an on-line version of the segment tree with the improved space bound $O(n \log_B n)$ blocks. Furthermore, our structure is dynamic with worst-case optimal update and query bounds (the result is not discussed in the survey). However, our segment tree structure still uses more than linear space. In [79] a data structure for the stabbing query problem using optimal $O(n)$ blocks of external memory is developed. The structure, called the *metablock tree*, answers queries in the optimal number of I/Os but is fairly involved and supports only insertions (not deletions) in non-optimal $O(\log_B N + (\log_B N)^2/B)$ I/Os amortized. In [Interval] we developed an optimal external version of the interval tree. This structure

	Space (blocks)	Query I/O bound	Update I/O bound
Priority search tree [74]	$O(n)$	$O(\log_2 N + t)$	
XP-tree [27]	$O(n)$	$O(\log_B n + T)$	
Metablock tree [79]	$O(n)$	$O(\log_B n + t)$	$O(\log_B n + (\log_B n)^2/B)$ amortized (inserts only)
P-range tree [121]	$O(n)$	$O(\log_B n + t + IL^*(B))$	$O(\log_B n + (\log_B n)^2/B)$ amortized
Path Caching [110]	$O(n \log_2 \log_2 B)$	$O(\log_B n + t)$	$O(\log_B n)$ amortized
Interval tree [Interval]	$O(n)$	$O(\log_B n + t)$	$O(\log_B n)$

Figure 3.11: Comparison of space efficient data structure for interval management.

improves on the metablock tree and settles an important open problem in databases and I/O algorithms. In Figure 3.11 our result is compared with previous space-efficient structures for the interval management problem. The structures in [Interval] use a novel balancing technique for search trees which can be regarded as weight-balancing of B-trees. This structure can be used to remove amortization from external as well as internal-memory data structures. Actually, the structures in [Interval] are the first external structures for any 2-dimensional range searching problem with *worst-case* update bounds. Fixing B to a constant in the structures result in worst-case efficient internal-memory segment and interval trees. The structures in [Interval] also work without assumptions on the size of the internal memory, whereas all the results on dynamic data structures cited in Section 3.3 assume that $m = O(B)$, that is, that the internal memory is capable of holding $O(B^2)$ elements. Finally, it should be mentioned that [79] also discusses a method for extending the metablock tree to answer three-sided queries. The resulting structure still uses linear space but it is static. It is capable of answering queries in $O(\log_B n + \log_2 B + t)$ I/Os.

Chapter 4

Graph Problems

Graph-theoretic problems arise in many large-scale computations. In this section we consider external-memory algorithms for graph problems, and for that purpose we introduce the following additional parameters:

$$\begin{aligned} V &= \text{number of vertices in the input graph;} \\ E &= \text{number of edges in the input graph.} \end{aligned}$$

We assume that $E \geq V$ and note that $N = V + E$.

Until recently only work on selected graph problems in restricted external-memory models has been done. In [122] transitive closure computations are considered in a model where the set of vertices fits in main memory. Related work is done in [75, 76]. In [97] work on graph traversal in external memory is done, which primarily addresses the problem of storing graphs and not the problem of performing computations on them. Related work on storing trees is done in [66, 71]. In [58] memory management problems for maintaining connectivity information and paths on graphs are studied, and in [85] linear relaxation problems are studied.

Recently, Chiang et al. [40, 42] considered graph problems in the general I/O model. They developed efficient algorithms for a large number of important graph problems, including Euler-tour computation, expression-tree evaluation, least common ancestors, connected and biconnected components, minimum spanning forest and topological sorting of planar s,t-graphs. Some of these algorithms are improved in [Buffer]. In [OBDD] we analyze the I/O complexity of Ordered Binary-Decision Diagram (OBDD) manipulation. OBDDs are graph-based data structures and represents the state-of-the-art structure for boolean function manipulation. We analyze existing algorithms in an I/O-environment, prove I/O lower bounds on the fundamental OBDD-manipulation algorithms, and develop new efficient algorithms. Some I/O graph algorithms are also reported in [83].

In internal memory any permutation of N elements can be produced in $O(N)$ time, and on an N processor PRAM it can be done in constant time. In any case the work is $O(N)$. However, as discussed previously it is in general not possible to perform arbitrary permutations in a linear number of I/Os, and this is one of the important features that distinguish the I/O-model from the other models. The difference becomes especially prominent when considering graph problems, as one can prove $\Omega(\text{perm}(N))$ lower bounds on several graph problems that can be solved in linear time in internal memory. As $O(\text{perm}(N)) = O(\text{sort}(N))$ for all realistic values of B and M as discussed in Section 2.2, the lower bound means that sort-

ing can normally be used as a subroutine in designing graph algorithms, something which is generally not possible in internal memory. It also means that $O(\text{sort}(N))$ I/O algorithms are optimal for all practical purposes.

In [42] $\Omega(\text{perm}(N))$ is proved to be a lower bound on a large number of important graph problems. Also in [40, 42] a large number of $O(\text{sort}(N))$ I/O algorithms are developed. These algorithms are mainly developed using *PRAM simulation* and an $O(\text{sort}(N))$ solution to the *list ranking* problem. We sketch the main ideas in these two results in Section 4.2, but before doing so we in Section 4.1 try to illustrate the difference in hardness of graph problems in internal and external memory. This is done by considering the I/O performance of the very simple internal-memory algorithm for the list ranking problem, and by discussing the "circuit evaluation" problem. We also discuss a very useful technique for traversing a graph using a priority queue. Finally, we in Section 4.3 survey the results obtained in [OBDD].

4.1 Basic Issues in External Graph Algorithm

Consider the list ranking problem defined as follows. We are given an N-vertex linked list stored as an (unsorted) sequence of vertices, each with a pointer to the successor vertex in the list (Figure 4.1). Our goal is to determine the rank of each vertex, that is, the number of links to the end of the list. In the PRAM world list ranking is a very fundamental graph problem which extracts the essence in many other problems, and it is used as an important subroutine in many parallel algorithms [8]. As mentioned this turns out also to be the case in external memory [40, 42].

In internal memory the list ranking problem is easily solved in $O(N)$ time. We simply traverse the list by following the pointers, and rank the vertices $N-1, N-2$ and so on in the order we meet them. In external memory, however, this algorithm performs terribly. Imagine that we run the algorithm on a machine where the internal memory is capable of holding two blocks ($m = 2$) when the list is blocked as indicated in Figure 4.2. Assume furthermore that we use a least recently used (LRU) paging strategy. First we load block 1 and give A rank $N-1$. Then we follow the pointer to E, that is, we load block 3 and rank E. Then we follow the pointer to D, loading block 2 while removing block 1 from internal memory. Until now we have done an I/O every time we follow a pointer. Now we follow the pointer to B, which means that we have to load block 1 again. To do so we have to remove block 3 from internal memory. This process continues and it is easy to realize that we do an I/O every time we follow a pointer. This means that the algorithm uses $O(N)$ I/Os as opposed to the linear external-memory bound $O(n) = O(N/B)$ I/Os. As mentioned in the introduction the difference between these two bounds can be very significant in practice as B typically is on the order of thousands.

In general the above type of behavior is characteristic for internal-memory graph algorithms when analyzed in an I/O-environment. The lesson is that one should be very careful

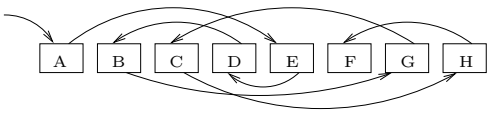


Figure 4.1: List ranking problem.

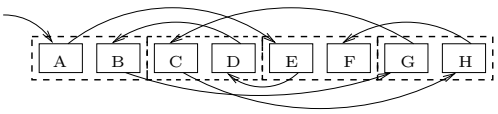


Figure 4.2: List ranking in external memory ($B = 2$).

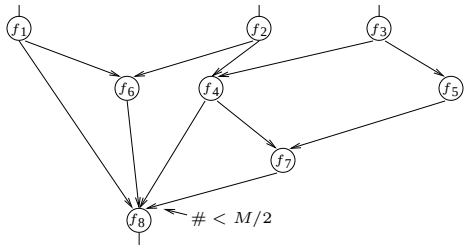


Figure 4.3: The circuit evaluation problem.

about following pointers and in graph algorithms be especially careful to ensure a high degree of locality in the access to data (what is normally referred to as locality of reference). Still the host of algorithms for graph problems, both in theory and practice, make extensive use of pointers, and it seems that in several practical applications the size of the internal memory has indeed been the limiting factor for the size of the problem instances one has been able to solve. Before we discuss the external solution to the list ranking problem, consider the related problem of "circuit evaluation" defined as follows [42]: Given a bounded fan-in boolean circuit whose description is stored in external memory, the problem is to evaluate the function computed by the network. It is assumed that the representation of the circuit is topologically sorted, that is, the labels of the vertices come from a total order $<$, and for every edge (v, w) we have $v < w$. Refer to Figure 4.3. Nothing is assumed about the functions in the vertices except that they take at most $M/2$ inputs. In [42] a solution to the problem is called a *time-forward processing*, which is motivated by thinking of vertex v as being evaluated at "time" v. The main issue in such an evaluation is to ensure that when one evaluates a particular vertex one has the values of its inputs in internal memory.

As in the list ranking case it is easy to realize that evaluating the circuit using the obvious linear time internal-memory algorithms, which traverse the circuit in topological order and at every vertex visit the immediate predecessor vertices, results in an $O(N)$ I/O bound in the worst case. However, using an external priority queue we can give a rather obvious algorithm for solving the problem in $O(\mathrm{sort}(N))$ I/Os [Buffer]. When we evaluate a vertex v we simply send the result "forward in time" to the appropriate vertices by inserting it in the priority queue with priority w for all edges (v, w). We can then obtain the inputs to the next vertex in the topological order just by performing a number of deletemin operations on the queue. The $O(n \log_m n)$ I/O bound follows immediately from the I/O bound on the priority queue operations proved in [Buffer]. As we will discuss in Section 4.2 and 4.3 this idea of traversing a graph using an external priority queue turns out to be a powerful tool when designing external-memory graph algorithms.

4.2 External List Ranking and PRAM Simulation

In [40, 42] a large number of efficient external-memory graph algorithms are designed from PRAM algorithms using PRAM simulation and an $O(\mathrm{sort}(N))$ I/O algorithm for external list ranking. In [42] it is proved that list ranking requires $O(\mathrm{perm}(N))$ I/Os in external memory, and $O(\mathrm{sort}(N))$ I/O algorithms for the problem are discussed in [40, 42, 83] and in [Buffer]. Thus the problem is solved optimally for all realistic values of B and M. In the following two subsection we sketch the ideas in the list-ranking results and the main idea in PRAM simulation.

4.2.1 External List Ranking

In order to develop an $O(\text{sort}(N))$ algorithm for list ranking Chiang et al. [40, 42] use a framework from PRAM algorithms for the problem [8, 46]. The general idea is recursive; First one produces an independent set of $\Theta(N)$ vertices, that is, a set of vertices none of which are successors of each other. Then $O(1)$ sorts and scans are used to *bridge* out the vertices in the set, that is, for a vertex v in the set one lets the predecessor of v point directly to the successor of v. This can be done because the vertices are independent, that is, because the predecessor and the successor of v are not in the set. Now the resulting chain is ranked recursively, and $O(1)$ sorts and scans are used to reintegrate the removed vertices in the final solution. Details can be found in [8, 40, 42, 46]. If the independent set can be found in $O(\text{sort}(N))$ I/Os the whole algorithm also uses $O(\text{sort}(N))$ I/Os, because $O(\text{sort}(N)$ I/Os is used to bridge out the independent set and because the algorithm is only called recursively on a constant fraction of the vertices.

In [42] a number of methods for computing the independent set in $O(\text{sort}(N))$ I/O are sketched. One of them uses time-forward processing as discussed in Section 4.1 (on a slightly modified version of the circuit evaluation problem) to find a 3-coloring of the list, and the independent set is then chosen to be the vertices colored by the most popular color. The 3-coloring algorithm works as follows: First the edges in the list ranking problem are separated into forward edges $\{(a,b)|a<b\}$ and backward edges $\{(a,b)|a>b\}$. Each of these sets is a set of chains. The vertices in the first set are then colored with 0 and 1; First the start vertex of each chain is identified and colored 0, and then time-forward processing is used to color the rest of the vertices in each chain in alternating colors. This can be done in $O(\text{sort}(N))$ I/Os using the algorithm in [Buffer]. After that the second set is colored in a similar way with 2 and 1, starting each chain with 2. Now if a vertex is given two different colors (because it ends one chain and starts another) it is colored 0 unless the two colors are 2 and 1, in which case it is colored 2. The result is a 3-coloring of the N vertices produced in $O(\text{sort}(N))$ I/Os.

The above 3-coloring leads to an $O(\text{sort}(N))$ I/O algorithm for list ranking. In [42] a $\Omega(\text{perm}(N))$ I/O lower bound is proved on the problem, and the algorithm is thus optimal for all realistic values of B and M. In [5] it was shown that if an algorithm is capable of performing $N!$ permutations then at least one of them requires $\Omega(\text{perm}(N))$ I/Os. In [42] it is shown that even if an algorithm is only capable of performing $(N!)^\alpha N^c$ ($0<\alpha\leq 1$, c arbitrary) permutations the lower bound still applies. This is used to prove a lower bound on the *proximate neighbors problem* defined as follows: Given N elements in external memory, each with a key $k\leq N/2$, such that for each possible value of k, exactly two elements have key-value k. The problem is then to permute the elements such that elements with identical key-value are in the same block. It is shown in [42] that an algorithm solving the proximate neighbors problem is capable of performing $\sqrt{N!}/N^{1/4}$ permutation, and therefore it must use $\Omega(\text{perm}(N))$ I/Os in the worst case. Finally, the $\Omega(\text{perm}(N))$ I/O lower bound for list ranking is obtained by reduction from a slightly modified version of the proximate neighbors problem [42, 128].

Finally, it should be mentioned that Kumar [83] recently has shown how to use a priority queue to solve the list ranking problem in a more direct way, and that Vengroff [127] has performed experiments which show that the external list ranking algorithms indeed performs substantially better than the internal-memory algorithm on large random instances of the problem.

4.2.2 PRAM Simulation

As the name suggests the general idea in PRAM simulation is to design an external-memory algorithm by simulating an existing PRAM algorithm. The underlying intuition is that the I/O and PRAM models are related in the sense that they both support independent computation on local data. The main idea in PRAM simulation is to simulate a single step of a N processor $O(N)$ space PRAM algorithm in $O(\text{sort}(N))$ I/Os. Then the simulation of the whole PRAM algorithm can also be performed in $O(\text{sort}(N))$ I/Os, provided that the PRAM algorithm possesses the "geometrically decreasing size" property, that is, that after a constant number of steps the number of active processors is decreased by a constant factor. The $O(\text{sort}(N))$ I/O bound then follows from the fact that only the active processors need to be simulated. Fortunately, many PRAM algorithms have this property and can therefore be simulated effectively.

In a single step of a PRAM algorithm each of the N processors reads $O(1)$ memory locations, performs some computation, and writes to $O(1)$ memory locations. The main problem in simulation such a step is to ensure that when the computation of one processor is simulated the content of the corresponding memory locations are in internal memory. To ensure this one first sorts a copy of the content of the memory based on the indices of the processors for which a given memory location will be operand. Then the computation can be performed by scanning through the sorted list, performing the computation for each of the processors, and writing the result to disk. Finally, the results can be sorted back according to the indices the processors would write them to. As only $O(1)$ sorts and scans are done in such a process, the simulation only uses $O(\text{sort}(N))$ I/Os as required.

A large number of $O(\text{sort}(N))$ I/O graph algorithms are developed in [40, 42] using the PRAM technique and the external list ranking algorithm. These includes algorithms for expression tree evaluation, centroid decomposition, least common ancestor, minimum spanning tree verification, connected components, minimum spanning forest, biconnected components, ear decomposition, and a number of problems on planar s,t-graphs.

4.3 I/O Complexity of OBDD Manipulation

Many problems in digital-systems design and verification, mathematical logic, concurrent system design and artificial intelligence can be expressed and solved in terms of boolean functions [33]. The efficiency of such solutions depends on the data structures used to represent the boolean functions and on the algorithms used to manipulate these data structures. Ordered Binary-Decision Diagrams (OBDDs) [32, 33] are the state-of-the-art data structure for boolean function manipulation and they have been successfully used to solve problems from all of the above mentioned areas. There exist implementations of OBDD software packages for a number of sequential and parallel machines [17, 31, 101, 102]. Even though there exist very different sized OBDD representations of the same boolean function, OBDDs in real applications tend to be very large. In [17] for example OBDDs of Gigabyte size are manipulated in order to verify logic circuit designs, and researchers in this area would like to be able to manipulate OBDDs orders of magnitude larger.

An OBDD is a *branching program* with some extra constraints. A branching program is a directed acyclic graph with one root, whose leaves (sinks) are labeled with boolean constants. The non-leaves are labeled with boolean variables and have two outgoing edges labeled 0 and 1, respectively. If a vertex has label x_i we say that it has index i. If f is the function

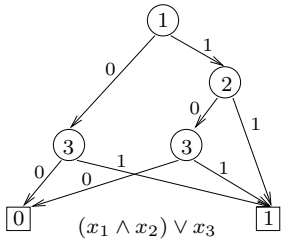


Figure 4.4: Example of OBDD.

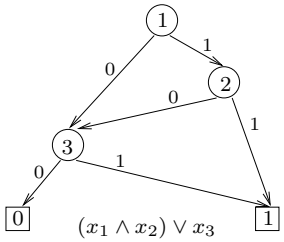


Figure 4.5: Reduced OBDD.

represented by the branching program, an evaluation of $f(a_1, \ldots, a_n)$ starts at the root and follows for a vertex labeled x_i the outgoing edge with label a_i. The label of the sink reached in this way equals $f(a_1, \ldots, a_n)$. An OBDD is a branching program for which an ordering of the variables in the vertices is fixed. For simplicity one normally assumes that this ordering is the natural one, $x_1, \ldots, x_n$. If a vertex with label x_j is a successor of a vertex with label x_i, the condition $j > i$ has to be fulfilled. An example of an OBDD is shown in Figure 4.4. An OBDD representing a boolean function of n variables can be of size 2^n, and different variable orderings can lead to representations of different size. There exist several algorithms (using heuristics) for choosing a variable-ordering that minimizes the OBDD-representation of a given function [88, 112].

In [32] Bryant proved that for a given variable ordering and a given boolean function there is exactly one OBDD—called the reduced OBDD —of minimal size. The OBDD in Figure 4.5 is the reduced version of the OBDD in Figure 4.4. As the reduced OBDD is canonical it is trivial to decide if two *reduced* OBDDs represent the same function or if a given *reduced* OBDD represents a satisfiable function. Bryant designed two fundamental operations on OBDDs—the *reduce* and the *apply* operation. The reduce operation computes the reduced version of a given OBDD, and the apply operation takes the OBDD representations of two functions and computes the representation of the function formed by combining them with a given binary logical operator. These two operations are (almost) all one needs in algorithms manipulation boolean function. Bryants algorithm for reducing an OBDD G with $|G|$ vertices requires $O(|G| \log |G|)$ time. Later algorithms running in $O(|G|)$ time have been developed [33, 118]. Bryant also gave an $O(|G_1| \cdot |G_2|)$ time algorithm for using the apply operation on two OBDDs of size $|G_1|$ and $|G_2|$. Later several authors have given alternative apply algorithms running in the same time bound [17, 101, 102].

One example of an application of OBDDs is given in Figure 4.6, where we wish to verify that the shown circuit fulfills its specification, that is, that it computes the desired function

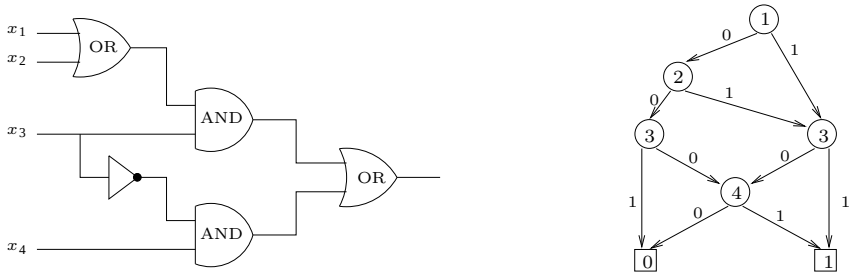


Figure 4.6: Verifying that a circuit computes the function $(\overline{x}_3 \wedge x_4) \vee (x_1 \wedge x_3) \vee (x_3 \wedge x_2)$.

$(\overline{x}_3 \wedge x_4) \vee (x_1 \wedge x_3) \vee (x_3 \wedge x_2)$. We can do so by building a reduced OBDD for the circuit and one for the specification. If these two OBDDs are equal we have verified the circuit. The OBDDs can easily be build in an incremental way using the apply operation, and it turns out that the circuit in Figure 4.6 indeed computes the given function (building the OBDDs for the circuit and the function both result in the OBDD given in Figure 4.6). This general method is used to verify even very large VLSI layouts, and it is not unusual to manipulated OBDDs of up to Gigabyte size in such applications [17]. Researchers in the area would like to be able to manipulate orders of magnitude larger OBDDs, but the limit on the size of the problem instances one has been able to solve in practice has generally been determined by the ability to find representations that fit in internal memory.

In [OBDD] we analyze the I/O performance of known OBDD-manipulation algorithms and develop new I/O-efficient algorithms. We show that in analogy with other internal-memory graph algorithms all existing reduce and apply algorithms have a poor worst-case performance, that is, they use $\Omega(|G|)$ and $\Omega(|G_1| \cdot |G_2|)$ I/Os, respectively. We develop new algorithms which use $O(\text{sort}(N))$ and $O(\text{sort}(|G_1| \cdot |G_2|))$ I/Os, respectively. Finally, we show that for a special class of algorithms, which include all existing reduce algorithms, $\Omega(\text{perm}(|G|))$ is a lower bound on the number of I/Os needed to reduce an OBDD of size $|G|$. Below we sketch the ideas behind these results for the reduce operation.

4.3.1 The Reduce Operation

In [32] Bryant proved that iterated use of the following two reduction rules on a OBDD with at most one 0-sink and one 1-sink yields the reduced OBDD: 1) If the two outgoing edges of vertex v lead to the same vertex w, then eliminate vertex v by letting all edges leading to v lead directly to w. 2) If two vertices v and w labeled with the same variable have the same 1-successor and the same 0-successor, then merge v and w into one vertex. For example, using rule two on one of the vertices in Figure 4.4 with index three yields the reduced OBDD in Figure 4.5.

All the reduction algorithms reported in the literature [17, 32, 33, 101, 102] basically work in the same way. They process the vertices level by level from the sinks to the root, and assign a (new) unique integer label to each unique sub-OBDD root. Processing a level, under the assumption that all lower levels have already been processed, is done by looking at the new labels of the children of the vertices on the level in question, checking if the reduction rules can be used, and assigning new labels to the vertices.

It is fairly easy to realize that all the existing reduce algorithms perform poorly in an I/O environment, that is, that they all use $O(|G|)$ I/Os in the worst case. The main reason is analogous to the reason why the internal-memory algorithms for the list-ranking and circuit evaluation problems perform poorly. The problem is the requirement to visit the children during the processing of a level of vertices. As there is no "nice" pattern in the way the children are visited (mainly because a vertex can have large fan-in), the algorithms can in the worst case be forced to do an I/O every time a child is visited. This is very similar to the problem encountered in time-forward processing. In [OBDD] we show that similar problems arise when performing the apply operation. Actually, we show that the poor performance can occur even if we assume that the OBDDs are initially blocked in some natural way, e.g. in a depth-first, breadth-first or levelwise manner. When we e.g. say that an OBDD is levelwise blocked, we mean that any given disk block only contains vertices from one level of the OBDD.

The main idea in the new reduce algorithm given in [OBDD] is very similar to the time-

forward processing algorithm in [Buffer]; When a vertex is given a new label we "inform" all its immediate predecessors about it in a "lazy" way using a priority queue. Thus the algorithm basically works like the previously know algorithms, except that every time a vertex v is given a new label, an element is inserted in an external priority queue for each of the immediate predecessors of v. The elements in the priority queue are ordered according to level and number of the "receiving" vertex. Thus when one on a higher level wants to know the new labels of children of vertices on the level, one can simply perform deletemin operations on the queue until all elements on the level in question have been obtained. This way the potentially expensive "pointer chasing" is avoided. That the algorithm overall uses $O(\text{sort}(|G|))$ I/Os again basically follows from the I/O-bounds on the priority queue operations.

The $\Omega(\text{perm}(N))$ I/O lower bound on the reduce operation is proved using techniques similar to the ones used in [42]. We define a variant of the proximate neighbors problem discussed earlier called the *split proximate neighbors problem* (SPN). In this problem we are as previously given N elements with a key each, such that precisely two elements have the same key. Furthermore, we require that the keys of the first $N/2$ elements (and consequently also the last $N/2$) are distinct, and that the keys of the first $N/2$ elements are sorted. Following the proof of the lower bound on the proximate neighbors problem, we can prove that an algorithm for the SPN problem must be capable of performing $(N!)^{1/3}$ permutations and thus we obtain an $O(\text{perm}(N))$ lower bound.

The general idea in the reduce lower bound proof is then a reduction of SPN to the reduce problem. However, unlike SPN where all elements are present somewhere in internal or external memory throughout the life of an algorithm for the problem, vertices may disappear and new ones may be created during a reduce operation. In the extreme case all vertices of an input OBDD are removed by a reduce algorithm and replaced by one (sink) vertex. Therefore we need to restrict our attention to a specific class of reduction algorithms in order to prove the lower bound. Intuitively the class we consider consists of all algorithms that work by assigning new labels to vertices and then check if the reduction rules can be used on a vertex by loading its two children to obtain their new labels (the class is defined precisely in [OBDD] using a pebble game). All known reduction algorithms belong to the class. Restricting the attention to this special class of algorithms the reduction now works as sketched on Figure 4.7, where it is shown how the elements in an SPN instance are encoded in the lowest levels of an OBDD. The marked vertices are the vertices containing elements from the instance, and the vertices to the left of the vertical dotted line contain the first half of the elements. Vertices containing elements with the same key are connected with an edge. Since we only consider algorithms from the special class, we know that at some point during the reduce algorithm

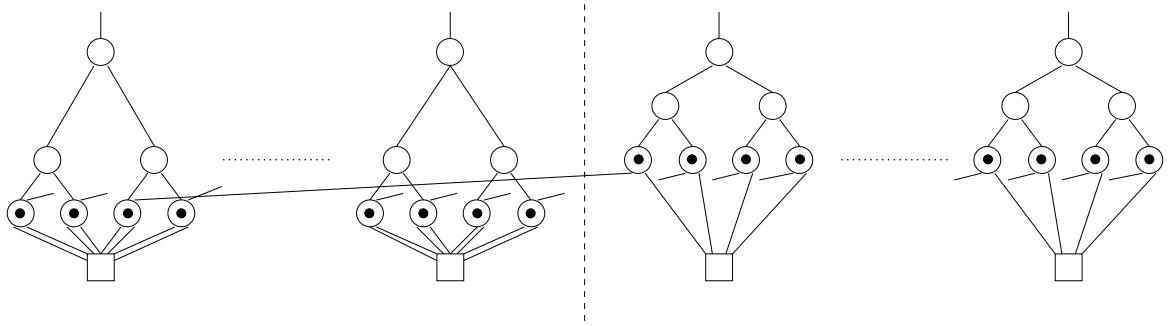


Figure 4.7: Sketch of how a SPN problem is reduced to the problem of reducing an OBDD.

the two vertices containing an SPN element with the same key will have to be in internal memory at the same time. Therefore, we can construct an algorithm for SPN from a reduce algorithm, and thus obtain a $O(\text{perm}(N))$ I/O lower bound on the latter.

4.4 Bibliographic Notes and Summary of our Contributions

Chiang et al. [42] proved lower bounds and used PRAM simulation and a list ranking algorithm to design efficient algorithms for a large number of graph problems. Some of the results are given in more detail in [40]. They gave several $O(\mathrm{sort}(N))$ I/O list ranking algorithms based on the framework used to design parallel algorithms for the problem. The difference between these algorithms is the way the independent set is produced. They gave a simple randomized algorithm based on an approach used in [8] and two deterministic algorithms that work by producing a 3-coloring of the list. Their deterministic algorithms only work under some (unrestrictive in practice) restriction on the block and memory size. The first deterministic algorithm only works if $B = O(N/\log^{(t)} N)$ for some fixed $t > 0$. Here $\log^{(t)} N$ is the iterated logarithm. Put another way, B should not be too large. The second algorithm relies on time-forward processing and the algorithm they develop for this problem using the distribution paradigm only works if $\sqrt{m/2}\log(m/2) \geq 2\log(2n/m)$, that is, $m = M/B$ should not be to small. By developing the priority queue in [Buffer] and obtaining the alternative time-forward processing algorithm we remove these restrictions. All the algorithms developed in [40, 42] mentioned in Section 4.2.2 rely on the algorithm for the list ranking problem. Thus by designing a list ranking (time-forward processing) algorithm without assumptions on the size of B and M we also remove the assumptions from all of these algorithms. Finally, Chiang in [40] also describes how to use the priority queue to obtain algorithms for e.g. expression tree evaluation directly without using PRAM simulation or list ranking. Recently, Kumar has also developed some I/O graph algorithms using external data structures [83].

Bryant introduced OBDDs and developed algorithms which traverse the OBDDs in a depth-first way [32, 33]. Later algorithms working in a breadth-first manner were developed [101]. In [17] and [102] it was realized that the depth-first or breadth-first traversals are one main reason why the traditional algorithms do not perform well when the OBDDs are too large to fit in internal memory, and new levelwise algorithms were developed. The general idea in these algorithms is to store the OBDDs in a level-blocked manner, and then try to access the vertices in a level-by-level pattern. Previous algorithms did not explicitly block the OBDDs. In [17, 102] speed-ups of several hundreds compared to the "traditional" algorithms are reported using this idea. Recently, speedups obtained by considering blocking issues between levels of internal memory have been reported in [80]. In [OBDD] we show that all the known reduce and apply algorithms in the worst case use $O(|G|)$ and $O(|G_1| \cdot |G_2|)$ I/Os, respectively, *even* the algorithms specifically designed with I/O in mind. One natural question to ask is of course why experiments with these algorithms then show so huge speedups compared to the traditional algorithms. The answer is partly that the traditional depth-first and breadth-first algorithms behave so poorly with respect to I/O that just considering I/O-issues, and actually try to block the OBDDs and access them in a "sequential" way, leads to large speedups. However, we believe that one major reason for the experimental success in [17, 102] is that the OBDDs in the experiments roughly are of the size of the internal memory of the machines used. This means that one level of the OBDDs fits in internal memory, which again explains the good performance because the worst case behavior occurs when one level of the OBDD does not fit in internal memory. In [OBDD] an $O(\mathrm{perm}(|G|))$ I/O lower bound on the reduce operation is proved under the assumption that the reduce algorithm belongs to a special class of algorithms. The class is defined using a graph pebble game. We believe that in order to state precisely in which model of computation the lower bounds proved in [42] hold one needs to consider a similar game. The $O(\mathrm{perm}(|G|))$ lower bound applies even if we know that the OBDD is blocked in a breadth-first, depth-first manner or level blocked (like in the algorithms in [17, 102]) manner. It also applies if the OBDD is what we call minimal-pair blocked, which is the intuitively best blocking strategy for the class of algorithms considered, namely a blocking that minimizes the number of pairs of vertices connected with an edge which are not in the same block. We believe that the $O(\mathrm{sort}(|G|))$ I/O reduce and $O(\mathrm{sort}(|G_1| \cdot |G_2|))$ I/O apply algorithms developed in [OBDD] are of practical interest due to relatively small constants in the asymptotic bounds.

Chapter 5

Conclusions

Beware of bugs in the above code; I have only proved it correct, not tried it
D. Knuth

Designing algorithms with good I/O performance is crucial in a number of areas where the I/O-bottleneck is becoming increasingly important. In the preceeding chapters we have surveyed important results in the area of I/O algorithms. However, while a number of basic techniques for designing efficient external-memory algorithms now exist, and while there exist efficient external algorithms for a number of basic and important computational geometry and graph problems, there are still a lot of open questions. The young field of I/O-efficient computation is to a large extent still wide open. Even though the experimental results reported so far are encouraging, a major future goal is to investigate the practical merits of the developed I/O algorithms.

5.1 Concluding Remarks on our Contributions

The work in this thesis has demonstrated that it can be very fruitful to take a data structure approach to the design of I/O algorithms. We have developed a number of efficient external data structures and shown how they can be used to develop alternative, new and improved I/O algorithms. We have shown how external data structures can be used as an alternative to normal internal data structures in plane-sweep algorithms and thus directly obtained efficient algorithms for a number of problems involving geometric objects. These algorithms are alternatives to other algorithms developed using special I/O design techniques. Exemplified by external fractional cascading, we have shown how the data structure approach can be a catalyzer for the development of new algorithms through the development of external-memory versions of known internal-memory techniques. Finally, in the case of external graph algorithms we have seen how the development of external data structures almost immediately can lead to improvements of previously known algorithms.

Part II

Papers

Chapter 6

The Buffer Tree: A New Technique for Optimal I/O-Algorithms

The Buffer Tree: A New Technique for Optimal I/O-Algorithms*

Lars Arge†

BRICS‡
Department of Computer Science
University of Aarhus
Aarhus, Denmark

August, 1996

Abstract

In this paper we develop a technique for transforming an internal-memory tree data structure into an external-memory structure. We show how the technique can be used to develop a search tree like structure, a priority queue, a (one-dimensional) range tree and a segment tree, and give examples of how these structures can be used to develop efficient I/O algorithms. All our algorithms are either extremely simple or straightforward generalizations of known internal-memory algorithms—given the developed external data structures. We believe that algorithms relying on the developed structure will be of practical interest due to relatively small constants in the asymptotic bounds.

1 Introduction

In the last few years, more and more attention has been given to Input/Output (I/O) complexity of existing algorithms and to the development of new I/O-efficient algorithms. This is due to the fact that communication between fast internal memory and slower external memory is the bottleneck in many large-scale computations. The significance of this bottleneck is increasing as internal computation gets faster, and especially as parallel computing gains popularity [107]. Currently, technological advances are increasing CPU speed at an annual rate of 40-60% while disk transfer rates are only increasing by 7-10% annually [113].

A lot of work has already been done on designing external memory versions of known internal-memory data structures (e.g. [16, 67, 74, 79, 82, 110, 119, 121, 130]), but practically all of these data structures are designed to be used in on-line settings, where queries should be

*This paper is a revised and extended version of BRICS report 94-16. An extended abstract version was presented at the Fourth Workshop on Algorithms and Data Structures (WADS'95)

†This work was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 7141 (project ALCOM II) and by Aarhus University Research Foundation. Part of the work was done while a Visiting Scholar at Duke University. Email: large@daimi.aau.dk

‡Acronym for Basic Research in Computer Science, a Center of the Danish National Research Foundation.

answered immediately and within a good worst case number of I/Os. This effectively means that using these structures to solve off-line problems yields non-optimal algorithms because they are not able to take full advantage of the large internal memory. Therefore a number of researchers have developed techniques and algorithms for solving large-scale off-line problems without using external memory data structures [5, 42, 67].

In this paper we develop external data structures that take advantage of the large main memory. This is done by only requiring good amortized performance of the operations on the structures, and by allowing search operations to be batched. The data structures developed can then be used in simple and I/O-efficient algorithms for computational geometry and graph problems. As pointed out in [42] and [67] problems from these two areas arise in many large-scale computations in e.g. object-oriented, deductive and spatial databases, VLSI design and simulation programs, geographic informations systems, constraint logic programming, statistics, virtual reality systems, and computer graphics.

1.1 I/O Model and Previous Results

We will be working in an I/O model introduced by Aggarwal and Vitter [5]. The model has the following parameters:

$$\begin{aligned} N &= \#\text{ of elements in the problem instance;} \\ M &= \#\text{ of elements that can fit into main memory;} \\ B &= \#\text{ of elements per block,} \end{aligned}$$

where $M < N$ and $1 \leq B \leq M/2$. The model captures the essential parameters of many of the I/O-systems in use today, and depending on the size of the data elements, typical values for workstations and file servers are on the order of $M = 10^6$ or 10^7 and $B = 10^3$. Large-scale problem instances can be in the range $N = 10^{10}$ to $N = 10^{12}$.

An I/O operation in the model is a swap of B elements from internal memory with B consecutive elements from external memory. The measure of performance we consider is the number of such I/Os needed to solve a given problem. Internal computation is for free. As we shall see shortly the quotients N/B (the number of blocks in the problem) and M/B (the number of blocks that fit into internal memory) play an important role in the study of I/O-complexity. Therefore, we will use n as shorthand for N/B and m for M/B. Furthermore, we say that an algorithm uses a linear number of I/O operations if it uses at most $O(n)$ I/Os to solve a problem of size N. In [133] the I/O model is extended with a parameter D. Here the external memory is partitioned into D distinct disk drives, and if no two blocks come from the same disk, D blocks can be transferred per I/O. The number D of disks range up to 10^2 in current disk arrays.

Early work on I/O algorithms concentrated on algorithms for sorting and permutation-related problems in the single disk model [5], as well as in the extended version of the I/O-model [98, 99, 131, 133]. External sorting requires $\Theta(n \log_m n)$ I/Os,[1] which is the external memory equivalent of the well-known $\Theta(N \log N)$ time bound for sorting in internal memory. Note that this means that $O(\frac{\log_m n}{B})$ is the I/O bound corresponding to the $O(\log_2 N)$ bound on the operations on many internal-memory data structures. More recently external-memory researchers have designed algorithms for a number of problems in different areas.

[1] We define for convenience $\log_m n = \max\{1, (\log n)/(\log m)\}$.

Most notably I/O-efficient algorithms have been developed for a large number of computational geometry [15, 67] and graph problems [42]. In [13] a general connection between the comparison-complexity and the I/O-complexity of a given problem is shown in the "comparison I/O model" where comparison of elements is the only allowed operation in internal memory.

1.2 Our Results

In this paper we develop a technique for transforming an internal-memory tree data structure into an external memory data structure. We use our technique to develop a number of external memory data structures, which in turn can be used to develop optimal algorithms for problems from the different areas previously considered with respect to I/O-complexity. All these algorithms are either extremely simple or straightforward generalizations of known internal-memory algorithms—given the developed external data structures. This is in contrast to the I/O-algorithms developed so far, as they are all very I/O-specific. Using our technique we on the other hand manage to isolate all the I/O-specific parts of the algorithms in the data structures, which is nice from a software engineering point of view. Ultimately, one would like to give the task of transforming an ordinary internal-memory algorithm into a good external memory one to the compiler. We believe that our technique and the developed structures will be useful in the development of algorithms for other problems in the mentioned areas as well as in other areas. Examples of this can be found in [12, 15, 40]. More specifically, the results in this paper are the following:

Sorting: We develop a simple dynamic tree structure (*The Buffer Tree*) with operations *insert*, *delete* and *write*. We prove amortized I/O bounds of $O(\frac{\log_m n}{B})$ on the first two operations and $O(n)$ on the last. Using the structure we can sort N elements with the standard tree-sort algorithm in the optimal number of I/Os. This algorithm is then an alternative to the sorting algorithms developed so far. The algorithm is the first I/O-algorithm that does not need all the elements to be present by the start of the algorithm.

Graph Problems: We extend the buffer tree with a *deletemin* operation in order to obtain an external-memory *priority queue*. We prove an $O(\frac{\log_m n}{B})$ amortized bound on the number of I/Os used by this operation. Using the structure it is straightforward to develop an extremely simple algorithm for "circuit-like" computations as defined in [42]. This algorithm is then an alternative to the "time-forward processing technique" developed in the same paper. The time-forward processing technique only works for large values of m, while our algorithm works for all m. In [42] the time-forward processing technique is used to develop an efficient I/O algorithm for external-memory list-ranking, which in turn is used to develop efficient algorithms for a large number of graph-problems.[2] All these algorithms thus inherit the constraint on m and our new algorithm removes it from all of them. Finally, the structure can of course also be used to sort optimally.

Computational Geometry Problems: We also extend the buffer tree with a *batched rangesearch* operation in order to obtain an external (one-dimensional) *range tree* structure. We prove an $O(\frac{\log_m n}{B}+r)$ amortized bound on the number of I/Os used by the operation. Here r is the number of *blocks* reported. Furthermore, we use our technique to develop an external version of the *segment tree* with operations *insert/delete* and *batched search* with the same

[2] Expression tree evaluation, centroid decomposition, least common ancestor, minimum spanning tree verification, connected components, minimum spanning forest, biconnected components, ear decomposition, and a number of problems on planar *st*-graphs.

I/O bounds as the corresponding operations on the range tree structure. The two structures enable us to solve the orthogonal line segment intersection, the batched range searching, and the pairwise rectangle intersection problems in the optimal number of I/O operations. We can solve these problems with exactly the same plane-sweep algorithms as are used in internal memory. As mentioned, large-scale computational geometry problems arise in many areas. The three intersection reporting problems mentioned especially arise in VLSI design and are certainly large-scale in such applications. The pairwise rectangle intersection problem is of special interest, as it is used in VLSI design rule checking [26]. Optimal algorithms for the three problems are also developed in [67], but as noted earlier these algorithms are very I/O-specific, while we manage to "hide" all the I/O-specific parts in the data structures and use the known internal-memory algorithms. A note should also be made on the fact that the search operations are *batched*. Batched here means that we will not immediately get the result of a search operation. Furthermore, parts of the result will be reported at different times when other operations are performed. This suffices in the plan-sweep algorithms we are considering, as the sequence of operations done on the data structure in these algorithms does not depend on the results of the queries in the sequence. In general, problems where the whole sequence of operations on a data structure is known in advance, and where there is no requirement on the order in which the queries should be answered, are known as *batched dynamic problems* [55].

As mentioned some work has already been done on designing external versions of known internal dynamic data structures, but practically all of it has been done in the I/O model where the size of the internal memory equals the block size. The motivation for working in this model has partly been that the goal was to develop structures for an on-line setting, where answers to queries should be reported immediately and within a good worst-case number of I/Os. This means that if we used these structures to solve the problems we consider in this paper, we would not be able to take full advantage of the large main memory. Consider for example the well-known B-tree [21, 51, 82]. On such a tree one can do an *insert* or *delete* operation in $O(\log_B n)$ I/Os and a *rangesearch* operation in $O(\log_B n + r)$ I/Os. This means that using a B-tree as sweep-structure in the standard plane-sweep algorithm for the orthogonal line segment intersection problem results in an algorithm using $O(N \log_B n + r)$ I/Os. But an optimal solution for this problem only requires $O(n \log_m n + r)$ I/Os [13, 67]. For typical systems B is less than m so $\log_B n$ is larger than $\log_m n$, but more important, the B-tree solution will be slower than the optimal solution by a factor of B. As B typically is on the order of thousands this factor is crucial in practice. The main problem with the B-tree in this context is precisely that it is designed to have a good worst-case on-line search performance. In order to take advantage of the large internal memory, we on the other hand use the fact that we only are interested in the *overall* I/O use of the algorithm for an *off-line* problem—that is, in a good amortized performance of the involved operations—and sometime even satisfied with *batched* search operations.

As mentioned we believe that one of the main contributions of this paper is the development of external-memory data structures that allow us to use the normal internal-memory algorithms and "hide" the I/O-specific parts in the data structures. Furthermore, we believe that our structures will be of practical interest due to relatively small constants in the asymptotic bounds. We hope in the future to be able to implement some of the structures in the transparent parallel I/O environment (TPIE) developed by Vengroff [126]. Results of experiments on the practical performance of several algorithms developed for the I/O model

are reported in [40, 41, 129].

The main organization of the rest of this paper is the following: In the next section we sketch our general technique. In section 3 we then develop the basic buffer tree structure which can be use to sort optimally, and in section 4 and 5 we extend this structure with a deletemin and batched rangesearch operation, respectively. The external version of the segment tree is developed in section 6. Using techniques from [99] all the developed structures can be modified to work in the D-disk model—that is, the I/O bounds can be divided by D. We discuss such an extension in Section 7. Finally, conclusions are given in Section 8.

2 A Sketch of the Technique

In this section we sketch the main ideas in our transformation technique. When we want to transform an internal-memory tree data structure into an external version of the structure, we start by grouping the (binary) nodes in the structure into super-nodes with fan-out $\Theta(m)$—that is, fan-out equal to the number of blocks that fits into internal memory. We furthermore group the leaves together into blocks obtaining an $O(\log_m n)$ "super-node height". To each of the super-nodes we then assign a "buffer" of size $\Theta(m)$ blocks. No buffers are assigned to the leaves. As the number of super-nodes on the level just above the leaves is $O(n/m)$, this means that the total number of buffers in the structure is $O(n/m)$.

Operations on the structure—updates as well as queries—are then done in a "lazy" manner. If we for example are working on a search tree structure and want to insert an element among the leaves, we do not right away search all the way down the tree to find the place among the leaves to insert the element. Instead, we wait until we have collected a block of insertions (or other operations), and then we insert this block in the buffer of the root. When a buffer "runs full" the elements in the buffer are "pushed" one level down to buffers on the next level. We call this a *buffer-emptying process*. Deletions or other and perhaps more complicated updates, as well as queries, are basically done in the same way as insertions. This means that we can have several insertions and deletions of the same element in the tree, and we therefore time stamp the elements when we insert them in the top buffer. It also means that the queries get batched in the sense that the result of a query may be generated (and reported) lazily by several buffer-emptying processes.

The main requirement needed to show the I/O bounds mentioned in the introduction is that we should be able to empty a buffer in $O(m + r')$ I/O operations. Here r' is the number of blocks reported by query operations in the emptied buffer. If this is the case, we can do an amortization argument by associating a number of credits to each block of elements in the tree. More precisely, each block in the buffer of node x must hold $O($the height of the tree rooted at $x)$ credits. As we only do a buffer-emptying process when the buffer runs full, that is, when it contains $\Theta(m)$ blocks, and as we can charge the r'-term to the queries that cause the reports, the blocks in the buffer can pay for the emptying-process as they all get pushed one level down. On insertion in the root buffer we then have to give each update element $O(\frac{\log_m n}{B})$ credits and each query element $O(\frac{\log_m n}{B} + r)$ credits, and this gives us the desired bounds. Of course we also need to consider e.g. rebalancing of the transformed structure. We will return to this, as well as the details in other operations, in later sections. Another way of looking at the above amortization argument is that we touch each block a constant number of times on each level of the structure. Thus the argument still holds if we can empty a buffer in a linear number of I/Os in the number of elements in the buffer. In later

sections we will use this fact several times when we show how to empty a buffer containing x blocks, where x is bigger than m, in $O(m+x) = O(x)$ I/Os. Note also that the amortization argument works as long as the fan-out of the super-nodes is $\Theta(m^c)$ for $0 < c \leq 1$, as the super-node height remains $O(\log_m n)$ even with this smaller fan-out. We will use this fact in the development of the external segment tree.

3 The Buffer Tree

In this section we will develop the basic structure—which we call the *buffer tree*—and only consider the operations needed in order to use the structure in a simple sorting algorithm. In later sections we then extend this basic structure in order to obtain an external priority queue and an external (one-dimensional) range tree.

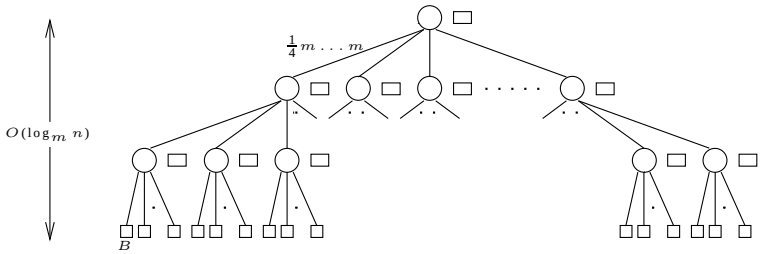


Figure 1: The buffer tree.

The buffer tree is an (a, b)-tree [73] with $a = m/4$ and $b = m$, extended with a buffer in each node. In such a tree all nodes except for the root have fan-out between $m/4$ and m, and thus the height of the tree is $O(\log_m n)$. The buffer tree is pictured in Figure 1. As discussed in section 2 we do the following when we want to do an update on the buffer tree: We construct a new element consisting of the element to be inserted or deleted, a time stamp, and an indication of whether the element is to be inserted or deleted. When we have collected B such elements in internal memory, we insert the block in the buffer of the root. If the buffer of the root still contains less than $m/2$ blocks we stop. Otherwise, we empty the buffer. The buffer-emptying process is described in Figure 2 and 5. We define *internal nodes* to be all nodes which do not have leaves as children, and the basic part of the process which is used on these nodes (corresponding to the discussion in the last section) is given in Figure 2. Note that the buffer-emptying process is only done recursively on internal nodes.

- Load the partitioning (or routing) elements of the node into internal memory.
- Repeatedly load (at most) $m/2$ blocks of the buffer into internal memory and do the following:
  1. Sort the elements from the buffer in internal memory. If two equal elements—an insertion and a deletion—"meet" during this process, and if the time stamps "fit", then the two elements annihilates.
  2. Partition the elements according to the partitioning elements and output them to the appropriate buffers one level down (maintaining the invariant that at most one block in a buffer is non-full).
- If the buffer of any of the children now contains more than $\frac{1}{2}m$ blocks, and if the children are internal nodes, then recursively apply the emptying-process on these nodes.

Figure 2: The buffer-emptying process on internal nodes.

Rebalancing after inserting an element below v:

```
DO v has b+1 children ->
  IF v is the root ->
    let x be a new node and make v its only child
  ELSE
    let x be the parent of v
  FI
  Let v' be a new node
  Let v' be a new child of x immediately after v
  Split v:
    Take the rightmost ⌈(b+1)/2⌉  children
     away from v and make them children of v'.
  Let v=x
OD
```

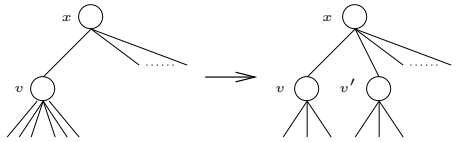


Figure 3: Insert in (a, b) tree.

Rebalancing after deleting an element below v:

```
DO v has a-1 children AND
   v' has less than a+t+1 children ->
  Fuse v and v':
    Make all children of v' children of v
  Let v=x
  Let v' be a brother of x
  IF x does not have a brother (x is the root)
     AND x only has one child ->
    Delete x
    STOP
  FI
  Let x be the parent of v.
OD
(* either v has more than a children and we
 are finished, or we can finish by sharing *)
IF v has a-1 children ->
  Share:
    Take s children away from v' and
    make them children of v.
FI
```

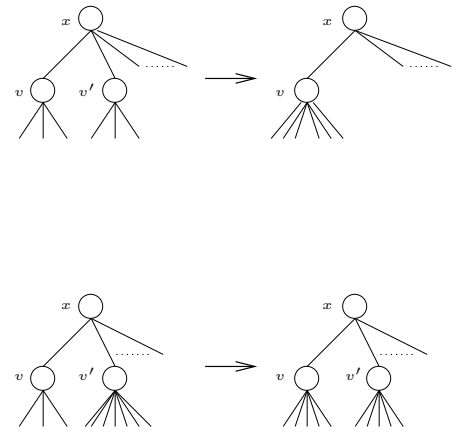


Figure 4: Delete from (a, b)-tree ($s = \lceil((b/2 - a) + 1)/2\rceil$ and $t = (b/2 - a) + s - 1$).

Only after finishing *all* buffer-emptying processes on internal nodes, we empty the buffers of the *leaf nodes* as we call the nodes which are not internal. That the buffer-emptying process on an internal node can be done in a linear number of I/Os as required is easily realized: The elements are loaded and written ones, and at most $O(m)$ I/Os are used on writing non-filled blocks every time we load $m/2$ blocks. Note that the cost of emptying a buffer containing $o(m)$ blocks remains $O(m)$, as we distribute the elements to $\Theta(m)$ children.

The emptying of a leaf buffer is a bit more complicated as we also need to consider rebalancing of the structure when we empty such a buffer. The algorithm is given in Figure 5. Basically the rebalancing is done precisely as on normal (a, b)-trees [73]. After finding the position of a new element among the elements in the leaves of an (a, b)-tree, the rebalancing

- As long as there is a leaf node v with a full buffer (size greater than $m/2$ blocks) do the following (x is the number of leaves of v):
  1. Sort the elements in the buffer of v with an optimal I/O sorting algorithm and remove "matching" insert/delete elements.
  2. Merge the sorted list with the sorted list of elements in the leaves of v while removing "matching" insert/delete elements.
  3. If the number of blocks of elements in the resulting list is **smaller** than x do the following:
     (a) Place the elements in sorted order in the leaves of v.
     (b) Add "dummy-blocks" until v has x leaves and update the partition elements in v.
  4. If the number of blocks of elements in the resulting list is **bigger** than x do the following:
     (a) Place the x smallest blocks in the leaves of v and update the partition elements of v accordingly.
     (b) Repeatedly insert one block of the rest of the elements and rebalance.
- Repeatedly delete one dummy block and rebalance—while performing a buffer-emptying process on the relevant nodes involved in a rebalance operation (v' of Figure 4) before the operation is done (if v' is a leaf node its buffer is emptied as described above).

  If the delete (or rather the buffer-emptying processes done as a result of it) results in any leaf buffer becoming full, these buffers are emptied as described above before the next dummy block is deleted.

Figure 5: Emptying the buffers of the leaf nodes.

is done by a series of "splits" of node in the structure. We give the algorithm in Figure 3. Similarly, after deleting an element in a leaf the rebalancing is accomplished by a series of node "fusions" possibly ending with a node "sharing". The algorithm is given in Figure 4. In the buffer tree case we need to modify the delete rebalancing algorithm slightly because of the buffers. The modification consists of doing a buffer-emptying process before every rebalance operation. More precisely, we do a buffer-emptying process on v' in Figure 4 when it is involved in a fuse or share rebalancing operation. This way we can do the actual rebalancing operation as normally, without having to worry about elements in the buffers. This is due to the fact that our buffer-emptying process on internal nodes maintains the invariant that if the buffer of a leaf node runs full then all nodes on the path to the root have empty buffers. Thus when we start rebalancing the structure (insert and delete the relevant blocks) after emptying all the leaf buffers (Figure 5), all nodes playing the role of v in split, fuse or share rebalance operations already have empty buffers. Also if the emptying of the buffer of v' results in a leaf buffer running full, the invariant will be fulfilled because all nodes on the path from v's parent x to the root have empty buffers. Note that the reason for not doing buffer-emptying processes on leaf nodes recursively, is to prevent different rebalancing operations from interfering with each other. This is also the reason for the special way of handling deletes with dummy blocks; while deletion of a block may result in several buffer-emptying processes, this is not the case for insertions as no buffer-emptying process are necessary in this rebalancing algorithm.

We can now prove our main theorem.

Theorem 1 *The total cost of an arbitrary sequence of N intermixed insert and delete operation on an initially empty buffer tree is $O(n \log_m n)$ I/O operations, that is, the amortized cost of an operation is $O(\frac{\log_m n}{B})$ I/Os.*

Proof: As discussed in Section 2 the total cost of all the buffer-emptying processes on *internal* nodes with *full* buffers is bounded by $O(n \log_m n)$ I/Os. This follows from the fact that one such process uses a linear number of I/Os in the number of blocks pushed one level down.

During the rebalancing operations we empty a number of *non-full* buffers using $O(m)$ I/Os, namely one for each rebalancing operation following a deletion of a block. Furthermore, it is easy to realize that the administrative work in a rebalancing operation—updating partitioning elements and so on—can be performed in $O(m)$ I/Os. In [73] it is shown that the number of rebalancing operations in a sequence of K updates on an initially empty (a, b)-tree is bounded by $O(K/(b/2 - a))$ if $b > 2a$. As we are inserting blocks in the $(m/4, m)$-tree underlying the buffer tree this means that the total number of rebalance operations in a sequence of N updates on the buffer tree is bounded by $O(n/m)$. Thus the total cost of the rebalancing is $O(n)$.

The only I/Os we have not counted so far are the ones used on emptying *leaf buffers*. The number of I/Os used on a buffer-emptying process on a leaf node is dominated by the sorting of the elements (Figure 5, step 1). As any given element will only once be involved in such a sorting the *total* number of I/Os used to empty leaf buffers is bounded by $O(n \log_m n)$. This proves the lemma. □

In order to use the transformed structure in a simple sorting algorithm, we need a empty/write operation that empties all the buffers and then reports the elements in the leaves in sorted order. The emptying of all buffers can easily be done just by performing a buffer-emptying process on all nodes in the tree—from the top. As emptying one buffer costs $O(m)$ I/Os amortized, and as the total number of buffers in the tree is $O(n/m)$, we have the following:

Theorem 2 *The amortized I/O cost of emptying all buffers of a buffer tree after performing N updates on it, and reporting all the remaining elements in sorted order, is $O(n)$.*

Corollary 1 *N elements can be sorted in $O(n \log_m n)$ I/O operations using the buffer tree.*

As mentioned the above result is optimal and our sorting algorithm is the first that does not require all the elements to be present by the start of the algorithm. In Section 7 we discuss how to avoid the sorting algorithm used in the buffer-emptying algorithm.

Before continuing to design more operations on the buffer tree a note should be made on the balancing strategy used. We could have used a simpler balancing strategy than the one presented in this section. Instead of balancing the tree bottom-up we can balance it in a top-down style. We can make such a strategy work, if we "tune" our constants (fan-out and buffer-size) in such a way that the maximal number of elements in the buffers of a subtree is guaranteed to be less that half the number of elements in the leaves of the subtree. If this is the case we can do the rebalancing of a node when we empty its buffer. More precisely, we can do a split, a fuse or a sharing in connection with the buffer-emptying process on a node, in order to guarantee that there is room in the node to allow all its children to fuse or split. In this way we can make sure that rebalancing will never propagate. Unfortunately, we have not been able to make this simpler strategy work when rangesearch operations (as discussed in Section 5) are allowed.

4 An External Priority Queue

Normally, we can use a search tree structure to implement a priority queue because we know that the smallest element in a search tree is in the leftmost leaf. The same strategy can be used to implement an external priority queue based on the buffer tree. There are a couple of problems though, because using the buffer tree we cannot be sure that the smallest element is in the leftmost leaf, as there can be smaller elements in the buffers of the nodes on the leftmost path. However, there is a simple strategy for performing a deletemin operation in the desired amortized I/O bound. When we want to perform a deletemin operation we simply do a buffer-emptying process on all nodes on the path from the root to the leftmost leaf. To do so we use $O(m \cdot \log_m n)$ I/Os amortized. After doing so we can be sure not only that the leftmost leaf consists of the B smallest elements, but also that (at least) the $\frac{1}{4}m \cdot B$ smallest elements in the tree are in the children (leaves) of the leftmost leaf. If we delete these elements and keep them in internal memory, we can answer the next $\frac{1}{4}m \cdot B$ deletemin operations without doing any I/Os. Of course we then also have to check insertions and deletions against the minimal elements in internal memory. This can be done in a straightforward way without doing extra I/Os, and a simple amortization argument gives us the following:

Theorem 3 *The total cost of an arbitrary sequence of N insert, delete and deletemin operations on an initially empty buffer tree is $O(n \log_m n)$ I/O operations, that is, the amortized cost of an operation is $O(\frac{\log_m n}{B})$ I/Os.*

Note that in the above result we use $m/4$ blocks of internal memory to hold the minimal elements. In some applications (e.g. in [15]) we would like to use less internal memory for the external priority queue structure. Actually, we can make our priority queue work with as little as $\frac{1}{4}m^{1/c}$ $(0 < c \leq 1)$ blocks of internal memory, by decreasing the fan-out and the size of the buffers to $\Theta(m^{1/c})$ as discussed in Section 2.

4.1 Application: Time-Forward Processing

As mentioned in the introduction a technique for evaluating circuits (or "circuit-like" computations) in external memory is developed in [42]. This technique is called time-forward processing. The problem is the following: We are given a bounded fan-in boolean circuit, whose description is stored in external memory, and want to evaluate the function computed by the network. It is assumed that the representation of the circuit is topologically sorted, that is, the labels of the nodes come from a total order $<$, and for every edge (v, w) we have $v < w$. Nothing is assumed about the functions in the nodes, except that they take at most $M/2$ inputs. Thinking of vertex v as being evaluated at "time" v motivates calling an evaluation of a circuit a time-forward processing. The main issue in such an evaluation is to ensure that when one evaluates a particular vertex one has the values of its inputs in internal memory.

In [42] an external-memory algorithm using $O(n \log_m n)$ I/Os is developed (here N is the number of nodes plus the number of edges). The algorithm uses a number of known as well as new I/O-algorithm design techniques and is not particularly simple. Furthermore, the algorithm only works for large values of m, more precisely it works if $\sqrt{m/2} \log(M/2) \geq 2 \log(2N/M)$. For typical machines this constraint will be fulfilled. Using our external priority queue however, it is obvious how to develop a simple alternative algorithm—without the

constraint on the value of m. When we evaluate a node v we simply send the result forward in time to the appropriate nodes, by inserting a copy of the result in the priority queue with priority w for all edges (v, w). We can then obtain the inputs to the next node in the topological order just by performing a number of deletemin operations on the queue. The $O(n \log_m n)$ I/O-bound follows immediately from Theorem 3.

In [42] a randomized and two deterministic algorithms for external-memory list ranking are developed. One of these algorithms uses time-forward processing and therefore inherits the constraint that m should not be too small. The other has a constraint on B not being to large (which in turn also results in a constraint on m not being to small). As mentioned, the list ranking algorithm is in turn used to develop efficient external algorithms for a number of problems. This means that by developing an alternative time-forward processing algorithm without the constraint on m, we have also removed the constraint from the algorithm for list ranking, as well as from a large number of other external-memory graph algorithms.

5 An External (one-dimensional) Range Tree Structure

In this section we extend the basic buffer tree with a rangesearch operation in order to obtain an external (one-dimensional) range tree structure.

Normally, one performs a rangesearch with x_1 and x_2 on a search tree by searching down the tree for the positions of x_1 and x_2 among the elements in the leaves, and then one reports all the elements between x_1 and x_2. However, the result can also be generated while searching down the tree, by reporting the elements in the relevant subtrees on the way down. This is the strategy we use on the buffer tree. The general idea in our rangesearch operation is the following: We start almost as when we do an insertion or a deletion. We make a new element containing the interval $[x_1, x_2]$ and a time stamp, and insert it in the tree. We then have to modify our buffer-emptying process in order to deal with the new rangesearch elements. The basic idea is that when we meet a rangesearch element in a buffer-empty process, we first determine whether x_1 and x_2 are contained in the same subtree among the subtrees rooted at the children of the node in question. If this is the case we just insert the element in the corresponding buffer. Otherwise we "split" the element in two—one for x_1 and one for x_2—and report the elements in the subtrees for which *all* elements in them are contained in the interval $[x_1, x_2]$. The splitting only occurs once and after that the rangesearch elements are pushed downwards in the buffer-emptying processes like insert and delete elements, while elements in the subtrees for which all the elements are in the interval are reported. As discussed in the introduction and Section 2 this means that the rangesearch operation gets batched.

In order to make the above strategy work efficiently we need to overcome several complications. One major complication is the algorithm for reporting all elements in a subtree. For several reasons we cannot just use the simple algorithm presented in Section 3, and empty the buffers of the subtree by doing a buffer-emptying process on all nodes and then report the elements in the leaves. The major reason is that the buffers of the tree may contain other rangesearch elements, and that we should also report the elements contained in the intervals of these queries. Also in order to obtain the desired I/O bound on the rangesearch operation, we should be able to report the elements in a tree in $O(n_a)$ I/Os, where n_a is the actual number of blocks in the tree, that is, the number of blocks used by elements which are not deleted by delete elements in the buffers of the tree. This number could be a low as zero.

However, if n_d is the number of blocks deleted by delete elements in the tree, we have that $n = n_a + n_d$. This means that if we can empty all the buffers in the tree—*and* remove all the delete elements—in $O(n)$ I/Os, we can charge the n_d part to the delete elements, adding $O(\frac{1}{B})$ to the amortized number of I/Os used by a delete operation (or put another way; we in total use $O(n)$ I/Os extra to remove *all* the delete elements).

In Subsection 5.1 we design an algorithm for emptying all the buffers of a (sub-) buffer tree in a linear number of I/Os. Our algorithm reports all relevant "hits" between rangesearch and normal elements in the tree. In Subsection 5.2 we then show precisely how to modify the buffer-emptying process on the buffer tree in order to obtain the efficient rangesearch operation.

5.1 Emptying all Buffers of an External Range Tree

In order to design the algorithm for emptying all buffers we need to restrict the operations on the structure. In the following we assume that we only try to delete elements from a buffer tree which were previously inserted in the tree. The assumption is natural (at least) in a batched dynamic environment. Having made this assumption we obtain the following useful properties: If d, i and s are matching delete, insert and rangesearch elements (that is, "$i = d$" and "i is contained in s"), and if we know that their time order is d, s, i (and that no other elements—especially not rangesearch elements—are between s and i in the time order), then we can report that i is in s and remove i and d. If we know that the time order is s, i (knowing that no element—especially not d—is between s and i in the time order), we can report that i is in s and interchange their time order. Similarly, if we know that the time order is d, s, we can again report that d is in s (because we know that there is an i matching d "on the other side of s") and interchange their time order.

We define a set of (insert, delete and rangesearch) elements to be in *time order representation* if the time order is such that all the delete elements are "older" than (were inserted before) all the rangesearch elements, which in turn are older than all the insert elements, *and* if the three groups of elements are internally sorted (according to x and not time order—according to x_1 as far as the rangesearch elements are concerned). Using the above properties about interchanging the time order, we can now prove two important lemmas.

Lemma 1 *A set of less than M elements in a buffer can be made into time order representation, while the relevant $r \cdot B$ matching rangesearch elements and elements are reported, in $O(m + r)$ I/Os.*

Proof: The algorithm simply loads the elements into internal memory and use the special assumption on the delete elements to interchange the time order and report the relevant elements as discussed above. Then it sorts the three groups of elements individually and writes them back to the buffer in time order representation. □

Lemma 2 *Let two sets S_1 and S_2 in time order representation be a subset of a set S such that all elements in S_2 are older than all elements in S_1, and such that each of the other elements in S is either younger or older than all elements in S_1 and S_2. S_1 and S_2 can be "merged" into one set in time order representation, while the relevant $r \cdot B$ matching rangesearch elements and elements are reported, in $O((|S_1| + |S_2|)/B + r)$ I/Os.*

When we in the following write that we report "hits", we actually accumulate elements to be reported in internal memory and report/write them as soon as we have accumulated a block.

1. Interchange the time order of d_1 and i_2 by "merging" them while removing delete/insert matches.
2. Interchange the time order of d_1 and s_2 by "merging" them while reporting "hits" in the following way:

   During the merge a third list of "active" rangesearch elements from s_2 is kept—except for the B most recently added elements—on disk.

   When a rangesearch element from s_2 has the smallest x (that is, x_1) value, it is insert in the list.

   When a delete element from d_1 has the smallest x-value, the list is scanned and it is reported that the element is in the interval of all rangesearch elements that have not yet "expired"—that is, elements whose x_2 value is less than the value of the element from d_2. At the same time all rangesearch elements that have expired are removed from the list.
3. Interchange the time order of s_1 and i_2 by "merging" them and reporting "hits" like in the previous step.
4. Merge i_1 with i_2, s_1 with s_2 and d_1 with d_2.

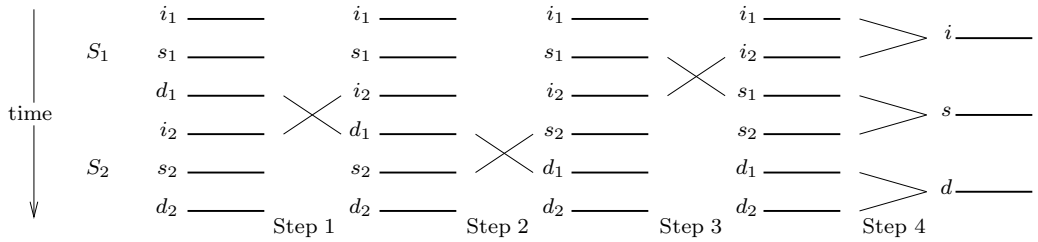


Figure 6: Merging sets in time order representation.

Proof: The algorithm for merging the two sets is given in Figure 6. In step one we push the delete elements d_1 in S_1 down in the time order by "merging" them with the insert elements i_2 in S_2, and in step two we push them further down by "merging" them with the rangesearch elements s_2 in S_2. That we in both cases can do so without missing any rangesearch-element "hits" follows from the time order on the elements and the assumption on the delete elements as discussed above. Then in step three the time order of s_1 and i_2 is interchanged, such that the relevant lists can be merged in step four. That step one and four are done in a linear number of I/Os is obvious, while a simple amortization argument shows that step two and three are also done in a linear number of I/Os, plus the number of I/Os used to report "hits". □

After proving the two lemmas we are now almost ready to present the algorithm for emptying the buffers of all nodes in a subtree. The algorithm will use that all elements in the buffers of nodes on a given level of the structure are always in correct time order compared to all *relevant* elements on higher levels. By relevant we mean that an element in the buffer of node v was inserted in the tree before all elements in buffers of the nodes on the path from v to the root of the tree. This means that we can assume that all elements on one level were inserted before all elements on higher levels.

Lemma 3 *All buffers of a (sub-) range tree with n leaves, where all buffers contain less than M elements, can be emptied and all the elements collected into time order representation in $O(n + r)$ I/Os. Here $r \cdot B$ is the number of matching element and rangesearch elements reported.*

1. Make three lists for each level of the tree, consisting of the elements on the level in question in time order representation:

   For a given level all buffers are made into time order representation using Lemma 1, and then the resulting lists are appended after each other to obtain the total time order representation.
2. Repeatedly and from the top level, merge the time order representation of one level with the time order representation of the next level using Lemma 2.

Figure 7: Emptying all buffers and collecting the elements in time order representation.

Proof: The empty algorithm is given in Figure 7. The correctness of the algorithm follows from Lemma 1 and Lemma 2 and the above discussion. It follows from Lemma 1 that step one creates the time order representation of the elements on each of the levels in a number of I/Os equal to $O(m)$ times the number of nodes in the tree, plus the number of I/Os used to report hits. That the total number of I/Os used is $O(n + r)$ then follows from the fact that the number of nodes in a tree with n leaves is $O(n/m)$. That one merge in step two takes a linear number of I/Os in the number of elements in the lists, plus the I/Os used to report hits, follows form Lemma 2. That the total number of I/Os used is $O(n + r)$ then follows from the fact that every level of the tree contains more nodes than all levels above it put together. Thus the number of I/Os used to merge the time order representation of level j with the time order representation of all the elements above level j is bounded by $O(m)$ times the number of nodes on level j. The bound then again follows from the fact that the total number of nodes in the tree is $O(n/m)$. □

5.2 Buffer-emptying Process on External Range Tree

Having introduced the time order representation and discussed how to empty the buffers of a subtree, we are now ready to describe the buffer-emptying process used on the external range tree. The process is given in Figure 8 and it relies on an important property, namely that when we start emptying a buffer the elements in it can be divided into two categories—what we call "old" and "new" elements. The new elements are those which were inserted in the buffer by the buffer-emptying process that just took place on the parent node, and which triggered the need for a buffer-emptying process on the current node. The old elements are the rest of the elements. We know that the number of old elements is less than M and that they were all inserted before the new elements. As we will maintain the invariant that we distribute elements from one buffer to the buffers one level down in time order representation, this means that we can construct the time order representation of *all* the elements in a buffer as described in the first two steps of the algorithm in Figure 8.

Now if we are working on a leaf node we can report the relevant "hits" between rangesearch element and normal element, just by merging the time order representation of the elements

- Load the less than M old elements in the buffer and make them into time order representation using Lemma 1.
- Merge the the old elements in time order representation with the new elements in time order representation using Lemma 2.
- If we are working on a **leaf** node:
    1. Merge (a copy of) the time order representation with the time order representation consisting of the elements in the children (leaves) using Lemma 2.
    2. Remove the rangesearch elements from the buffer.
- If we are working on an **internal** node:
    1. Scan the delete elements and distribute them to the buffers of the relevant children.
    2. Scan the rangesearch elements and compute which of the subtrees below the current node should have their elements reported.
    3. For every of the relevant subtrees do the following:
        (a) Remove and store the delete elements distributed to the buffer of the root of the subtree in step one above.
        (b) Empty the buffers of the subtree using Lemma 3.
        (c) Merge the resulting time order representation with the time order representation consisting of the delete elements stored in (a) using Lemma 2.
        (d) Scan the insert and delete elements of the resulting time order representation and distribute a copy of the elements to the relevant leaf buffers.
        (e) Merge the time order representation with the time order representation consisting of the elements in the leaves of the subtree using Lemma 2.
        (f) Remove the rangesearch elements.
        (g) Report the resulting elements as being "hit" by the relevant search elements in the buffer.
    4. Scan the rangesearch elements again and distribute them to the buffers of the relevant children.
    5. Scan the insert elements and distribute them to the buffers of the relevant children. Elements for the subtrees which were emptied are distributed to the leaf buffer of these trees.
    6. If the buffer of any of the children now contains more than $m/2$ elements then recursively apply the buffer-emptying process on these nodes.
- When all buffers of the relevant internal nodes are emptied (and the buffers of all relevant leaf nodes have had their rangesearch elements removed) then empty all leaf buffers involved in the above process (and rebalance the tree) using the algorithm given in Figure 5 (Section 3).

Figure 8: Range tree buffer-emptying process.

in the buffer with the time order representation consisting of the elements in the leaves below the node. Then we can remove the rangesearch elements and we are ready to empty the leaf buffer (and rebalance) with the algorithm used on the basic buffer tree.

If we are working on an internal node v things are a bit more complicated. After computing which subtrees we need to empty, we basically do the following for each such tree: We empty the buffers of the subtree using Lemma 3 (step 3, b). We can use Lemma 3 as we know that all buffers of the subtree are non-full, because the buffer-emptying process is done recursively top-down. As discussed the emptying also reports the relevant "hits" between elements and rangesearch elements in the buffers of the subtree. Then we remove the elements which are deleted by delete elements in the buffer of v (step 3, c). Together with the relevant insert elements from the buffer of v, the resulting set of elements should be inserted in or deleted from the tree. This is done by inserting them in the relevant leaf buffers (step 3, d and step

5), which are then emptied at the very end of the algorithm. Finally, we merge the time order representation of the elements from the buffers of the subtree with the elements in the leaves of the structure (step 3, e). Thus we at the same time report the relevant "hits" between elements in the leaves and rangesearch elements from the buffers, and obtain a total list of (undeleted) elements in the subtree. These elements can then be reported as being "hit" by the relevant rangesearch elements from the buffer of v (step 3, g).

After having written/reported the relevant subtrees we can distribute the remaining elements in the buffer of v to buffers one level down—remembering to maintain the invariant that the elements are distributed in time order representation—and then recursively empty the buffers of the relevant children. When the process terminates we empty the buffers of all leaf nodes involved in the process. As these leaf nodes now do not contain any rangesearch elements, this can be done with the algorithm used on the basic buffer tree in Section 3.

Theorem 4 *The total cost of an arbitrary sequence of N intermixed insert, delete and rangesearch operations performed on an initially empty range tree is $O(n \log_m n + r)$ I/O operations. Here $r \cdot B$ is the number of reported elements.*

Proof: The correctness of the algorithm follows from the above discussion. It is relatively easy to realize (using Lemma 1, 2 and 3) that one buffer-emptying process uses a linear number of I/Os in the number of elements in the emptied buffer and the number of elements in the leaves of the emptied subtrees, plus the number of I/Os used to report "hits" between elements and rangesearch elements. The only I/Os we can not account for using the standard argument presented in Section 2 are the ones used on emptying the subtrees. However, as discussed in the beginning of the section, this cost can be divided between the elements reported and the elements deleted, such that the deleted elements pay for their own deletion. The key point is that once the elements in the buffers of the internal nodes of a subtree is removed and inserted in the leaf buffers by the described process, they will only be touched again when they are inserted in or deleted from the tree by the rebalancing algorithm. This is due to the fact mentioned in Section 3 that when a buffer is emptied all buffers on the path to the root are empty, and the fact that we empty all relevant leaf buffers at the end of our buffer-emptying algorithm. □

5.3 Application: Orthogonal Line Segment Intersection Reporting

The problem of *orthogonal line segment intersection reporting* is defined as follows: We are given N line segments parallel to the axes and should report all intersections of orthogonal segments. The optimal plane-sweep algorithm (see e.g. [108]) makes a vertical sweep with a horizontal line, inserting the x coordinate of a vertical segments in a search tree when its top endpoint is reached, and deleting it again when its bottom endpoint is reached. When the sweep-line reaches a horizontal segment, a rangesearch operation with the two endpoints of the segment is performed on the tree in order to report intersections. In internal memory this algorithm will run in the optimal $O(N \log_2 N + R)$ time.

Using precisely the same algorithm and our range tree data structure, and remembering to empty the tree when we are done with the sweep, we immediately obtain the following (using Theorem 4 and Lemma 3):

Corollary 2 *Using our external range tree the orthogonal line segment intersection reporting problem can be solved in $O(n \log_m n + r)$ I/Os.*

As mentioned an algorithm for the problem is also developed in [67], but this algorithm is very I/O specific whereas our algorithm ''hides" the I/O in the range tree. That the algorithm is optimal in the comparison I/O model follows from the $\Omega(N \log_2 N + R)$ comparison model lower bound, and the general connection between comparison and I/O lower bounds proved in [13].

6 An External Segment Tree

In this section we use our technique to develop an external memory version of the segment tree. As mentioned this will enable us to solve the batched range searching and the pairwise rectangle intersection problems in the optimal number of I/Os.

The segment tree [26, 108] is a well-known data structure used to maintain a dynamically changing set of segments whose endpoints belongs to a fixed set, such that given a query point all segments that contain the point can be found efficiently. Such queries are normally called *stabbing queries*. The internal-memory segment tree consists of a static binary tree (the base tree) over the sorted set of endpoints, and a given segment is stored in up to two nodes on each level of the tree. More precisely an interval is associated with each node, consisting of all endpoints below the node, and a segment is stored in all nodes where it contains this interval but not the interval associated with the parent node. The segments stored in a node is just stored in an unordered list. To answer a stabbing query with a point x, one just has to search down the structure for the position of x among the leaves and report all segments stored in nodes encountered in this search.

Because a segment can be stored in $O(\log_2 N)$ nodes the technique sketched in section 2, where we just group the nodes in an internal version of the structure into super-nodes, does not apply directly. The main reason for this is that we would then be forced to use many I/Os to store a segment in these many lists. Instead, we need to change the definition of the segment tree. Our external segment tree is sketched in Figure 9. The base structure is a perfectly balanced tree with branching factor $\sqrt{m}$ over the set of endpoints. We assume without loss of generality that the endpoints of the segments are all distinct and that $\sqrt{m}$ divides n. A buffer and $m/2 - \sqrt{m}/2$ lists of segments are associated with each node. A list (block) of segments is also associated with each leaf. A set of segments is stored in this structure as follows: The first level of the tree (the root) partitions the data into $\sqrt{m}$ slabs σ_i, separated by dotted lines in Figure 9. The *multislabs* for the root are then defined as contiguous ranges of slabs, such as for example $[\sigma_1, \sigma_4]$. There are $m/2 - \sqrt{m}/2$ multislabs and the lists associated with a node are precisely a list for each of the multislabs. Segments

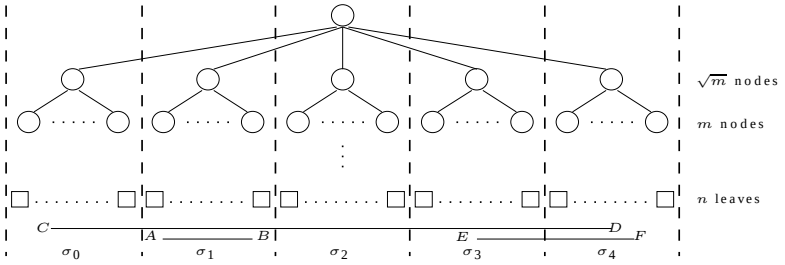


Figure 9: An external segment tree based on a set of N segments, three of which, $\overline{AB}$, $\overline{EF}$ and $\overline{EF}$, are shown.

such as $\overline{CD}$ that completely span one or more slabs are then called *long segments*, and a copy of each long segment is stored in a list associated with the largest multislab it spans. Thus, $\overline{CD}$ is stored in the list associated with the multislab $[\sigma_1, \sigma_3]$. All segments that are not long are called *short segments* and are not stored in any multislab list. Instead, they are passed down to lower levels of the tree where they may span recursively defined slabs and be stored. $\overline{AB}$ and $\overline{EF}$ are examples of short segments. Additionally, the portions of long segments that do not completely span slabs are treated as small segments. There are at most two such synthetically generated short segments for each long segment. Segments passed down to a leaf are just stored in one list. Note that we at most store one block of segments in each leaf. A segment is thus stored in at most two list on each level of the base tree.

Given an external segment tree (with empty buffers) a stabbing query can in analogy with the internal case be answered by searching down the tree for the query value, and at every node encountered report all the long segments associated with each of the multislabs that span the query value. However, answering queries on an individual basis is of course not I/O-efficient. Instead we use the buffer approach as discussed in the next subsection.

6.1 Operations on the External Segment Tree

Usually, when we use a segment tree to solve e.g. the batched range searching problem, we use the operations insert, delete and query. However, a delete operation is not really necessary, as we in the plane-sweep algorithm always know at which "time" a segment should be deleted when it is inserted in the tree. So in our implementation of the external segment tree we will not support the delete operation. Instead, we require that a delete time is given when a segment is inserted in the tree. Note that we already assume (like one normally does in internal memory) that we know the set of x coordinates of the endpoints of segments to be inserted in the tree. In general these assumptions mean that our structure can only be used to solve batched dynamic problems as discussed in the introduction.

It is easy to realize how the base tree structure can be build in $O(n)$ I/O operations given the endpoints in sorted order. First we construct the leaves by scanning through the sorted list, and then we repeatedly construct one more level of the tree by scanning through the previous level of nodes (leaves). In constructing one level we use a number of I/Os proportional to the number of nodes on the previous level, which means that we in total use $O(n)$ I/Os as this is the total number of nodes and leaves in the tree.

When we want to perform an insert or a query operation on the buffered segment tree we do as sketched in Section 2. We make a new element with the segment or query point in question, a time-stamp, and—if the element is an insert element—a delete time. When we have collected a block of such elements, we insert them in the buffer of the root. If the buffer of the root now contains more than $m/2$ elements we perform a buffer-emptying process on it. The buffer-emptying process is presented in Figure 10, and we can now prove the following:

Theorem 5 *Given a sequence of insertions of segments (with delete time) intermixed with stabbing queries, such that the total number of operations is N, we can build an external-memory segment tree on the segments and perform all the operation on it in $O(n \log_m n + r)$ I/O operations.*

Proof: In order to build the base tree we first use $O(n \log_m n)$ I/Os to sort the endpoints of the segments and then $O(n)$ I/Os to build the tree as discussed above. Next, we perform all

On internal nodes:

- Repeatedly load $m/2$ blocks of elements into internal memory and do the following:
  1. Store segments:
     Collect all segments that should be stored in the node (long segments). Then for every multislab list in turn insert the relevant long segment in the list (maintaining the invariant that at most one block of a list is non full). At the same time replace every long segment with the small (synthetic) segments which should be stored recursively.
  2. Report stabbings:
     For every multislab list in turn decide if the segments in the list are stabbed by any query point. If so then scan through the list and report the relevant elements while removing segments which have expired (segments for which all the relevant queries are inserted after their delete time).
  3. Distribute the segments and the queries to the buffers of the nodes on the next level.
- If the buffer of any of the children now contains more than $\frac{1}{2}m$ blocks, the buffer-emptying process is recursively applied on these nodes.

On leaf nodes:

- Do exactly the same as with the internal nodes, except that when distributing *segments* to a child/leaf they are just inserted in a the segment block associated with the leaf.
  As far as the *queries* are concerned, report stabbings with segments from the multislab lists as on internal nodes (and the lists associated with the leaves) and remove the query elements.

Figure 10: The buffer-emptying process.

the operations. In order to prove that this can be done in $O(n \log_m n + r)$ I/Os we should argue that we can do a buffer-emptying process in a linear number of I/Os. The bound then follows as previously.

First consider the buffer-emptying process on an internal node. Loading and distributing the segments to buffers one level down can obviously be done in a linear number of I/Os. The key to realizing that step one also uses $O(m)$ I/Os on each memory load is that the number of multislab lists is $O(m)$. In analogy with the distribution of elements to buffers one level down, this means that the number of I/Os we use on inserting non-full blocks in the multislab lists is bounded by $O(m)$. The number used on full blocks is also $O(m)$, as this is the number of segments and as every segment is at most stored in one list. The number of I/Os charged to the buffer-emptying process in step two is also $O(m)$, as this is the number of I/Os used to load non-full multislab list blocks. The rest of the I/Os used to scan a multislab list can either be charged to a stabbing reporting or to the deletion of an element. We can do so by assuming that every segment holds $O(1/B)$ credits to pay for its own deletion. This credit can then be accounted for (deposited) when we insert a segment in a multislab list. Thus a buffer-emptying process on an internal node can be performed in a linear number of I/Os as required.

As far as leaf nodes are concerned almost precisely the same argument applies, and we have thus proved that we can build the base tree and perform all the operations on the structure in $O(n \log_m n + r)$ I/Os, where r is the number of stabbings reported so far. However, in order to report the remaining stabbings we need to empty all the buffers of the structure. We do so as in Section 3 simply by performing buffer-emptying processes on all nodes level by level starting at the root. As there are $O(n/\sqrt{m})$ nodes in the tree one might think that this process would cost us $O(n\sqrt{m})$ I/Os, plus the number of I/Os used to report stabbings. The problem seems to be that there is $n/\sqrt{m}$ leaf nodes, each having $O(m)$ multislab lists, and that we when we empty the buffers of these nodes can be forced to use an I/O for each of the

multislab lists which are not paid for by reportings. However, the number of multislab lists actually containing *any* segments must be bounded by $O(n \log_m n)$, as that is the number of I/Os performed so far. Thus it is easy to realize that $O(n \log_m n + r)$ must be a bound on the number of I/Os we have to pay in order to empty the buffers of all the leaf nodes. Furthermore, as the number of internal nodes is $O(n/m)$, the buffers of these nodes can all be emptied in $O(n)$ I/Os. This completes the proof of the lemma.

□

6.2 Applications of the External Segment Tree

Having developed an external segment tree we can obtain efficient external-memory algorithms by using it in standard plane-sweep algorithms.

The batched range searching problem—given N points and N rectangles, report for each rectangle all the points that lie inside it—can be solved with a plane-sweep algorithm in almost the same way as the orthogonal line segment intersection problem. The optimal plane sweep algorithm makes a vertical sweep with a horizontal line, inserting a segment (rectangle) in the segment tree when the top segment of a rectangle is reached, and deleting it when the bottom segment is reached. When a point is reached a stabbing query is performed with it. Using our external segment tree in this algorithm yields the following:

Corollary 3 *Using the external range tree the batched range searching problem can be solved in $O(n \log_m n + r)$ I/O operations.*

The problem of pairwise rectangle intersection is defined similar to the orthogonal line segment intersection problem. Given N rectangles in the plane (with sides parallel to the axes) we should report all intersecting pairs. In [26] it is shown that if we—besides the orthogonal line segment intersection problem—can solve the batched range searching problem in time $O(N \log_2 N + R)$, we will in total obtain a solution to the rectangle intersection problem with the same (optimal) time bound. Thus we in external-memory obtain the following:

Corollary 4 *Using our external data structures the pairwise rectangle intersection problem can be solved in $O(n \log_m n + r)$ I/O operations.*

That both algorithms are optimal in the comparison I/O model follows by the internal memory comparison lower bound and the result in [13]. Like in the orthogonal line segment intersection case, optimal algorithms for the two problems are also developed in [67].

7 Extending the Results to the D-disk Model

As mentioned in the introduction an approach to increase the throughput of I/O systems is to use a number of disks in parallel. One method of using D disks in parallel is *disk striping*, in which the heads of the disks are moving synchronously, so that in a single I/O operation each disk read or writes a block in the same location as each of the others. In terms of performance, disk striping has the effect of using a single large disk with block size $B' = DB$. Even though disk striping does not in theory achieve asymptotic optimality when D is very large, it is often the method of choice in practice for using parallel disks [129].

The non optimality of disk striping can be demonstrated via the sorting bound. While sorting N elements using disk striping and one of the the one-disk sorting algorithms requires

$O(n/D \log_{m/D} n)$ I/Os the optimal bound is $O(n/D \log_m n)$ I/Os [5]. Note that the optimal bound results in a linear speedup in the number of disk. Nodine and Vitter [98] managed to develop a theoretical optimal D-disk sorting algorithm based on merge sort, and later they also developed an optimal algorithm based on distribution sort [99]. In the latter algorithm it is assumed that $4DB \leq M - M^{1/2+\beta}$ for some $0 < \beta < 1/2$, an assumption which clearly is non-restrictive in practice. The algorithm works as normal distribution sort by repeatedly distributing a set of N elements into $\sqrt{m}$ sets of roughly equal size, such that all elements in the first set is smaller than all elements in the second set, and so on. The distribution is done in $O(n/D)$ I/Os, and the main issue in the algorithm is to make sure that the elements in one of the smaller sets can be read efficiently in parallel in the next phase of the algorithm, that is, that they are distributed relatively evenly among the disks.

To obtain the results in this paper we basically only used three "paradigms"; *distribution*, *merging* and *sorting*. We used distribution to distribute the elements in the buffer of a node to the buffers of nodes on the next level, and to multislab lists in the segment tree case. We used merging of two lists when emptying all buffers in a (sub-) buffer tree, and we sorted a set of elements when emptying the leaf buffers of the buffer tree.[3] While we of course can use an optimal D-disk sorting algorithm instead of a one-disk algorithm, and while it is easy to merge *two* lists in the optimal number of I/Os on parallel disks, we need to modify our use of distribution to make it work with D disks. As mentioned Nodine and Vitter [99] developed an optimal way of doing distribution, but only when the distribution is done $O(\sqrt{m})$-wise. As already mentioned we can make our buffer tree work with fan-out and buffer size $\Theta(\sqrt{m})$ instead of $\Theta(m)$, and thus we can use the algorithm from [99] to make our structure work in the D-disk model. The external segment tree already has fan-out $\sqrt{m}$, but we still distribute elements (segments) to $\Theta(m)$ multislab list. Thus to make our external segment tree work on D disks we decrease the fan-out to $m^{1/4}$, which does not change the asymptotic I/O bounds of the operations, but decreases the number of multislab lists to $\sqrt{m}$. Thus we can use the algorithm from [99] to do the distribution. To summerize our structures work in the general D-disk model under the non-restrictive assumption that $4DB \leq M - M^{1/2+\beta}$ for some $0 < \beta < 1/2$.

8 Conclusion

In this paper we have developed a technique for transforming an internal-memory tree data structure into an efficient external memory structure. Using this technique we have developed an efficient external priority queue and batched dynamic versions of the (one-dimensional) range tree and the segment tree. We have shown how these structures allow us to design efficient external-memory algorithms from known internal algorithms in a straightforward way, such that all the I/O specific parts of the algorithms are "hidden" in the data structures. This is in great contrast to previously developed algorithms for the considered problems. We have also used our priority queue to develop an extremely simple algorithm for "circuit evaluation", improving on the previously know algorithm.

Recently, several authors have used the structures developed in this paper or modified versions of them to solve important external-memory problems. In [12] the priority queue is used to develop new I/O efficient algorithms for ordered binary-decision diagram manipula-

[3] Note that we could actually do without the sorting by distributing elements in sorted order when emptying a buffer in the buffer tree, precisely as we in the range tree structure distribute them in time order representation.

tion, and in [40] it is used in the development of several efficient external graph algorithm. In [15] an extension of the segment tree is used to develop efficient new external algorithms for a number of important problems involving line segments in the plane, and in [16] the main idea behind the external segment tree (the notion of multislabs) is used to develop an optimal "on-line" versions of the interval tree.

We believe that several of our structures will be efficient in practice due to small constants in the asymptotic bounds. We hope in the future to be able to implement some of the structures in the transparent parallel I/O environment (TPIE) developed by Vengroff [126].

Acknowledgments

I would like to thank all the people in the algorithms group at University of Aarhus for valuable help and inspiration. Special thanks also go to Mikael Knudsen for the discussions that lead to many of the results in this paper, to Sven Skyum for many computational geometry discussions, and to Peter Bro Miltersen, Erik Meineche Schmidt and Darren Erik Vengroff for help on the presentation of the results in this paper. Finally, I would like to thank Jeff Vitter for allowing me to be a part of the inspiring atmosphere at Duke University.

Chapter 7

External-Memory Algorithms for Processing Line Segments in Geographic Information Systems

External-Memory Algorithms for Processing Line Segments in Geographic Information Systems*

Lars Arge†
BRICS¶
Dept. of Computer Science
University of Aarhus
Aarhus, Denmark

Darren Erik Vengroff‡
Dept. of Computer Science
Brown University
Providence, RI 02912
USA

Jeffrey Scott Vitter§
Dept. of Computer Science
Duke University
Durham, NC 27708–0129
USA

January 1996

Abstract

In the design of algorithms for large-scale applications it is essential to consider the problem of minimizing I/O communication. Geographical information systems (GIS) are good examples of such large-scale applications as they frequently handle huge amounts of spatial data. In this paper we develop efficient new external-memory algorithms for a number of important problems involving line segments in the plane, including trapezoid decomposition, batched planar point location, triangulation, red-blue line segment intersection reporting, and general line segment intersection reporting. In GIS systems, the first three problems are useful for rendering and modeling, and the latter two are frequently used for overlaying maps and extracting information from them.

1 Introduction

The Input/Output communication between fast internal memory and slower external storage is the bottleneck in many large-scale applications. The significance of this bottleneck is increasing as internal computation gets faster, and especially as parallel computing gains popularity [107]. Currently, technological advances are increasing CPU speeds at an annual rate of 40–60% while disk transfer rates are only increasing by 7–10% annually [113]. Internal memory sizes are also increasing, but not nearly fast enough to meet the needs of important

*An extended abstract version of this paper was presented at the Third Annual European Symposium on Algorithms (ESA '95).

†Supported in part by the ESPRIT II Basic Research Actions Program of the EC under contract No. 7141 (Project ALCOM II). This work was done while a Visiting Scholar at Duke University. Email: large@daimi.aau.dk.

‡Supported in part by the U.S. Army Research Office under grant DAAH04–93–G–0076 and by the National Science Foundation under grant DMR–9217290. This work was done while a Visiting Scholar at Duke University. Email: dev@cs.brown.edu.

§Supported in part by the National Science Foundation under grant CCR–9007851 and by the U.S. Army Research Office under grant DAAH04–93–G–0076. Email: jsv@cs.duke.edu.

¶Acronym for Basic Research in Computer Science, a Center of the Danish National Research Foundation.

large-scale applications, and thus it is essential to consider the problem of minimizing I/O communication.

Geographical information systems (GIS) are a rich source of important problems that require good use of external-memory techniques. GIS systems are used for scientific applications such as environmental impact, wildlife repopulation, epidemiology analysis, and earthquake studies and for commercial applications such as market analysis, facility location, distribution planning, and mineral exploration [70]. In support of these applications, GIS systems store, manipulate, and search through enormous amounts of spatial data [52, 84, 114, 124]. NASA's EOS project GIS system [52], for example, is expected to manipulate petabytes (thousands of terabytes, or millions of gigabytes) of data!

Typical subproblems that need to be solved in GIS systems include point location, triangulating maps, generating contours from triangulated elevation data, and producing map overlays, all of which require manipulation of line segments. As an illustration, the computation of new scenes or maps from existing information—also called map overlaying—is an important GIS operation. Some existing software packages are completely based on this operation [124]. Given two thematic maps (piecewise linear maps with, e.g., indications of lakes, roads, pollution level), the problem is to compute a new map in which the thematic attributes of each location is a function of the thematic attributes of the corresponding locations in the two input maps. For example, the input maps could be a map of land utilization (farmland, forest, residential, lake), and a map of pollution levels. The map overlay operation could then be used to produce a new map of agricultural land where the degree of pollution is above a certain level. One of the main problems in map overlaying is "line-breaking," which can be abstracted as the red-blue line segment intersection problem.

In this paper, we present efficient external-memory algorithms for large-scale geometric problems involving collections of line segments in the plane, with applications to GIS systems. In particular, we address region decomposition problems such as trapezoid decomposition and triangulation, and line segment intersection problems such as the red-blue segment intersection problem and more general formulations.

1.1 The I/O Model of Computation

The primary feature of disks that we model is their extremely long access time relative to that of solid state random-access memory. In order to amortize this access time over a large amount of data, typical disks read or write large blocks of contiguous data at once. Our problems are modeled by the following parameters:

$$\begin{aligned} N &= \#\text{ of items in the problem instance;} \\ M &= \#\text{ of items that can fit into internal memory;} \\ B &= \#\text{ of items per disk block,} \end{aligned}$$

where $M < N$ and $1 \leq B \leq M/2$. Depending on the size of the data items, typical values for workstations and file servers in production today are on the order of $M = 10^6$ or 10^7 and $B = 10^3$. Large-scale problem instances can be in the range $N = 10^{10}$ to $N = 10^{12}$.

In order to study the performance of external-memory algorithms, we use the standard notion of I/O complexity [5, 133]. We define an *input/output operation* (or simply *I/O* for short) to be the process of reading or writing a block of data to or from the disk. The I/O complexity of an algorithm is simply the number of I/Os it performs. For example,

reading all of the input data requires N/B I/Os. We will use the term *scanning* to describe the fundamental primitive of reading (or writing) all items in a set stored contiguously on external storage by reading (or writing) the blocks of the set in a sequential manner.

For the problems we consider we define two additional parameters:

$$\begin{aligned} K &= \#\text{ of queries in the problem instance;} \\ T &= \#\text{ of items in the problem solution.} \end{aligned}$$

Since each I/O can transmit B items simultaneously, it is convenient to introduce the following notation:

$$n = \frac{N}{B}, \qquad k = \frac{K}{B}, \qquad t = \frac{T}{B}, \qquad m = \frac{M}{B}.$$

We will say that an algorithm uses a linear number of I/O operations if it uses at most $O(n)$ I/Os to solve a problem of size N.

An increasingly popular approach to further increase the throughput of I/O systems is to use a number of disks in parallel. The number D of disks range up to 10^2 in current disk arrays. One method of using D disks in parallel is *disk striping* [133], in which the heads of the disks are moved synchronously, so that in a single I/O operation each disk reads or writes a block in the same location as each of the others. In terms of performance, disk striping has the effect of using a single large disk with block size $B' = DB$. Even though disk striping does not in theory achieve asymptotic optimality [133] when D is very large, it is often the method of choice in practice for using parallel disks, especially when D is moderately sized [129].

1.2 Previous Results in I/O-Efficient Computation

Early work on I/O algorithms concentrated on algorithms for sorting and permutation related problems [5, 49, 98, 99, 133]. External sorting requires $\Theta(n \log_m n)$ I/Os,[1] which is the external-memory equivalent of the well-known $\Theta(N \log N)$ time bound for sorting in internal memory. Work has also been done on matrix algebra and related problems arising in scientific computation [5, 129, 133]. More recently, researchers have designed external-memory algorithms for a number of problems in different areas, such as in computational geometry [67] and graph theoretic computation [42]. In [13] a general connection between the comparison-complexity and the I/O complexity of a given problem is shown, and in [11] alternative solutions for some of the problems in [42] and [67] are derived by developing and using dynamic external-memory data structures.

1.3 Our Results

In this paper, we combine and modify in novel ways several of the previously known techniques for designing efficient algorithms for external memory. In particular we use the *distribution sweeping* and *batch filtering* paradigms of [67] and the *buffer tree* data structure of [11]. In addition we also develop a powerful new technique that can be regarded as a practical external-memory version of batched fractional cascading on an external-memory version of a segment tree. This enables us to improve on existing external-memory algorithms as well as to develop new algorithms and thus partially answer some open problems posed in [67].

In Section 2 we introduce the *endpoint dominance problem*, which is a subproblem of *trapezoid decomposition*. We introduce an $O(n \log_m n)$-I/O algorithm to solve the endpoint

[1] We define for convenience $\log_m n = \max\{1, (\log n)/\log m\}$.

Problem	I/O bound of new result	Result using modified internal memory algorithm
Endpoint dominance.	$O(n \log_m n)$	$O(N \log_B n)$
Trapezoid decomposition.	$O(n \log_m n)$	$O(N \log_B n)$
Batched planar point location.	$O((n+k) \log_m n)$	
Triangulation.	$O(n \log_m n)$	$\Omega(N)$
Segment sorting.	$O(n \log_m n)$	$O(N \log_B n)$
Red-blue line segment intersection.	$O(n \log_m n + t)$	$O(N \log_B n + t)$
Line segment intersection.	$O((n+t) \log_m n)$	$\Omega(N)$

Figure 1: Summary of results.

dominance problem, and we use it to develop an algorithm with the same asymptotic I/O complexity for trapezoid decomposition, *planar point location*, *triangulation of simple polygons* and for the *segment sorting* problem. In Section 3 we give external-memory algorithms for line segment intersection problems. First we show how our segment sorting algorithm can be used to develop an $O(n \log_m n + t)$-I/O algorithm for *red-blue line segment intersection*, and then we discuss an $O((n+t) \log_m n)$-I/O algorithm for the *general segment intersection* problem.

Our results are summarized in Table 1. For all but the batched planar point location problem, no algorithms specifically designed for external memory were previously known. The batched planar point location algorithm that was previously known [67] only works when the planar subdivision is monotone, and the problems of triangulating a simple polygon and reporting intersections between other than orthogonal line segments are stated as open problems in [67].

For the sake of contrast, our results are also compared with modified internal-memory algorithms for the same problems. In most cases, these modified algorithms are plane-sweep algorithms modified to use B-tree-based dynamic data structures rather than binary tree-based dynamic data structures, following the example of a class of algorithms studied experimentally in [41]. Such modifications lead to algorithms using $O(N \log_B n)$ I/Os. For two of the algorithms the known optimal internal-memory algorithms [36, 37] are not plane-sweep algorithms and can therefore not be modified in this manner. It is difficult to analyze precisely how those algorithms perform in an I/O environment; however it is easy to realize that they use at least $\Omega(N)$ I/Os. The I/O bounds for algorithms based on B-trees have a logarithm of base B in the denominator rather than a logarithm of base m. But the most important difference between such algorithms and our results is the fact that the updates to the dynamic data structures are handled on an individual basis, which leads to an extra multiplicative factor of B in the I/O bound, which is very significant in practice.

As mentioned, the red-blue line segment intersection problem is of special interest because it is an abstraction of the important map-overlay problem, which is the core of several vector-based GISs [9, 10, 106]. Although a time-optimal internal-memory algorithm for the general intersection problem exists [37], a number of simpler solutions have been presented for the red-blue problem [35, 38, 87, 106]. Two of these algorithms [38, 106] are not plane-sweep algorithms, but both sort segments of the same color in a preprocessing step with a plane-sweep algorithm. The authors of [106] claim that their algorithm will perform well with inadequate internal memory owing to the fact that data are mostly referenced sequentially.

A closer look at the main algorithm reveals that it can be modified to use $O(n \log_2 n)$ I/Os in the I/O model, which is only a factor of $\log m$ from optimal. Unfortunately, the modified algorithm still needs $O(N \log_B n)$ I/Os to sort the segments.

In this paper we focus our attention on the single disk model. As described in Section 1.1, striping can be used to implement our algorithms on parallel disk systems with $D > 1$. Additionally, techniques from [98] and [100] can be used to extend many of our results to parallel disk systems. In the conference version of this paper we conjectured that all our results could be improved by the optimal factor of D on parallel disk systems with D disks, but it is still an open problem whether the required merges can be done efficiently enough to allow this.

2 The Endpoint Dominance Problem

In this section we consider the *endpoint dominance problem* (EPD) defined as follows: Given N non-intersecting line segments in the plane, find the segment directly above each endpoint of each segment.

EPD is a powerful tool for solving other important problems as we will illustrate in Section 2.1. As mentioned in the introduction a number of techniques for designing efficient I/O-efficient algorithms have been developed in recent years, including distribution sweeping, batch filtering [67] and buffer trees [11]. However, we do not seem to be able to efficiently solve EPD using these techniques directly. Section 2.2 briefly review some of the techniques and during that process we try to illustrate why they are inadequate for solving EPD. Fortunately, as we will demonstrate in Section 2.3, we are able to combine the existing techniques with several new ideas in order to develop an I/O-efficient algorithm for the problem, and thus for a number of other important problems.

2.1 Using EPD to solve other Problems

In this section we with three lemmas illustrate how an I/O-efficient solution to EPD can be used in the construction of I/O-efficient solutions to other problems.

Lemma 1 *If EPD can be solved in $O(n \log_m n)$ I/Os, then the trapezoid decomposition of N non-intersecting segments can be computed in $O(n \log_m n)$ I/Os.*

Proof: We solve two instances of EPD, one to find the segments directly above each segment endpoint and one (with all y coordinates negated) to find the segment directly below each endpoint—see Figure 2 for an example of this on a simple polygon. We then compute the locations of all $O(N)$ vertical trapezoid edges. This is done by scanning the output of the two EPD instances in $O(n)$ I/Os. To explicitly construct the trapezoids, we sort all trapezoid vertical segments by the IDs of the input segments they lie on, breaking ties by x coordinate. This takes $O(n \log_m n)$ I/Os. Finally, we scan this sorted list, in which we find the two vertical edges of each trapezoid in adjacent positions. The total amount of I/O used is thus $O(n \log_m n)$. □

Lemma 2 *If EPD can be solved in $O(n \log_m n)$ I/Os, then a simple polygon with N vertices can be triangulated in $O(n \log_m n)$ I/O operations.*

Proof: After computing the trapezoid decomposition of a simple polygon, the polygon can be triangulated in $O(n)$ I/Os using a slightly modified version of an algorithm from [61]. □

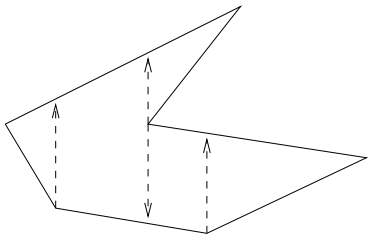


Figure 2: Using EPD to compute the trapezoid decomposition of a simple polygon.

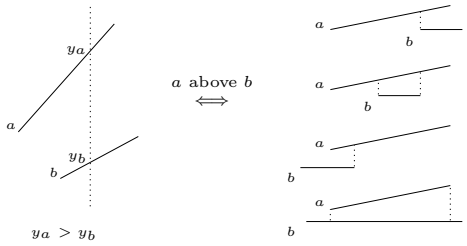


Figure 3: Comparing segments. Two segments can be related in four different ways.

We define a segment $\overline{AB}$ in the plane to be above another segment $\overline{CD}$ if we can intersect both $\overline{AB}$ and $\overline{CD}$ with the same vertical line l, such that the intersection between l and $\overline{AB}$ is above the intersection between l and $\overline{CD}$. Note that two segments are in comparable if they cannot be intersected with the same vertical line. Figure 3 demonstrates that if two segments are comparable then it is enough to consider vertical lines through the four endpoints to obtain their relation. The problem of sorting N non-intersecting segments in the plane is to extending the partial order defined in the above way to a total order. This problem will become important in the solution to the red-blue line segment intersection problem in Section 3.1.

Lemma 3 *If EPD can be solved in $O(n \log_m n)$ I/Os, then a total ordering of N non-intersecting segments can be found in $O(n \log_m n)$ I/Os.*

Proof: We first solve EPD on the input segments augmented with the segment S_∞ with endpoints $(-\infty, \infty)$ and (∞, ∞). The existence of S_∞ ensures that all input segment endpoints are dominated by some segment. We define an aboveness relation $\searrow$ on elements of a non-intersecting set of segments S such that $\overline{AB} \searrow \overline{CD}$ *if and only if* either $(C, \overline{AB})$ or $(D, \overline{AB})$ is in the solution to EPD on S. Here $(A, \overline{BC})$ denotes that $\overline{BC}$ is the segment immediately above A. Similarly, we solve EPD with negated y coordinates and a special segment $S_{-\infty}$ to establish a belowness relation $\nearrow$. As discussed sorting the segments corresponds to extending the partial order defined by $\searrow$ and $\nearrow$ to a total order.

In order to obtain a total order we define a directed graph $G = (V, E)$ whose nodes consist of the input segments and the two extra segments S_∞ and $S_{-\infty}$. The edges correspond to elements of the relations $\searrow$ and $\nearrow$. For each pair of segments $\overline{AB}$ and $\overline{CD}$, there is an edge from $\overline{AB}$ to $\overline{CD}$ iff $\overline{CD} \searrow \overline{AB}$ or $\overline{AB} \nearrow \overline{CD}$. To sort the segments we simply have to topologically sort G. As G is a planar s,t-graph of size $O(N)$ this can be done in $O(n \log_m n)$ I/Os using an algorithm of [42]. □

2.2 Buffer Trees and Distribution Sweeping

In internal memory EPD can be solved optimally with a simple plane-sweep algorithm; We sweep the plane from left to right with a vertical line, inserting a segment in a search tree

when its left endpoint is reached and removing it again when the right endpoint is reached. For every endpoint we encounter we also do a search in the tree to identify the segment immediately above the point.

In [11] a number of external-memory data structures called buffer trees are developed for use in plane-sweep algorithms. Buffer trees are data structures that can support the processing of a batch of N updates and K queries on an initially empty dynamic data structure of elements *from a totally ordered set* in $O((n+k)\log_m n + t)$ I/Os. They can be used to implement plane-sweep algorithms in which the entire sequence of updates and queries is known in advance. The queries that such plane-sweep algorithms ask of their dynamic data structures need not be answered in any particular order; the only requirement on the queries is that they must all eventually be answered. Such problems are known as *batch dynamic problems* [55]. The plane-sweep algorithm for EPD sketched above can be stated as a batched dynamic problem. However, the requirement that the element stored in the buffer tree is taken from a totally ordered set is not fulfilled in the algorithm, as we do not know any total order of the segments. Actually, as demonstrated in Lemma 3, finding such an ordering is an important application of EPD. Therefore, we cannot use the buffer tree as the tree structure in the plane-sweep algorithm to get an I/O-efficient algorithm. For the other problems we are considering in this paper, the known internal-memory plane-sweep solutions cannot be stated as batched dynamic algorithms (since the updates depend on the queries) or else the elements involved are not totally ordered.

In [67] a powerful external memory version of the plane-sweep paradigm called distribution sweeping is introduced. Unfortunately, direct application of distribution sweeping appears insufficient to solve EPD. In order to illustrate why distribution sweeping is inadequate for the task at hand, let us briefly review how it works. We divide the plane into m vertical *slabs*, each of which contains $\Theta(n/m)$ input objects, for example points or line segment endpoints. We then sweep down vertically over all of the slabs to locate components of the solution that involve interaction of objects in different slabs or objects (such as line segments) that completely span one or more slabs. The choice of m slabs is to ensure that one block of data from each slab fits in main memory. To find components of the solution involving interaction between objects residing in the same slab, we recursively solve the problem in each slab. The recursion stops after $O(\log_m n/m) = O(\log_m n)$ levels when the subproblems are small enough to fit in internal memory. In order to get an $O(n\log_m n)$ algorithm one therefore need to be able to do one sweep in $O(n)$ I/Os. Normally this is accomplished by preprocessing the objects by using an optimal algorithm to sort them by y-coordinate. This e.g. allows one to avoid having to perform a sort before each recursive application of the technique, because as the objects are distributed to recursive subproblems their y ordering is retained. The reason that distribution sweeping fails for EPD is that there is no necessary relationship between the y ordering of endpoints of segments and their endpoint dominance relationship. In order to use distribution sweeping to get an optimal algorithm for EPD we instead need to sort the segments in a preprocessing step which leaves us with the same problem we encountered in trying to use buffer trees for EPD.

As know techniques fails to solve EPD optimally we are led instead to other approaches as discussed in the next section.

2.3 External-Memory Segment Trees

The *segment tree* [23, 108] is a well-known dynamic data structure used to store a set of segments in one dimension, such that given a query point all segments containing the point can be found efficiently. Such queries are called *stabbing queries*. An external-memory segment tree based on the approach in [11] is shown in Figure 4. The tree is perfectly balanced over the endpoints of the segments it represents and has branching factor $\sqrt{m/4}$. Each leaf represents $M/2$ consecutive segment endpoints. The first level of the tree partitions the data into $\sqrt{m/4}$ intervals σ_i—for illustrative reasons we call them slabs—separated by dotted lines on Figure 4. Multislabs are defined as contiguous ranges of slabs, such as for example $[\sigma_1, \sigma_4]$. There are $m/8 - \sqrt{m}/4$ multislabs. The key point is that the number of multislabs is a quadratic function of the branching factor. The reason why we choose the branching factor to be $\Theta(\sqrt{m})$ rather than $\Theta(m)$ is so that we have room in internal memory for a constant number of blocks for each of the $\Theta(m)$ multislabs. The smaller branching factor at most about doubles the height of the tree.

Segments such as $\overline{CD}$ that completely span one or more slabs are called *long segments*. A copy of each long segment is stored in the largest multislab it spans. Thus, $\overline{CD}$ is stored in $[\sigma_1, \sigma_3]$. All segments that are not long are called *short segments* and are not stored in any multislab. Instead, they are passed down to lower levels of the tree where they may span recursively defined slabs and be stored. $\overline{AB}$ and $\overline{EF}$ are examples of short segments. The portions of long segments that do not completely span slabs are treated as small segments. There are at most two such synthetically generated short segments for each long segment and total space utilization is thus $O(n \log_m n)$ blocks.

To answer a stabbing query, we simply proceed down a path in the tree searching for the query value. At each node we encounter, we report all the long segments associated with each of the multislabs that span the query value.

Because of the size of the nodes and auxiliary multislab data, the buffer tree approach is inefficient for answering single queries. In batch dynamic environments, however, it can be used to develop optimal algorithms. In [11], techniques are developed for using external-memory segment trees in a batch dynamic environment such that inserting N segments in the tree and performing K queries requires $O((n+k)\log_m n + t)$ I/Os.

It is possible to come close to solving EPD by first constructing an external-memory

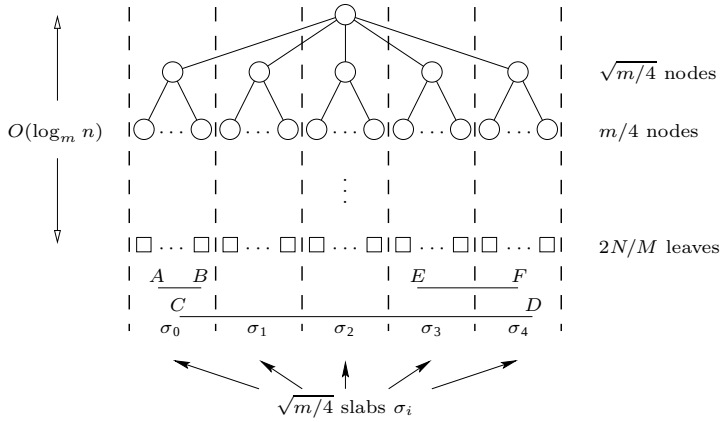


Figure 4: An external-memory segment tree based on a buffer tree over a set of N segments, three of which, $\overline{AB}$, $\overline{CD}$, and $\overline{EF}$, are shown.

segment tree over the projections of the segments onto the x-axis and then performing stabbing queries at the x coordinates of the endpoints of the segments. However, what we want is the single segment directly above each query point in the y dimension, as opposed to all segments it stabs. Fortunately, we are able to modify the external segment tree in order to efficiently answer a batch of this type of queries. The modification requires two significant improvements over existing techniques. First, as discussed in Section 2.3.1, we need to strengthen the definition of the structure, and the tree construction techniques of [11] must be modified in order to guarantee optimal performance when the structure is built. Second, as discussed in Section 2.3.2 the batched query algorithm must be augmented using techniques similar to fractional cascading [39].

2.3.1 Constructing Extended External Segment Trees

We will construct what we call an *extended external segment tree* using an approach based on distribution sweeping. When we are building an external segment tree on *non-intersecting* segments in the plane we can compare all segments in the same multislab just by comparing the order of their endpoints on one of the boundaries. An extended external segment tree is just an external segment tree as described in the last section built on non-intersecting segments, where the segments in each of the multislabs are sorted. Before discussing how to construct an extended external segment tree I/O-efficiently we will show a crucial property, namely that the segments stored in the multislab lists of a node in such a structure can be sorted efficiently. We will use this extensively in the rest of the paper. When we talk about sorting segments in the multislab lists of a node we imagine that they are "cut" to the slab boundaries, that is, that we have removed the part of the segments that are stored recursively further down the structure. Note that this might result in another total order on the segments than if we considered the whole segment.

Lemma 4 *The set of N segment stored in the multislab lists of an internal node of an extended external segment tree can be sorted in $O(n)$ I/O operations.*

Proof: We claim that we can construct a sorted list of the segments by repeatedly looking at the top segment in each of the multislabs, and selecting one of them to go to the sorted list.

To prove the claim, assume for the sake of contradiction that there exists a top segment s in one of the multislab lists which is above the top segment in all the other multislab lists it is comparable with, but which must be below a segment t in a total order. If this is the case there exist a series of segment $s_1, s_2 \ldots, s_i$ such that t is above s_1 which is above s_2 and so on ending with s_i being above s. But if s_i is above s then so is the top segment in the multislab list containing s_i contradicting the fact that s is above the top segment in all multislab lists it is comparable with.

As the number of multislab lists is $O(m)$ there is room for a block from each of them in internal memory. Thus the sorted list can be constructed in $O(n)$ I/Os by performing a standard external-memory merge of $O(m)$ sorted lists into a single sorted list. □

In order to construct an extended external segment tree on N segments, we first use an optimal sorting algorithm to create a list of all the endpoints of the segments sorted by x-coordinate. This list is used during the whole algorithm to find the medians we use to split the interval associated with a given node into $\sqrt{m/4}$ vertical slabs. We now construct the $O(m)$ sorted multislab lists associated with the root in the following way: First we scan through

the segments and distribute the long segments to the appropriate multislab list. This can be done in $O(n)$ I/Os because we have enough internal memory to hold a block of segments for each multislab list. Then we sort each of these lists individually with an optimal sorting algorithm. Finally, we recursively construct an extended external segment tree for each of the slabs. The process continues until the number of endpoints in the subproblems falls below $M/2$.

Unfortunately, this simple algorithm requires $O(n \log_m^2 n)$ I/Os, because $O(n \log_m n)$ I/Os are used to sort the multislab lists on each level of the recursion. To avoid this problem, we modify our algorithm to construct the multislab lists of a node not only from a list of segments but also from two other *sorted* lists of segments. One sorted list consists of segments that have one endpoint in the x range covered by the node under construction and one to the left thereof. The other sorted list is similar but contains segments entering the range from the right. Both lists are sorted by the y coordinate at which the segments enter the range of the node being constructed. In the construction of the structure the two sorted lists will contain segments which was already stored further up the tree. We begin to build a node just as we did before, by scanning through the unsorted list of segments, distributing the long segments to the appropriate multislab lists, and then sorting each multislab list. Next, we scan through the two sorted lists and distribute the long segments to the appropriate multislab lists. Segments will be taken from these lists in sorted order, and can thus be merged into the previously sorted multislab lists at no additional asymptotic cost. This completes the construction of the sorted multislab lists, and now we simply have to produce the input for the algorithm at each of the $\sqrt{m/4}$ children of the current node. The $\sqrt{m/4}$ unsorted lists are created by scanning through the list of segments as before, distributing the segments with both endpoints in the same slab to the list associated with the slab in question. The $2\sqrt{m/4}$ sorted lists of boundary crossing segments are constructed from the sorted multislab lists generated at the current level; First we use a linear number of I/Os sort the segments (Lemma 4) and then the $2\sqrt{m/4}$ lists can be constructed by scanning through the sorted list of segments, distributing the boundary crossing segments to the appropriate of $2\sqrt{m/4}$ lists. These lists will automatically be sorted.

In the above process all the distribution steps can be done in a linear number of I/Os, because the number of lists we distribute into always is $O(m)$, which means that we have enough internal memory to hold a block of segments for each output list. Thus, each level of recursion uses $O(n)$ I/Os plus the number of I/Os used on sorting. The following lemma then follows from the fact that each segment only ones is contained in a list that is sorted:

Lemma 5 *An extended external segment tree on N non-intersecting segments in the plane can be constructed in $O(n \log_m n)$ I/O operations.*

2.3.2 Filtering Queries Through an Extended External Segment Tree

Having constructed an extended external segment tree, we can now use it to find the segments directly above each of a series of K query points. In solving EPD, we have $K = 2N$, and the query points are the endpoints of the original segments. To find the segment directly above a query point p, we examine each node on the path from the root of the tree to the leaf containing p's x coordinate. At each such node, we find the segment directly above p by examining the sorted segment list associated with each multislab containing p. This segment can then be compared to the segment that is closest to the query point p so far, based on

segments seen further up the tree, to see if it is the new globally closest segment. All K queries can be processed through the tree at once using a technique similar to batch filtering [67], in which *all* queries are pushed through a given level of the tree before moving on to the next level.

Unfortunately, the simple approach outlined in the preceding paragraph is not efficient. There are two problems that have to be dealt with. First, we must be able to look for a query point in many of the multislabs lists corresponding to a given node simultaneously. Second, searching for the position of a point in the sorted list associated with a particular multislab may require many I/Os, but as we are looking for an $O(n \log_m n)$ solution we are only allowed to use a linear number of I/Os to find the positions off *all* the query points. To solve the first problem, we will take advantage of the internal memory that is available to us. The second problem is solved with a notion similar to fractional cascading [38, 39]. The idea behind fractional cascading on internal-memory segment trees [123] is that instead of searching for the same element in a number of sorted lists of different nodes, we augment the list at a node with sample elements from lists at the node's children. We then build bridges between the augmented list and corresponding elements in the augments lists of the node's children. These bridges obviate the need for full searches in the lists at the children. We take a similar approach for our external-memory problem, except that we send sample elements from parents to children. Furthermore, we do not use explicit bridges. Our approach uses ideas similar to ones used in [18, 20].

As a first step towards a solution based on fractional cascading, we preprocess the extended external segment tree in the following way (corresponding to "building bridges"): For each internal node, starting with the root, we produce a set of sample segments. For each of the $\sqrt{m/4}$ slabs (*not* multislabs) we produce a list of samples of the segments in the multislab lists that span it. The sample list for a slab consists of every $(2\sqrt{m/4})$th segment in the sorted list of segments that spans it, and we "cut" the segments to the slab boundaries. All the samples are produced by scanning through the sorted list of all segments in the node produced as in Lemma 4, distributing the relevant segments to the relevant sample lists. This can be done efficiently simply by maintaining $\sqrt{m/4}$ counters during the scan, counting how many segments so far have been seen spanning a given slab. For every slab we then augment the multislab lists of the corresponding child by merging the sampled list with the multislab list of the child that contains segments spanning the whole x-interval. This merging happens before we proceed to preprocessing the next level of the tree. At the lowest level of internal nodes, the sampled segments are passed down to the leaves.

We now prove a crucial lemma about the I/O complexity of the preprocessing steps and the space of the resulting data structure:

Lemma 6 *The preprocessing described above uses $O(n \log_m n)$ I/Os. After the preprocessing there are still $O(N)$ segments stored in the multi-lists on each level of the structure. Furthermore, each leaf contains less than M segments.*

Proof: Before any samples are passed down the tree, we have at most $2N$ segments represented at each level of the tree. Let N_i be the number of long segments, both original segments and segments sent down from the previous level, among all the nodes at level i of the tree after the preprocessing step. At the root, we have $N_0 \leq 2N$. We send at most $N_i/(2\sqrt{m/4}) \cdot \sqrt{m/4} = N_i/2$ segments down from level i to level $i+1$. Thus, $N_{i+1} \leq 2N + N_i/2$. By induction on i, we can show that for all i, $N_i = \left(4 - (1/2)^{i-1}\right) N = O(N)$. From Lemma 4 and the fact that

the number of multislab lists is $O(m)$—and thus that we can do a distribution or a merge step in a single pass of the data—it follows that each segment on a given level is read and written a constant number of times during the preprocessing phase. The number of I/Os used at level i of the tree is thus $O(n_i)$, where $n_i = N_i/B$. Since there are $O(\log_m n)$ levels, we in total use $O(n \log_m n)$ I/Os.

Before preprocessing, the number of segments stored in a node is less than the number of endpoints in the leaves below the node. To be precise, a leaf contains less than $M/2$ segments and a node i levels up the tree from a leaf contains less than $M/2 \cdot (\sqrt{m/4})^i$ segments. After preprocessing, the number of segments N_l in a leaf at level l in the tree must be $N_l \leq M/2 + \frac{N_{l-1}}{2\sqrt{m/4}}$, where N_{l-1} is the maximal number of segments in a node at level $l-1$; this is because at most every $(2\sqrt{m/4})$th of these segments are sent down to the leaf. Thus,

$$N_l \leq M/2 + \frac{M/2 \cdot \sqrt{m/4} + N_{l-2}/2\sqrt{m/4}}{2\sqrt{m/4}} \leq M/2 + M/4 + \frac{N_{l-2}}{(2\sqrt{m/4})^2}$$

and so on, which means that $N_l < M$. □

Having preprocessed the tree, we are now ready to filter the K query points through it. We assume without loss of generality that $K = O(N)$. If $K = \Omega(N)$ we break the queries into K/N groups of $K' = N$ queries and process each group individually. For EPD, we have $K = 2N$, so this grouping is not necessary. But as we will see later, grouping reduces the overall complexity of processing a batch of queries when K is very large. Since our fractional cascading construction is done backwards (sampled segments sent downwards), we filter queries from the leaves to the root rather than from the root to the leaves. To start off, we sort the K query points by their x coordinates in $O(k \log_m k)$ I/Os. We then scan the sorted list of query points to determine which leaf a given query belongs to. This produces an unsorted list of queries for each leaf as indicated on Figure 5a). Next we iterate through the leaves, and for each leaf find all dominating segments of the queries assigned to the leaf that are among the segments in the leaf. This is done by loading the entire set of segments stored at that leaf (which fits in memory according to Lemma 6), and then use an internal-memory algorithm to find the dominating segment for each query. As the total size of the data in all the leaves is $O(N)$, the total I/O complexity of the process is $O(k + n)$. In order to prepare for the general step of moving queries up the tree, we sort the queries that went into each leaf based on the order of the segments that we found to be directly above them, ending up in a situation as indicated in Figure 5b). This takes $O(k \log_m k)$ I/Os.

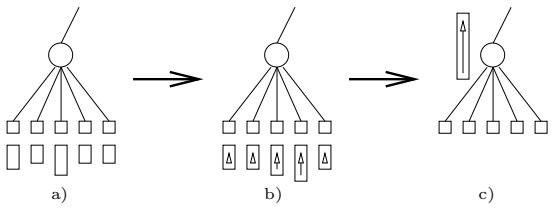


Figure 5: Filtering queries through the structure. An arrow in a list indicate that it is sorted.

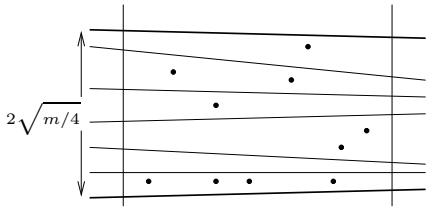


Figure 6: All queries between sampled segments (indicated by fat lines) must appear together in the list of queries for the slab.

Each filtering step of the algorithm begins with a set of queries at a given level, partitioned by the nodes at that level and ordered within the nodes by the order of the segments found to be directly above them on the level. This is exactly what the output of the leaf processing was. The filtering step should produce a similar configuration on the next level up the tree. For one node this is indicated on Figure 5c). Remember that throughout the algorithm we also keep track of the segment found to be closest to a given query point so far, such that when the root is reached we have found the dominating segment off all query points.

To perform one filtering step on a node we merge the list of queries associated with its children (slabs) and the node's multislab lists. The key property that allows us to find the dominating segments among the segments stored in the node in an I/O-efficient manner, and sort the queries accordingly, is that the list of queries associated with a child of the node cannot be to unsorted relative to their dominating segment in the node. This is indicated in Figure 6.

In order to produce for each slab a list of the queries in the slab, sorted according to dominating segment in the node, we again produce and scan through a sorted list of segments in the multislab list of the node, just like when we generated the samples that were passed down the tree in the the preprocessing phase. This time, however, instead of generating samples to pass down the tree, we insert a given segment in a list for each slab it spans. Thus if a segment completely spans four slabs it is inserted in four lists. If, during the scan, we encounter a segment which was sampled in slab s in the sampling phase then we stop the scan and process the queries in the list of queries for s between the sampled segment just encountered and the last sampled segment. As previously discussed these queries appear together in the sorted (according to dominating segment on the last level) list of queries for s. When this is done we clear the list of segments spanning s and continue the scan. The scan continues until all multislab segments have been processed. The crucial property is now that during the scan we can hold all the relevant segments in main memory because at no time during the scan do we store more than $2\sqrt{m/4}$ segments for each slab, that is, $2\sqrt{m/4} \cdot \sqrt{m/4} = m/2$ segments in total. Thus we can perform the scan, not counting the I/Os used to process the queries, in a linear number of I/Os.

To process the queries in a slab between two sampled segments we maintain $2\sqrt{m/4}$ output blocks, each of which corresponds to a segment between the two sampled segments. The block for a segment is for queries with the segment as dominating segment among the segments in the multislab list. As we read queries from the output of the child, we place them in the appropriate output block for the slab. If these output blocks become full, we write them back to disk. Once all queries between the two sampled segments have been processed, we concatenate the outputs associated with each of the segments between the samples. This results in a list of queries sorted according to dominating segment in the node, and this list is appended to an output list for the slab. All of the above is done in a number of I/Os linear in the number of queries processed.

When we finish the above process, we merge the sorted output query lists of all the slabs to produce the output of the current node in a linear number of I/Os.

As discussed above, once this process has reached the root, we have the correct answers to all queries. The total I/O complexity of the algorithm is given by the following theorem.

Theorem 1 *An extended external segment tree on N non-intersecting segments in the plane can be constructed, and K query points can be filtered through the structure in order to find the dominating segments for all these points, in $O((n+k)\log_m n)$ I/O operations.*

Proof: According to Lemma 5 and Lemma 6 construction and preprocessing together require $O(n \log_m n)$ I/Os.

Assuming $K \leq N$, sorting the K queries takes $O(n \log_m n)$ I/Os. Filtering the queries up one level in the tree takes $O(n)$ I/Os for the outer scan and $O(k)$ I/Os to process the queries. This occurs through $O(\log_m n)$ levels, giving an overall I/O complexity of $O(n \log_m n)$.

When $K > N$, we can break the problem into $K/N = k/n$ sets of N queries. Each set of queries can be answered as shown above in $O(n \log_m n)$ I/Os, giving a total I/O complexity of $O(k \log_m n)$. □

Theorem 1 immediately gives us the following bound for EPD, for which $K = 2N$.

Corollary 1 *The endpoint dominance problem can be solved in $O(n \log_m n)$ I/O operations.*

We then immediately get the following from Lemma 1, 2 and 3.

Corollary 2 *The trapezoid decomposition and the total order of N non-intersecting segments in the plane, as well as the triangulation of a simple polygon, can all be computed in $O(n \log_m n)$ I/O operations.*

It remains open whether a simple polygon can be triangulated in $O(n)$ I/Os when the input vertices are given by their order on the boundary of the polygon, which would match the linear internal-memory bound [36].

As a final direct application of our algorithm for EPD we consider the *multi-point planar point location problem*. This is the problem of reporting the location of K query points in a planar subdivision defined by N line segments. In [67] an $O((n+k) \log_m n)$-I/O algorithm for this problem is given for *monotone* subdivisions of the plane. Using Theorem 1 we can immediately extended the result to arbitrary planar subdivisions.

Lemma 7 *The multi-point planar point location problem can be solved in $O((n+k) \log_m n)$ I/O operations.*

3 Line Segment Intersection

In this section we design algorithms for line segment intersection reporting problems. In Section 3.1 we develop an I/O-efficient algorithm for the red-blue line segment intersection problem and in Section 3.2 we develop an algorithm for the general line segment intersection problem.

3.1 Red-Blue Line Segment Intersection

Using our ability to sort segments as described in Section 2, we can now overcome the problems in solving the red-blue line segment intersection problem with distribution sweeping. Given input sets S_r of non-intersecting red segments and S_b of non-intersecting blue segments, we construct two intermediate sets

$$T_r = S_r \cup \bigcup_{(p,q) \in S_b} \{(p,p),(q,q)\}$$

$$T_b = S_b \cup \bigcup_{(p,q) \in S_r} \{(p,p),(q,q)\}$$

Each new set is the union of the input segments of one color and the endpoints of the segments of the other color (or rather zero length segments located at the endpoints). Both T_r and T_b are of size $O(|S_r| + |S_b|) = O(N)$. We sort both T_r and T_b using the algorithm from the previous section, and from now on assume they are sorted. This preprocessing sort takes $O(n \log_m n)$ I/Os.

We now locate intersections between the red and blue segments with a variant of distribution sweeping with a branching factor of $\sqrt{m}$. As discussed in Section 2.2, the structure of distribution sweeping is that we divide the plane into $\sqrt{m}$ slabs, not unlike the way the plane was divided into slabs to build an external segments tree in Section 2.3. We define *long segments* as those crossing one or more slabs and *short segments* as those completely contained in a slab. Furthermore, we shorten the long segments by "cutting" them at the right boundary of the slab that contain their left endpoint, and at the left boundary of the slab containing their right endpoint. This may produce up to two new short segments for each long segment, and below we show how to update T_r and T_b accordingly in $O(n)$ I/Os. We also show how to report all T_i intersections between the long segments of one color and the long and short segments of the other color in $O(n+t_i)$ I/Os. Next, we use one scan to partition the sets T_r and T_b into $\sqrt{m}$ parts, one for each slab, and we recursively solve the problem on the short segments contained in each slab to locate their intersections. Each original segment is represented at most twice at each level of recursion, thus the total problem size at each level of recursion remains $O(N)$ segments. Recursion continues through $O(\log_m n)$ levels until the subproblems are of size $O(M)$ and thus can be solved in internal memory. This gives us the following result:

Theorem 2 *The red-blue line segment intersection problem on N segments can be solved in $O(n \log_m n + t)$ I/O operations.*

Now, we simply have to fill in the details of how we process the segments on one level of the recursion. First, we consider how to insert the new points and segments generated when we cut a long segment at the slab boundaries into the sorted orders T_r and T_b. Consider a cut of a long red segment s into three parts. Changing T_r accordingly is easy, as we just need to insert the two new segments just before or after s in the total order. In order to insert all new red endpoints generated by cutting long red segments (which all lie on a slab boundary) in T_b, we first scan through T_r generating the points and distributing them to $\sqrt{m}$ lists, one for each boundary. The lists will automatically be sorted and therefore it is easy to merge them into T_r in a simple merge step. Altogether we update T_r and T_b in a $O(n)$ I/Os.

Next, we consider how intersections involving long segments are found. We divide the algorithm into two parts; reporting intersections between long and short segments of different colors and between long segments of different colors.

Because T_r and T_b are sorted, we can locate interactions between long and short segments using the distribution-sweeping algorithm used to solve the orthogonal segment intersection problem in [67]. We use the algorithm twice and treat long segments of one color as horizontal segments and short segments of the other color as vertical segments. We sketch the algorithm for long red and blue short segments (details can be found in [67]); We sweep from top to bottom by scanning through the sorted list of red segments and blue endpoints T_r. When a top endpoint of a small blue segment is encountered, we insert the segment in an *active list* (a stack where we keep the last block in internal memory) associated with the slab containing the segment. When a long red segment is encountered we then scan through all the active

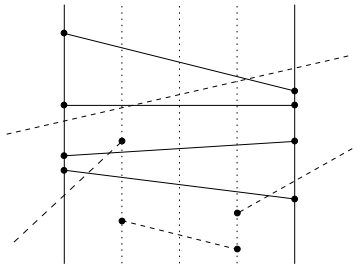


Figure 7: Long blue segments (dashed lines) can interact with multislab in three ways.

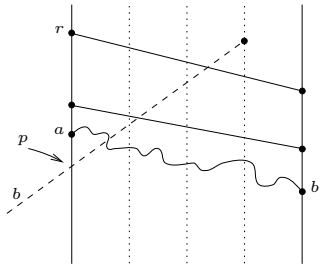


Figure 8: Proof of Lemma 8. The segment between a and b must intersect b.

lists associated with the slabs it completely spans. During this scan we know that every small blue segment in the list either is intersected by the red segment or will not be intersected by any of the following red segments (because we process the segments in sorted order), and can therefore be removed from the list. A simple amortization argument then shows that we use $O(n + t_i)$ I/Os to do this part of the algorithm.

Next we turn to the problem of reporting intersections between long segments of different colors. We define a multislab as in Section 2.3.1 to be a slab defined by two of the $\sqrt{m}$ boundaries. In order to report the intersections we scan through T_r and distribute the long red segments into the $O(m)$ multislabs. Next, we scan through the blue set T_b, and for each blue segment we report the intersections with the relevant long red segments. This is the same as reporting intersections with the appropriate red segments in each of the multislab lists. Now consider Figure 7. A long blue segments can "interact" with a multislab in three different ways. It can have one endpoint in the multislab, it can cross the multislab completely, or it can be totally contained in the multislab. First, let us concentrate on reporting intersections with red segments in multislabs for which the blue segment intersects the left boundary. Consider a blue segment b and a multislab m containing its right endpoint, and define y_p to be the y coordinate of a point p. We have the following:

Lemma 8 *If a blue segment b intersects the left boundary of a multislab at point p then all blue segments processed after b will intersect the same boundary at a point q below p. Let r be the left endpoint of a red segment in the multislab list. If $y_r \geq y_p$ and b intersects the red segment, then b intersects all red segments in the multislab list with left endpoints in the y-range $[y_p, y_r]$. The case $y_r \leq y_p$ is symmetric.*

Proof: The first part follows immediately from the fact that we process the segments in sorted order. Figure 8 demonstrates that the second part holds. □

Using this lemma we can now complete the design of the algorithm for our problem using a merging scheme. As discussed above, we process the blue segment in T_b one at a time and report intersections with red segments in multislabs list where the blue segment intersect the left boundary. For each such multislab list we do the following: We scan *forward* from the current position in the list until we find the first red segment s_r whose left endpoint lies below the intersection between the blue segment and the multislab boundary. Then we scan backward or forward as necessary in the multislab list in order to report intersections. Lemma 8 shows that the algorithm reports all intersections because all intersected segments lies consecutively above or belove s_r. Furthermore, it shows that we can use blocks efficiently such

that we in total only scan through each multislabs list once without reporting intersections. Thus, our algorithm uses a total of $O(n + t_i)$ I/Os.

This takes care of the cases where the blue segment completely spans a multislab or where it has its right, and only the right, endpoint in the multislab. The case where the blue segment only has its left endpoint in the multislab can be handled analogously. The remaining case can be handled with the same algorithm, just by distributing the blue segments instead of the red segments, and then processing one long red segment at a time. To summarize, we have shown how to perform one step of the distribution sweeping algorithm in $O(n+t_i)$ I/Os, and thus proven Theorem 2.

3.2 General Line Segment Intersection

The general line segment intersection problem cannot be solved by distribution sweeping as in the red-blue case, because the $\nearrow$ and $\searrow$ (Lemma 3) relations for sets of intersecting segments are not acyclic, and thus the preprocessing phase to sort the segments cannot be used to establish an ordering for distribution sweeping. However, as we show below, extended external segment trees can be used to establish enough order on the segments to make distribution sweeping possible. The general idea in our algorithm is to build an extended external segment tree on all the segments, and during this process to eliminate any inconsistencies that arise because of intersecting segments *on the fly*. This leads to a solution for the general problem that integrates all the elements of the red-blue algorithm into one algorithm. In this algorithm, intersections are reported both during the construction of an extended external segment tree and during the filtering of endpoints through the structure.

In order to develop the algorithm we need an external-memory priority queue [11]. Given m_p blocks of internal memory, N insert and delete-min operations can be performed on such a structure in $O(n \log_{m_p} n)$ I/Os. If we chose m_p to be m^c for some constant c $(0 < c < 1)$, we can perform the N operations using $O(n \log_m n)$ I/Os. In the construction of an extended external segment tree for general line segment intersection, we use two priority queues for each multislab. In order to have enough memory to do this, we reduce the fan-out of the extended segment tree from $\sqrt{m/4}$ to $(m/4)^{1/4}$. This does not change the asymptotic height of the tree, but it means that each node will have less than $\sqrt{m}/4$ multislabs. We chose m_p to be $\sqrt{m}$. Thus, with two priority queues per multislab, each node of the external segment tree still requires less than $m/2$ blocks of internal memory. Exactly what goes into the priority queues and how they are used will become clear as we describe the algorithm.

3.2.1 Constructing the Extended External Segment Tree

In the construction of an extended external segment tree in Section 2.3.1 we used the fact that the segments did not intersect in order to establish an ordering on them. The main idea in our algorithm is a mechanism for breaking *long segments* into smaller pieces every time we discover an intersection during construction of the multislab lists of a node. In doing so we manage to construct an extended segment tree with no intersections between long segments stored in the multislab lists of the same node.

In order to construct the extended external segment tree on the N (now possibly intersecting) segments, we as in Section 2.3.1 first in $O(n \log_m n)$ I/Os create a sorted list of all the endpoints of the segments. The list is sorted by x coordinate, and used during the whole algorithm to find the medians we use to split the interval associated with a node into $(m/4)^{1/4}$

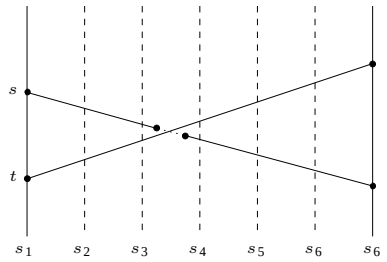


Figure 9: Breaking a segment.

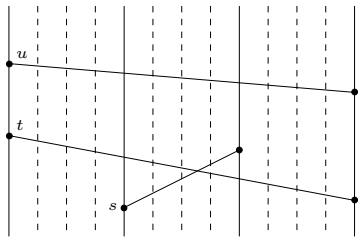


Figure 10: Proof of lemma 9.

vertical slabs. Recall that in Section 2.3.1 one node in the tree was constructed from three lists, one sorted list of segments for each of the two extreme boundaries and one unsorted list of segments. In order to create a node we start as in the non-intersecting case by scanning through the unsorted list of segments, distributing the long segments to the appropriate multislab lists. Next, we sort the multislab lists individually according to the left (or right) segment endpoint. Finally, we scan through the two sorted lists and distribute the segments from these lists. The corresponding multislab lists will automatically be sorted according to the endpoint on one of the boundaries.

Now we want to remove inconsistencies by removing intersections between long segments stored in the multislab lists. We start by removing intersections between segments stored in the same list. To do so we initialize two external priority queues for each of the multislabs, one for each boundary. Segments in these queues are sorted according to the order of the their endpoint on the boundary in question, and the queues are structured such that a delete-min operation returns the topmost segment. We process each of the multislab lists individually as follows: We scan through the list and check if any two consecutive segments intersect. Every time we detect an intersection we report it, remove one of the segment from the list, and break it at the intersection point as indicated on Figure 9. This creates two new segments. If either one of them are long we insert it in both the priority queues corresponding to the appropriate multislab list. Any small segments that are created are inserted into a special list of segments which is distributed to the children of the current node along with normal small segments. The left part of s on Figure 9 between s_1 and s_3 is for example inserted in the queues for multislab $[s_1, s_3]$, and the part to the right of s_3 is inserted in the special list. It should be clear that after processing a multislab list in this way the remaining segments are non-intersecting (because every consecutive pair of segments are non-intersecting), and it will thus be consistently sorted. As we only scan through a multislab list ones the whole process can be done in a linear number of I/Os in the number of segments processed, plus the I/Os used to manipulate the priority queues.

Unfortunately, we still have inconsistencies in the node because segments in different multislab lists can intersect each other. Furthermore, the newly produced long segments in the priority queues can intersect each other as well as segments in the multislab lists. In order to remove the remaining intersections we need the following lemma.

Lemma 9 *If the minimal (top) segments of all the priority queues and the top segments of all the multislab lists are all non-intersecting, then the top-most of them is not intersected by any long segment in the queues or lists.*

Proof: First, consider the top segment in the two priority queues corresponding to the two boundaries of a single multislab. If these two segments do not intersect, then they must indeed

be the same segment. Furthermore, no other segment in these queues can intersect this top segment. Now consider the top segment in the multislab list of the same multislab. As the two segments are non-intersecting one of them must be completely above the other. This segment is not intersected by any segment corresponding to the same multislab. Now consider this top segment in all the multislabs. Pick one of the top-most of these non intersecting segments and call it s. Now consider Figure 10. Assume that s is intersected by another segment t in one of the queues or multislab lists. By this assumption t is not the top segment in its multislab. Call the top segment in this multislab u. Because u does not intersect either t or s, and as it is on top of t, it also has to be on top of s. This contradicts the assumption that s is above all the top segments. □

Our algorithm for finding and removing intersections now proceeds as follows. We repeatedly look at the top segment in each of the priority queues and multislab lists. If any of these segments intersect, we report the intersection and break one of the segments as before. If none of the top segments intersect we know from Lemma 9 that the topmost segment is not intersected at all. This segment can then be removed and stored in a list that eventually becomes the final multislab list for the slab in question. When we have processed all segments in this way, we end up with $O(m)$ sorted multislab list of non-intersecting segments. We have enough internal memory to buffer a block from each of the lists involved in the process, so we only need a number of I/Os linear in the number of segments processed (original and newly produced ones), plus the number of I/Os used to manipulate the priority queues.

Finally, as in Section 2.3.1, we produce the input to the next level of recursion by distributing the relevant segments (remembering to include the newly produced small segments) to the relevant children. As before, this is done in a number of I/Os linear in the number of segments processed. We stop the recursion when the number of *original* endpoints in the subproblems fall below $M/4$.

If the total number of intersections discovered in the construction process is T then the number of new segments produced is $O(T)$, and thus the number of segments stored on each level of the structure is bounded by $O(N + T)$. As in Section 2.3.1 we can argue that each segment is only contained in one list being sorted and thus we use a total of $O((n + t) \log_m(n + t)) = O((n + t) \log_m n)$ I/Os to sort the segments. In constructing each node we only use a linear number of I/Os, plus the number of I/Os used on priority queue operations. Since the number of priority queue operations is $O(T)$, the total number of of I/Os we use to construct the whole structure is bounded by $O((n + t) \log_m n)$.

3.2.2 Filtering Queries Through the Structure

We have now constructed an extended external segment tree on the N segments, and in the process of doing so we have reported some of the intersections between them. The intersections that we still have to report must be between segments stored in different nodes. In fact intersections involving segments stored in a node v can only be with segments stored in nodes below v or in nodes on the path from v to the root. Therefore we will report all intersections if, for all nodes v, we report intersections between segments stored at v and segments stored in nodes below v. But in v segments stored in nodes below v must be similar to the segments we called small in the red-blue line segment algorithm. Thus, if in each node v we had a list of endpoints of segments stored in nodes below v, sorted according to the long segment in v immediately on top of them, we could report the remaining intersections with

the algorithm that was used in Section 3.1.

In order to report the remaining intersections we therefore preprocess the structure and filter the endpoints of the $O(N+T)$ segments stored in the structure through the structure as we did in section 2.3.2. At each node the filtering process constructs a list of endpoints below the node sorted according to dominating segment among the segments stored in the node. At each node we can then scan this list to collect the relevant endpoints and then report intersections with the algorithm used in Section 3.1. For all nodes on one level of the structure the cost is linear in the number of segments and endpoints processed, that is, $O(n+t)$ I/Os, plus a term linear in the number of new intersections reported.

Recall that the preprocessing of the structure in Section 2.3.2 consisted of a sampling of every $(2\sqrt{m/4})$th segment of every slab in a node, which was then augmented to the segments stored in the child corresponding to the slab. The process was done from the root towards the leaves. We will do the same preprocessing here, except that because we decreased the fanout to $(m/4)^{1/4}$ we only sample every $(2(m/4)^{1/4})$th segment in a slab. However, as we are building the structure on intersecting segments we should be careful not to introduce intersections between segments stored in the multislab lists of a node when augmenting the lists with sampled segments. Therefore we do the preprocessing while we are building the structure. Thus, in the construction process described in the previous section, after constructing the sorted multislab lists of a node, we sample every $(2(m/4)^{1/4})$th segment in each slab precisely as in Section 2.3.2. We then send these segments down to the next level together with the other "normal" segments that need to be recursively stored further down the tree. However, we want to make sure that the sampled segments are not broken, but stored on the next level of the structure. Otherwise we cannot I/O-efficiently filter the query points through the structure, as the sampled segments are stored on the next level to make sure that the points are not to unsorted relative to the segments stored in a node. Therefore we give the sampled segments a special mark and make sure that we only break unmarked segments. We can do so because two marked segments can never intersect, otherwise they would have been broken on the previous level.

By the same argument used in Section 2.3.2 to prove Lemma 6 we can prove that the augmentation of sampled segments does not asymptotically increase the number of segments stored on each level of the structure. Also all the sampling and augmentation work can be done in a linear number of I/Os on each level of the structure. This means that the number of I/Os used to construct the structure is kept at $O((n+t)\log_m n)$, even when the preprocessing is done as an integrated part of it.

After the construction and preprocessing we are ready to filter the $O(N+T)$ endpoints through the $O(\log_m n)$ levels of the structure. Recall by referring to Figure 5 that in order to do the filtering we first sort the points by x coordinate and distribute them among the leaves. Then for each leaf in turn we find the dominating segments of the points assigned to the leaf and sort the points accordingly. Finally, the points are repeatedly filtered one level up until they reach the root.

The sorting of the points by x coordinate can be done in $O((n+t)\log_m(n+t)) = O((n+t)\log_m n)$ I/Os. Also each of the filtering steps can be done in a linear number of I/Os by the same argument as in Section 2.3.2 and the previous discussion. However, our structure lacks one important feature which we used in Section 2.3.2 to find dominating segments in the leaves. As in Section 2.3.2 we can argue that a leaf represents less than $M/4$ endpoints of the *original* segments, but as new segments and thus endpoints are introduced during the construction of the structure we cannot guarantee that the number of segments stored

in a leaf is less than $M/2$. Therefore, we cannot find the dominating segments by loading all segments stored in a leaf into internal memory and using an internal memory algorithm. Also, the segments stored in a leaf may intersect each other and we need to find and reports such intersections. However, assuming that we can report such intersections and produce the sorted list of endpoints for each leaf, the rest of the algorithm runs in $O((n+t)\log_m n + t')$ I/Os, where T' is the number of intersections found during the filtering of the endpoints through the structure. If $T_1 = T + T'$ is the total number of intersections reported then this number is clearly $O((n+t_1)\log_m n)$.

In order to overcome the problems with leaves containing more than M segments we do the following: We collect all the segments stored in such leaves. The number of collected segments must be less than $2T_1$ (actually less than $2T$), since the involved leaves contain more than M segments but less than $M/2$ of the original N segments. The same holds for the number of endpoints assigned to the leaves. We then recursively build an external extended segment tree on these segments and filter the relevant endpoints through the structure in order to report intersections between the segments and produce a list of the points sorted according to dominating segment. If we do not count the I/Os used to process the leaves in this tree this costs us $O((t_1+t_2)\log_m n)$ I/Os. Here T_2 is the number of intersections reported. Now we again need to collect the less than $2T_2$ segments in the leaves of the new tree containing more than M segments and recursively solve the problem for those segments. The process stops when all leaves contain less than M segments, and the total number of I/Os used on all the structure is then $O(n\log_m n + 2\sum_i t_i \log_m n) = O((n+t_t)\log_m n)$, where T_t is the total number of intersections reported. This completes our algorithm for the general segment intersection problem, giving us the following theorem:

Theorem 3 *All T intersections between N line segments in the plane can be reported in $O((n+t)\log_m n)$ I/O operations.*

4 Conclusions

In this paper, we have presented efficient external-memory algorithms for large-scale geometric problems involving collections of line segments in the plane, with applications to GIS systems. We have obtained these algorithms by combining buffer trees and distribution sweeping with a powerful new variant of fractional cascading designed for external memory.

The following two important problems, which are related to those we have discussed in this paper, remain open:

- If given the vertices of a polygon in the order they appear around its perimeter, can we triangulate the polygon in $O(n)$ I/O operations?
- Can we solve the general line segment intersection reporting problem in the optimal $O(n\log_m n + t)$ I/O operations?

Chapter 8

Optimal Dynamic Interval Management in External Memory

Optimal Dynamic Interval Management in External Memory*

Lars Arge†
BRICS‡
Dept. of Computer Science
University of Aarhus
Aarhus, Denmark

Jeffrey Scott Vitter‡
Dept. of Computer Science
Duke University
Durham, NC 27708–0129
USA

August 1996

Abstract

We present a space- and I/O-optimal external-memory data structure for answering stabbing queries on a set of dynamically maintained intervals. Our data structure settles an open problem in databases and I/O algorithms by providing the first optimal external-memory solution to the dynamic interval management problem, which is a special case of 2-dimensional range searching and a central problem for object-oriented and temporal databases and for constraint logic programming. Our data structure is simultaneously optimal in space and I/O, and it is the first optimal external data structure for a 2-dimensional range searching problem that has worst-case as opposed to amortized update bounds. Part of the data structure uses a novel balancing technique for efficient worst-case manipulation of balanced trees, which is of independent interest.

1 Introduction

In recent years there has been much effort in developing efficient external-memory data structures for *range searching*, which is a fundamental primitive in several large-scale applications, including spatial databases and geographic information systems (GIS) [52, 70, 84, 114, 124], graphics [60], indexing in object-oriented and temporal databases [79, 109], and constraint logic programming [78, 79]. Often the amount of data manipulated in such applications are too large to fit in main memory and must reside on disk, and in such cases the Input/Output (I/O) communication can become a bottleneck. NASA's EOS project GIS system [52] is an example of such an application, as it is expected to manipulate petabytes (thousands of terabytes, or millions of gigabytes) of data! The effect of the I/O bottleneck is getting more pronounced as internal computation gets faster, and especially as parallel computing gains

* An extended abstract version of this paper was presented at the 1996 IEEE Symposium on Foundations of Computer Science (FOCS'96).

† Supported in part by the ESPRIT Long Term Research Programme of the EU under project number 20244 (ALCOM–IT). Part of this work was done while a Visiting Scholar at Duke University. Email: large@brics.dk.

‡ Supported in part by the National Science Foundation under grant CCR–9522047 and by the U.S. Army Research Office under grant DAAH04–93–G–0076. Email: jsv@cs.duke.edu.

‡ Acronym for Basic Research in Computer Science, a Center of the Danish National Research Foundation.

popularity [107]. Currently, technological advances are increasing CPU speeds at an annual rate of 40–60% while disk transfer rates are only increasing by 7–10% annually [113].

In this paper we consider the special case of external 2-dimensional range searching called *dynamic interval management*, which is highlighted in [78, 79, 110, 121] as one of the important special cases of external range searching because of its applications in object-oriented databases and constraint logic programming. The problem of developing a space and I/O-time optimal external data structure for the problem is mentioned in [110] as one of the major theoretical open problems in the area, and in [79] it is even called "the most elegant open question." In this paper we develop such an optimal structure.

1.1 Memory model and previous results

We will be working in the standard model for external memory with one (logical) disk [82, 5]. We assume that each external-memory access transmits one page of B units of data, which we count as one I/O. We measure the efficiency of an algorithm in terms of the number of I/O operations it performs. Often one also makes the assumption that the main memory is capable of holding $O(B^2)$ units, that is, $O(B)$ blocks of data. Throughout most of this paper we also make this assumption, but in section 4 we show how to do without it.

While B-trees and their variants [21, 51] have been an unqualified success in supporting external dynamic 1-dimensional range searching, they are inefficient at handling more general problems like 2-dimensional or higher-dimensional range searching. The problem of 2-dimensional range searching in both main and external memory has been the subject of much research. Many elegant data structures like the range tree [24], the priority search tree [89], the segment tree [23], and the interval tree [53, 54] have been proposed for use in main memory for 2-dimensional range searching and its special cases (see [43] for a detailed survey). Most of these structures are not efficient when mapped to external memory. However, the practical need for I/O support has led to the development of a large number of external data structures that do not have good theoretical worst-case update and query I/O bounds, but do have good average-case behavior for common problems. Such methods include the grid file [95], various quad-trees [114, 115], z-orders [103] and other space filling curves, k-d-B-tress [111], hB-trees [86], cell-trees [68], and various R-trees [69, 117]. The worst-case performance of these data structures is much worse than the optimal bounds achievable for dynamic external 1-dimensional range searching using B-trees (see [79] for a complete reference on the field).

Recently some progress has been made on the construction of external 2-dimensional range searching structures with provably good performance. In [79] the *dynamic interval management* problem is considered, in which intervals can be inserted and deleted, and given a query interval all current intervals that intersect the query interval must be reported. This problem is crucial for indexing constraints in constraint databases and in temporal databases [78, 79, 109]. A key component of external dynamic interval management is answering *stabbing queries*. Given a set of intervals, to answer a stabbing query with a point q one has to report all intervals that contain q. By regarding an interval $[x, y]$ as the point (x, y) in the plane, a stabbing query with q reduces to the special case of two-sided 2-dimensional range searching called *diagonal corner queries* with corner (q, q) on the $x = y$ line, as shown in Figure 1. The metablock tree developed in [79] answers diagonal corner queries in optimal $O(\log_B N + T/B)$ I/Os using optimal $O(N/B)$ blocks of external memory, where T denotes the number of points reported. The structure is fairly involved and supports only insertions

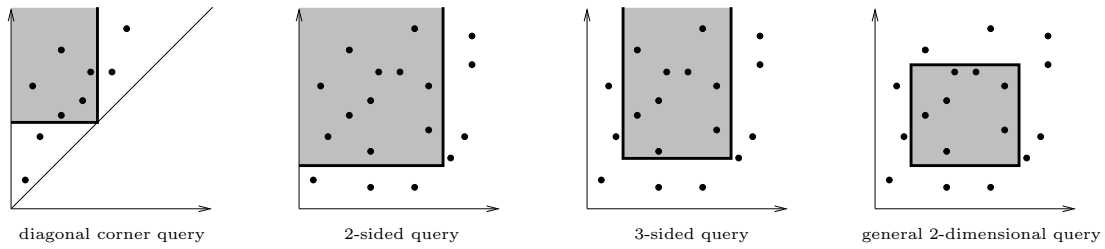


Figure 1: Different types of 2-dimensional range queries.

(not deletions) in $O(\log_B N + (\log_B N)^2/B)$ I/Os amortized.

As mentioned a number of elegant internal-memory solutions exist for 2-dimensional range searching. The priority search tree [89] for example can be used to answer slightly more general queries than diagonal corner queries, namely 3-sided range queries (Figure 1), in optimal query and update time using optimal space. A number of attempts have been made to externalize this structure, including [27, 74, 110], but they are all non-optimal. The structure in [74] uses optimal space but answers queries in $O(\log_2 N + T/B)$ I/Os. The structure in [27] also uses optimal space but answers queries in $O(\log_B N + T)$ I/Os. In both papers a number of non-optimal dynamic versions of the structures are also developed. In [110] a technique called *path caching* for transforming an efficient internal-memory data structure into an I/O efficient one is developed. Using this technique on the priority search tree results in a structure that can be used to answer 2-sided queries, which are slightly more general than diagonal corner queries, but slightly less general than 3-sided queries; see Figure 1. This structure answers queries in the optimal $O(\log_B N + T/B)$ I/Os and supports updates in amortized $O(\log_B N)$ I/Os, but uses slightly non-optimal $O((N/B)\log_2\log_2 B)$ space. Various other external data structures for answering 3-sided queries are also developed in [79] and in [110]. In [121] another attempt is made on designing a structure for answering 3-sided queries and a dynamic structure called the *p-range tree* is developed. The structure uses linear space, answers queries in $O(\log_B N + T/B + IL^*(B))$ I/Os and supports updates in $O(\log_B N + (\log_B N)^2/B)$ I/Os amortized. The symbol $IL^*(\cdot)$ denotes the iterated $\log^*$ function, that is, the number of times one must apply $\log^*$ to get below 2. It should be mentioned that the p-range tree can be extended to answer general 2-dimensional queries, and that very recently a static structure for 3-dimensional queries has been developed in [130].

The segment tree [23] can also be used to solve the stabbing query problem, but even in internal memory it uses more than linear space. Some attempts have been made to externalizing this structure [28, 110] and they all use $O((N/B)\log_2 N)$ blocks of external memory. The best of them [110] is static and answers queries in the optimal $O(\log_B N + T/B)$ I/Os.

1.2 Overview of our results

Our main results in this paper is an optimal external-memory data structure for the stabbing query problem. As mentioned, this result leads to the first known optimal solution to the interval management problem, and thus it settles an open problem highlighted in [79, 110, 121]. Our data structure uses $O(N/B)$ blocks of external memory to maintain a set of N intervals, such that insertions and deletions can be performed in $O(\log_B N)$ I/Os and such that stabbing queries can be answered in $O(\log_B N + T/B)$ I/Os. In Figure 2 we compare our result

	Space (blocks)	Query I/O bound	Update I/O bound
Priority search tree [74]	$O(N/B)$	$O(\log_2 N + T/B)$	
XP-tree [27]	$O(N/B)$	$O(\log_B N + T)$	
Metablock tree [79]	$O(N/B)$	$O(\log_B N + T/B)$	$O(\log_B N + (\log_B N)^2/B)$ amortized (inserts only)
P-range tree [121]	$O(N/B)$	$O(\log_B N + T/B + IL^*(B))$	$O(\log_B N + (\log_B N)^2/B)$ amortized
Path Caching [110]	$O((N/B)\log_2\log_2 B)$	$O(\log_B N + T/B)$	$O(\log_B N)$ amortized
Our Result	$O(N/B)$	$O(\log_B N + T/B)$	$O(\log_B N)$

Figure 2: Comparison of our data structure for stabbing queries with previous data structures.

with previous solutions. Note that unlike previous non-optimal solutions to the problem, the update I/O bounds for our data structure are worst-case. Previously no external data structure with worst-case update bounds was known for *any* 2-dimensional range searching problem. Also, as mentioned, our structure works without the assumption often made that the internal memory is capable of holding $O(B^2)$ items.

Our solution to the stabbing query problem is an external-memory version of the interval tree [53, 54]. In Section 2 we present the basic structure where the endpoints of the intervals stored in the structure belong to a fixed set of points. In Section 3 we then remove the fixed endpoint-set assumption. In internal memory the assumption is normally removed by using a BB[α]-tree [96] as base search tree structure [90], and this leads to amortized update bounds. But as BB[α]-trees are not very suitable for implementation in external memory, we develop a special *weight-balanced B-tree* for use in our external interval tree structure. Like in internal memory this results in amortized update bounds. In Section 4 we then show how to remove the amortization from the structure and how to avoid making the assumption about the size of the internal memory.

Our weight-balanced B-tree is of independent interest because we can use it in the design of other efficient external data structures and use it to remove amortization from other external-memory as well as internal-memory data structures. For example fixing B to a constant in our result yields an internal-memory interval tree with worst-case update bounds. Our B-tree structure can also be used as a (simpler) alternative to the rather complicated structure developed in [136] in order to add range restriction capabilities to internal-memory dynamic data structures. It seems possible to use the techniques in [136] to remove the amortization from the update bound of the internal interval tree, but our method is much simpler.

Finally, in Section 5 we discuss how to use the ideas behind our external interval tree to develop an external version of the segment tree with space bound $O((N/B)\log_B N)$. This improves upon previously known data structures [28, 110], which use $O((N/B)\log_2 N)$ blocks of external memory. Our structure has worst-case optimal query and update I/O bounds, whereas the other known structures are only query optimal in the static case. Fixing B to a constant yields an internal-memory segment tree (without the fixed endpoint set assumption) with worst case update bounds. Again we believe that our way of removing the amortization is simpler than a possible complicated use of the techniques in [136].

2 External-memory interval tree with fixed endpoint set

In this section we present the basic structure where the endpoints of the intervals have to belong to a fixed set. We also assume that the internal memory is capable of holding $O(B)$ blocks of data. As mentioned, we remove these assumptions in Sections 3 and 4.

2.1 Preliminaries

Our external-memory interval tree makes use of two kinds of secondary structures: the B-tree and a "corner structure" [79].

B-trees [21, 51] or more generally (a, b)-trees [73] are search tree structures suitable for external memory:

Lemma 1 *A set of N elements can be stored in a B-tree structure using $O(N/B)$ blocks of external memory such that updates and queries can be performed in $O(\log_B N)$ I/Os.*

A B-tree on N sorted elements can be built in $O(N/B)$ I/Os and the T smallest (largest) elements can be reported in $O(T/B + 1)$ I/Os

A "corner structure" [79] is a stabbing query data structure that is efficient when the number of segments stored in it is $O(B^2)$.

Lemma 2 (Kanellakis, Ramaswamy, Vengroff, Vitter [79]) *A set of $k \leq B^2$ segments can be represented using $O(k/B)$ blocks of external memory such that a stabbing query can be answered in $O(T/B + 1)$ I/O operations where T is the number of reported segments.*

As discussed in [79] the corner structure can be made dynamic by maintaining a special update block. Updates are inserted in this block and the structure is then rebuilt using $O(B)$ I/Os once B updates have been inserted. This leads to the following lemma:

Lemma 3 *A set of $k \leq B^2$ segments can be represented using $O(k/B)$ blocks of external memory such that a stabbing query can be answered in $O(T/B + 1)$ I/Os and an update can be performed in $O(1)$ I/Os amortized. The structure can be constructed in $O(k/B)$ I/Os.*

Note that the assumption on the size of the internal memory is (among other things) used to assure that all the segments in the corner structure fit in internal memory during a rebuilding process, that is, a rebuilding is simply done by loading the whole structure into internal memory, rebuild it, and write it back to external memory. In Section 4.2 we show how the update bounds can be made worst-case, and in that process we remove the assumption on the size of the internal memory. In that section it will also become clearer why the structure is called a "corner structure."

2.2 The structure

An internal-memory interval tree consists of a binary tree over the endpoints of the segments stored in the structure, with the segments stored in secondary structure in the internal nodes of this tree [53]. We associate an interval X_v with every node v consisting of all the endpoints below v. The interval X_r of the root r is thus divided into two by the intervals of its children, and we store a segment in r if it contains the "boundary" between these two intervals. Intervals on the left (right) side of the boundary are stored recursively in the left (right) subtree.

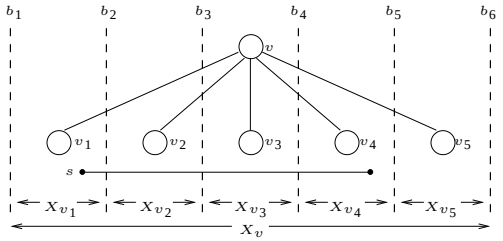


Figure 3: A node in the base tree

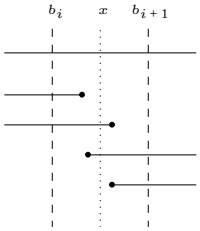


Figure 4: Reporting segments

Segments in r are stored in two structures: a search tree sorted according to left endpoints of the segments and one sorted according to right endpoints. To do a stabbing query with x we report the segments in r that contain x and then recurse to the subtree corresponding to x. If x is contained in the interval of r's left child, we find the segments in r containing x by traversing the segments in r sorted according to left endpoints, from the segments with smallest left endpoints toward the ones with largest left endpoints, until we meet a segment that does not contain x. All segments after this segment in the sorted order will not contain x.

When we want to externalize the interval tree structure and obtain a structure with height $O(\log_B N)$, we need to increase the fan-out of the nodes in the tree over the endpoints. This creates a number of problems when we want to store segments in secondary structures such that queries can be answered efficiently. The starting idea behind our successful externalization of the structure, as compared with previous attempts [27, 110], is that the nodes in our structure have fan-out $\sqrt{B}$ instead of B, following ideas from [11, 15]. The implications of this smaller fan-out are explained later in this section.

The external-memory interval tree on a set of intervals I with endpoints in a fixed set E is defined in the following way (we assume without loss of generality that no two intervals in I have the same endpoint, and that $|E| = (\sqrt{B})^i B$ for some $i \geq 0$): The *base tree* is a perfectly balanced tree over the endpoints E that has branching factor $\sqrt{B}$. Each leaf represents B consecutive points from E. As in the internal memory case we associate an interval X_v with each internal node v consisting of all endpoints below v. The interval X_v is further divided into the $\sqrt{B}$ subintervals associated with the children of v; see Figure 3. For illustrative purposes we call the subintervals *slabs* and the left (right) endpoint of such a slab a *slab boundary*. Finally, we define a *multislab* to be a contiguous range of slabs, such as for example $X_{v_2}X_{v_3}$ in Figure 3.

An internal node v now stores segments from I that cross one or more of the slab boundaries associated with v, but not any of the slab boundaries associated with v's parent. A leaf stores segments with both endpoints among the endpoints represented by the leaf. The number of segments stored in a leaf is less than $B/2$ and they can therefore be stored in one block. Each internal node v contains a number of secondary structures used to store the set of segments I_v stored in the node:

- For each of the $\sqrt{B}+1$ slab boundaries b_i, $1 \leq i \leq \sqrt{B}+1$,
  - A right *slab list* containing segments from I_v with right endpoint between b_i and b_{i+1}, sorted according to right endpoint.
  - A left *slab list* containing segments from I_v with left endpoint between b_i and b_{i-1}, sorted according to left endpoint.

– $O(\sqrt{B})$ *multislab lists*—one for each choice of a boundary to the right of b_i. The list for boundary b_j ($j > i$) contains segments from I_v with left endpoint between b_{i-1} and b_i and right endpoint between b_j and b_{j+1}. The list is sorted according to right endpoint of the segments.

- If the number of segments stored in a multislab list is less than $\Theta(B)$ we instead store them in a special *underflow structure* together with the segments from other multislab lists containing $o(B)$ segments.

  To be more precise, only if a multislab contains more than B segments we definitely store the segments in the multislab list. If the number of segments is between $B/2$ and B then the segments are either stored in the multislab list or in the underflow structure. Note that the underflow structure always contains less than B^2 segments.

An important point is that there are about $(\sqrt{B})^2/2 = B/2$ multislab lists associated with each internal node, since we used a fan-out of $\sqrt{B}$ (instead of B), and thus we have room in each internal node to store the pointers to its secondary data structures. The penalty for using a fan-out of $\sqrt{B}$ is only a factor of 2 in the height of the tree.

A segment is thus stored in two or three structures: two slab lists, and possibly in a multislab list or in the underflow structure. As an example, segment s in Figure 3 will be stored in the left slab list of b_2, in the right slab list of b_4, and in the multislab list corresponding to these two slab boundaries. Note the similarity between the slab lists and the sorted lists of segments in the nodes of an internal-memory segment tree. As in the internal case s is stored in two sorted lists—one for each of its endpoints. This represents the part of s to the left of the leftmost boundary contained in s, and the part to the right of the rightmost boundary contained in s. Unlike in the internal case, in the external case we also need to store/represent the part of s between the two extreme boundaries. This is done using the multislab lists.

All lists in a node are implemented using the B-tree structure from Lemma 1, and the underflow structure is implemented using the corner structure from Lemma 3. Finally, we maintain information like size and place of each of the $O(B)$ structures associated with a node in $O(1)$ "administration" blocks. The number of blocks used in a node that cannot be accounted for by segments (blocks that are not at least half-filled with segments) is $O(\sqrt{B})$: $O(1)$ administration blocks, one block for each of the $2\sqrt{B}$ slab lists, and one for the underflow structure. As each leaf contains B endpoints the tree has $O(|E|/B\sqrt{B})$ internal nodes, which again means that we use a total of $O(|E|/B)$ blocks to store the whole structure. Note that if we did not store segments belonging to a multislab having $o(B)$ segments in the underflow structure, we could have $\Omega(B)$ sparsely utilized blocks in each node, which would result in a non-optimal space bound.

2.2.1 Operations on the structure

In order to do a stabbing *query* with x on our external interval tree we search down the structure for the leaf containing x. At each internal node we pass in this search we should report the relevant segment among the segments stored in the node. Let v be one such node and assume that x falls between slab boundary b_i and slab boundary b_{i+1}. We do the following on v's secondary structures:

- We load the $O(1)$ administration blocks.
- We load and report the segments in all multislab lists that contain segments that cross b_i and b_{i+1}.
- We do a stabbing query with x on the underflow structure and report the result.
- We report segments from b_i's right slab list from the largest toward the smallest (according to right endpoint) until we meet a segment that does not contain x.
- We report segments from b_{i+1}'s left slab list from the smallest toward the largest until we meet a segment that does not contain x.

It is easy to realize that this process will indeed report all the relevant segments. All segments stored in v that contain x are either stored in one of the relevant multislab lists, in the underflow structure, or in the right slab list of b_i or the left slab list of b_{i+1}; refer to Figure 4. Our algorithm correctly reports all the relevant segments in the right slab list of b_i, because if a segment in the right-to-left order of this list does not contain x then neither does any other segment to the left of it. A similar argument holds for the left-to-right search in the left slab list of b_{i+1}. Again note the similarity with the internal interval tree.

The query algorithm uses an optimal number of I/O operations, as can be demonstrated as follows: The search with x in the base structure uses $O(\log_{\sqrt{B}} N) = O(\log_B N)$ I/Os. In each node on the search path we only use $O(1)$ I/Os that are not "paid for" by reportings (blocks read that contain $\Theta(B)$ output segments): We use $O(1)$ I/Os to load the administration blocks, $O(1)$ overhead to query the underflow structure, and $O(1)$ overhead for each of the two searches in the slab lists. The underflow structure is again crucial because it means that all blocks loaded in order to report the segments in the relevant multislab lists contain $\Theta(B)$ segments.

Lemma 4 *There exists a data structure using $O(N/B)$ blocks of external memory to store segments with endpoints among a set E of size N, such that stabbing queries can be performed in $O(\log_B N + T/B)$ I/Os in the worst case.*

In order to *insert* a new segment s into the structure we do the following: We search down the base tree until we find the first node where s crosses one or more slab boundaries. Then we load the $O(1)$ administration blocks and insert s into the two relevant slab lists. If the multislab list that we want to insert s into exists, we also insert s into that list. Otherwise the other segments (if any) belonging to s's multislab list are stored in the underflow structure and we insert s in this structure. If that brings the number of segments belonging to the same multislab as s up to B, we delete them from the underflow structure by rebuilding it totally, and create a new multislab list with the B segments. Finally, we update and store the administration blocks.

In order to *delete* a segment s we again search down the structure until we find the node where s is stored. We then delete s from the two relevant slab lists. If s is stored in the underflow structure (we can figure that out by checking if the multislab list for s exists) we just delete it from the structure. Otherwise we delete s from the multislab list it is stored in. If the number of segments in this list falls below $B/2$ by the deletion of s, we remove the list and insert all the segments in the underflow structure. The latter is done by rebuilding the underflow structure. Finally, we again update and store the administration blocks.

To analyze the I/O usage of an update, note that for both inserts and deletes we use $O(\log_B N)$ I/Os to search down the base tree, and then we in *one* node use $O(\log_B N)$ I/Os to update the secondary list structures. The manipulation of the underflow structure uses $O(1)$ I/Os amortized, except in the cases where the structure is rebuilt, where it uses $O(B)$ I/Os. In the latter case there must have been at least $B/2$ "inexpensive" updates involving segments in the same multislab as s since the time of the last rebuilding; by "inexpensive" we mean updates where no rebuilding is done. Hence the amortized I/O use remains $O(1)$. To summarize, we have shown the following:

Theorem 1 *There exists a data structure using $O(N/B)$ blocks of external memory to store segments with endpoints among a set E of size N, such that stabbing queries can be performed in $O(\log_B N + T/B)$ I/Os worst-case and such that updates can be performed in $O(\log_B N)$ I/Os amortized.*

3 General external-memory interval tree

In order to make our interval tree "dynamic" (that is, to remove the assumption that the intervals have endpoints in a fixed set E) we will have to use a dynamic search tree as base tree. There are many possible choices for such a tree: height-balanced, degree-balanced, weight-balanced, and so on. The different choices of dynamic tree lead to different balancing methods, but normally (and also in this case) when search trees are augmented with secondary structures it turns out that a rebalancing operation requires work proportional to the number of items in the subtree with the node being rebalanced as root. In internal memory a natural choice of dynamic base tree is therefore the BB[α]-tree [96], because in this structure a node with w items below it (with "weight" w) can only be involved in a rebalancing operation for every $\Omega(w)$ updates that access (pass through) the node [30, 90]. This leads to an $O(1)$ amortized bound on performing a rebalancing operation. Unfortunately BB[α]-trees are not suitable for implementation in external memory. The main problem is of course that BB[α]-trees are binary and that there does not seem to be an easy way of grouping nodes together in order to increase the fan-out while maintaining the other useful properties of BB[α]-trees.

B-trees on the contrary are very suitable for implementation in external memory, so they are a natural choice as dynamic base structure. However, these trees do not have the property that a node of weight w can only be involved in a rebalance operation for every $\Omega(w)$ updates that access the node. Therefore we first develop an elegant variant of B-trees, which we call *weight-balanced B-trees*, that possesses the needed property. A node in a normal B-tree can have subtrees of widely varying weights. The ratio between the largest subtree weight and the smallest one can be exponential in the height of the node. In our weight-balanced B-tree this ratio is a small constant factor. Fortunately, as we shall see later, the rebalancing operations on the weight-balanced B-tree (splits of nodes instead of rotations) allow us to spread the rebalancing work over a number of updates in an easy way, and thus we can remove the amortization from the update bounds.

3.1 Weight-balanced B-tree

In a normal B-tree [21, 51] all leaves have the same depth, and each internal node has at least a and at most $2a - 1$ children, where a is some constant. In weak B-trees or (a, b)-trees [73] a wider range in the number of children is allowed. Here we define the weight-balanced B-tree

by imposing constraints on the weight of subtrees rather than on the number of children. The other B-tree characteristics remain the same: The leaves are all on the same level (level 0), and rebalancing is done by splitting and fusing internal nodes.

Definition 1 *The weight $w(v_l)$ of a leaf v_l is defined as the number of elements in it. The weight of an internal node v is defined as $w(v) = \sum_{v=parent(c)} w(c)$.*

Lemma 5 *The weight $w(v)$ of an internal node v is equal to the number of elements below v.*

Proof: Easy by induction. □

Definition 2 *We say that T is a* weight-balanced B-tree *with branching parameter a and leaf parameter k, where $a > 4$ and $k > 0$, if the following conditions hold:*

- *All leaves of T have the same depth and weight between k and $2k - 1$.*
- *An internal node on level l has weight less than $2a^l k$.*
- *An internal node on level l except for the root has weight greater than $\frac{1}{2}a^l k$.*
- *The root has more than one child.*

These definitions yield the following lemma:

Lemma 6 *All nodes in a weight-balanced B-tree with parameters a and k except for the root have between $a/4$ and $4a$ children. The root has between two and $4a$ children.*

Proof: The leaves actually fulfill the weight constraint on the internal node, since $k > \frac{1}{2}a^0 k$ and $2k - 1 < 2a^0 k$. The minimal number of children an internal node on level l can have is $\frac{1}{2}a^l k / 2a^{l-1}k = a/4$, and the maximal number of children is $2a^l k / \frac{1}{2}a^{l-1}k = 4a$. For the root the upper bound follows from the same argument and the lower bound is by definition. □

The following observation about weight-balanced trees assures that they are balanced:

Lemma 7 *The height of a weight-balanced B-tree with parameters a and k on N elements is $O(\log_a(N/k))$.*

Proof: Follows directly from Definition 2 and Lemma 6. □

In order to do an update on the weight-balanced B-tree we search down the tree to find the leaf in which to do the update. Then we do the actual update, and we may then need to rebalance the tree in order to make it satisfy Definition 2. For simplicity we only consider inserts in this section. In the next section, where we will use the weight-balanced B-tree as base tree in our external interval tree, we discuss how deletes can be handled using the global rebuilding technique [105]. After finding the leaf w to insert a new elements into, we do the following: We insert the element in the sorted order of elements in w. If w now contains $2k$ elements we spilt it into two leaves w and w', each containing k elements, and insert a reference to w' in $parent(w)$. After the insertion of the new element the nodes on the path from w to the root of T can be out of balance; that is, the node v_l on level l can have weight $2a^l k$. In order to rebalance the tree we therefore split all such nodes starting with the nodes

on the lowest level and working toward the root. In the case of v_l we want to split it into two nodes v_l' and v_l'' of weight $a^l k$ and insert a reference to the new node in $parent(v_l)$ (if $parent(v_l)$ does not exists, that is, if we are splitting the root, we also create a new root with two children). But a perfect split is not generally possible if we want to split between two of v_l's children, such that v_l' gets the first (leftmost) i children for some i and v_l'' gets the rest of the children. However, we can always find an i such that if we split at the ith child the weights of both v_l' and v_l'' will be between $a^l k - 2a^{l-1}k$ and $a^l k + 2a^{l-1}k$, since nodes on level l have weight less than $2a^{l-1}k$. As $a > 4$, the nodes v_l' and v_l'' now fulfill the constraints of Definition 2; that is, their weights are between $\frac{1}{2}a^l k$ and $2a^l k$. Note that if we strengthen the assumption about a to be $a > 8$, we can even split the node at child $i-1$ or $i+1$ instead of at child i, and still fulfill the constraints after doing the split. We will use this property in the next section.

Because of the definition of weight the split of a node v does not change the weight of $parent(v)$. This means that the structures are relatively simple to implement, as each node only need to know on which level it is and the weight of each of its children. This information can easily be maintained during an insertion of a new element. The previous discussion and Lemma 7 combine to prove the following:

Lemma 8 *The number of rebalancing operations (splits) after an insertion of a new element in a weight-balanced B-tree T with parameters a and k is bounded by $O(\log_a(|T|/k))$.*

The following lemma now states the fact that will become crucial in our application.

Lemma 9 *After a split of a node v_l on level l into two nodes v_l' and v_l'', at least $a^l k/2$ inserts have to pass through v_l' (or v_l'') to make it split again. After a new root r in a tree containing N elements is created, at least $3N$ inserts have to be done before r splits again.*

Proof: After a split of v_l the weights of v_l' and v_l'' are less than $a^l k + 2a^{l-1}k < 3/2a^l$, as $a > 4$. Each node will split again when its weight reaches $2a^l k$, which means that its weight (the number of inserts passing through it) must increase by $a^l k/2$. When a root r is created on level l it has weight $2a^{l-1}k = N$. It will not split before it has weight $2a^l k > 2 \cdot 4a^{l-1}k = 4N$. □

As mentioned in the introduction one example of how the weight-balanced B-tree can be used as a simpler alternative to existing *internal-memory* data structures, is in adding range restriction capabilities to dynamic data structures as described in [136]. We will not go into details with the construction, but refer the interested reader to [136]. The essence in the construction is to have a base search tree and then augment every internal node of this tree with another dynamic data structure on the set of items in the leaves below the node. The critical point in the construction is to choose a base tree structure such that one does not have to do too much work on the reconstruction of the secondary structures during rebalancing of the base tree. Using a BB[α]-tree as the base tree results in good amortized update bounds as discussed, and in [136] it is shown how good worst-case bounds can be obtained by a complicated redefinition of the BB[α]-tree and the operations on it. However, using our weight-balanced B-tree with branching parameter $a = 4$ and leaf parameter $k = 1$ as base tree in the construction, it is easy to obtain good worst case bounds. The result follows from the fact that rebalancing in our tree is done by splitting nodes and the fact that $\Omega(w)$ updates have to pass through a node of weight w between updates which cause

rebalancing operations involving the node (Lemma 9). The large number of updates between splits of a node immediately implies the good amortized bound, and using the global rebuilding technique [105], where the rebuilding of the secondary structures is done lazily over the next $O(w)$ updates that access the node, results in the worst-case bound. The problem in using the global rebuilding technique directly on the BB[α]-tree is that rebalancing on this structure is done by rotations, which means that we can not simply continue to query and update the "old" secondary structure while we are lazily building new ones. The ideas and techniques used in our construction is very similar to the ones presented in the succeeding sections of this paper and are therefore omitted.

In order to obtain an external-memory search tree structure suitable for use in our "dynamic" interval tree we choose $4a = \sqrt{B}$ and $2k = B$ and obtain a structure with properties as described in the following theorem.

Theorem 2 *There exists a search tree data structure that uses $O(N/B)$ blocks of external memory to store N elements. A search or an insertion can be performed in $O(\log_B N)$ I/Os in the worst case.*

Each internal node v in the data structure, except for the root, has $\Theta(\sqrt{B})$ children. If v is at level l these children divide the $\Theta((\sqrt{B})^l B)$ elements below v into $\Theta(\sqrt{B})$ sets. Rebalancing is done by splitting nodes. When a node v is split, at least $\Theta((\sqrt{B})^l B)$ elements must have been inserted below v since v was created or since the last time v was split. The number of elements below the root is N. In order for a new root to be created, $\Theta(N)$ elements have to be inserted into the data structure.

Proof: Each internal node can be represented using $O(1)$ blocks. As the number of internal nodes is smaller than the number of leaves, the space bound follows from the fact that each leaf contains $\Theta(B)$ elements. A split can be performed in $O(1)$ I/Os: We simply load the $O(1)$ blocks storing the node into internal memory, compute where to split the node, and write the $O(1)$ blocks defining the two new nodes with updated information back to external memory. Finally, we update the information in the parent in $O(1)$ I/Os. This means that the insertion and search I/O bound follow directly from Lemmas 7 and 8.

The second part of the theorem follows directly from Definition 2 and Lemma 5, 6, and 9. □

3.2 Using the weight-balanced B-tree to dynamize our data structure

As discussed earlier we now make our external interval tree "dynamic" by replacing the static base tree in Section 2 with the newly developed weight-balanced B-tree. To insert a segment we first insert the two new endpoints in the base tree and perform the necessary rebalancing, and then we insert the segment as described in Section 2. If we can do the rebalancing in $O(\log_B N)$ I/Os we end up with an insert operation with the same I/O bound. As rebalancing is done by splitting nodes we need to consider how to split a node when it contains secondary structures. As previously discussed the key property we will use is that a node of weight w can only be involved in a rebalance operation for every $\Omega(w)$ update that accesses the node. Our goal will therefore be to split a node in $O(w)$ I/Os.

Figure 5 shows that when v is split into two new nodes, what really happens is that v's slabs split along a slab boundary b such that v' gets slabs on one side of b and v'' gets the rest. Then b is inserted among $parent(v)$'s slab boundaries. Now consider Figure 6 to realize

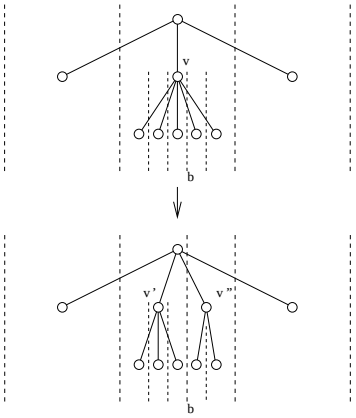


Figure 5: Splitting a node

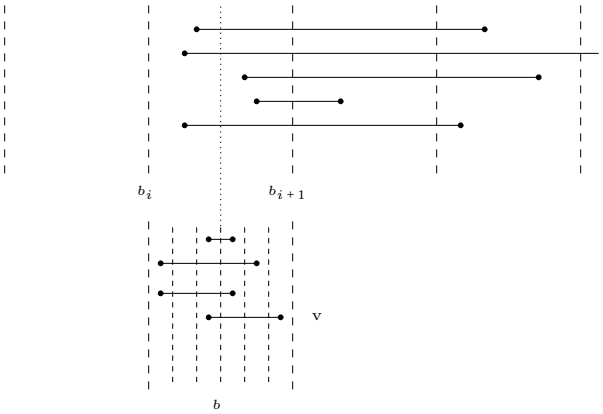


Figure 6: Moving segments

that when we split v we need to move a number of segments, which fall into two categories: segments stored in v that contain b and (some of the) segments stored in *parent*(v) that contain b. The first set of segments need to be removed from v's secondary structures and inserted in the appropriate secondary structures of *parent*(v). The second set of segments need to be moved from some of *parent*(v) structures to some other new ones. We also need to split v's underflow structure (and the administration blocks) to complete the split. Note that the number of segments that need to be moved is bounded by the weight of v, since they all have one endpoint in X_v.

First consider the segments in v. To remove segments containing b from the right slab lists of v, and collect them sorted according to right endpoint, we do the following for every right slab list of v: We scan through the list (leaves of the B-tree) and output each segment to one of two lists, one containing the segments that should stay in v and one containing the segments that should move. Then we destroy the slab list (B-tree) and build a new one from the segments that should stay in the list. After collecting the segments to be moved from each of the right slab lists in this way, we can produce the total list of segments to be moved sorted according to right endpoint, simply by appending the lists after each other starting with the list from the leftmost slab boundary. In a similar way we scan through the left slab lists to remove the same set of segments and produce a list of them sorted according to left endpoint. Call the two list produced in this way L_r and L_l. The segments in L_r and L_l need to be inserted in the slab lists of b in *parent*(v), but as also some segments from *parent*(v) might need to be inserted in these two lists, we postpone the insertion until we have moved all the relevant segments in *parent*(v). According to Lemma 1 we can scan through all the K elements in a slab list (B-tree) in sorted order, and build a new slab list on K sorted elements, in $O(K/B)$ I/Os; hence, we can perform the above process in $O(|X_v|/B)$ I/Os. We also need to delete the segments in L_l and L_r from the relevant multislab lists and the underflow structure. Note that if just one segment needs to be removed from a multislab list all segments in the list need to be removed. Therefore we simply remove all multislab lists that contain b, again using $O(|X_v|/B)$ I/Os. Finally, we split the underflow structure and at the same time we delete the segments containing b from the structure. This is done by loading it into internal memory and for each of the two new nodes build a new structure on the relevant segments. If the structure contains k segments we by Lemma 3 use $O(k/B)$

I/Os to do this, so in total we use $O(|X_v|/B)$ I/Os on the whole process.

Now consider Figure 6 again. The segments we need to consider in $parent(v)$ all have one of their endpoints in X_v. For simplicity we only consider segments with *left* endpoint in this interval. Segments with right endpoint in the interval can be handled similarly. All segments with left endpoint in X_v stored in $parent(v)$ cross boundary b_{i+1}. This means that we need to consider each of these segments in one or two of $\sqrt{B}+1$ lists, namely in the left slab list of b_{i+1} and maybe in one of $\sqrt{B}$ multislab lists. When we introduce the new boundary b, some of the segments in b_{i+1}'s left slab list need to be moved to the new left slab list for b. Therefore we scan through the list and produce two new lists, one for b_{i+1} and one for the new boundary b, just like when we processed a slab list in v. We merge the list for b with L_l and build a B-tree on all the segments. As all segments touched in this process have an endpoint in $|X_v|$, we use $O(|X_v|/B)$ I/Os to perform it. Similarly some of the segments in multislabs with b_{i+1} as left boundary need to be moved to new multislabs corresponding to the new boundary. Note that some of them might be stored in the underflow structure and that we do not need to move such segments. To move the relevant segments we scan through each of the relevant multislab lists and split them in two as previously. The key property is now that *all* segments stored in the $\sqrt{B}$ lists in question have their left endpoint in X_v. This means that the total size of the lists is $O(|X_v|)$ and we can again complete the process in $O(|X_v|/B)$ I/Os. If any of the multislab lists contain less than $B/2$ segments after this process, we destroy them and insert the segments in the underflow structure. This is again done by rebuilding it totally using $O(|X_v|/B)$ I/Os. Finally, to complete the split process we update the administration blocks of both v and $parent(v)$.

To summarize, we have shown how to split a node v in $O(|X_v|/B)$ I/Os. If v is at level l, $|X_v|$ is $\Theta((\sqrt{B})^l B)$ according to Theorem 2, which means that we use $O((\sqrt{B})^l)$ I/Os. Also according to Theorem 2, $O((\sqrt{B})^l B)$ is the number of inserts that must pass the node between splits. Thus we only use $O(1/B)$ I/Os amortized to split a node, which again, by the discussion in the beginning of this section, leads to the following lemma:

Lemma 10 *There exists a data structure that uses $O(N/B)$ blocks of external memory to store N segments, such that stabbing queries can be answered in $O(\log_B N + T/B)$ I/Os and such that a new segment can be inserted in $O(\log_B N)$ I/Os amortized.*

Now as mentioned in Section 3.1 we can also handle deletions by using the global rebuilding technique [105]. When we want to delete a segment s, we delete it from the secondary structures as described in Section 2 *without* deleting the endpoints of s from the base tree. Instead we just mark the two endpoints in the leaves of the base tree as deleted. This does not increase the number of I/Os needed to do a later update or query operation, but it does not decrease it either. We can afford being lazy with respect to deletes, since the query and update I/O bounds remain of the same order even if we do not do any rebalancing in connection with deletes for up to, say, $N/2$ delete operations (Note that such a lazy approach does not work for inserts). When we have performed too many deletes and the number of undeleted endpoints becomes less than $N/2$, we simply rebuild the whole structure. It is easy to realize that we can rebuild the structure in $O(N \log_B N)$ I/Os, which leads to an $O(\log_B N)$ amortized delete I/O bound: First we scan through the leaves of the old base tree and construct a sorted list of the undeleted endpoints. This list can then be used to construct the leaves of the new base tree, and the rest of the tree can easily be build on top of these leaves. All of this can be done in $O(N/B)$ I/Os. Finally, we in $N/4 \cdot O(\log_B N)$ I/Os insert

the $N/4$ segments stored in the old structure in the new data structure. This completes our description of the external-memory interval tree:

Theorem 3 *There exists an external interval tree that uses $O(N/B)$ blocks of external memory to store N segments, such that stabbing queries can be answered in $O(\log_B N + T/B)$ I/Os worst case and such that updates can be performed in $O(\log_B N)$ I/Os amortized.*

4 Removing amortization

In this section we show how we can make the update bound of our structure worst-case instead of amortized. First of all note that the amortization in Theorem 3 follows from three amortized results: the amortized updates of the underflow structure (Lemma 3), the amortization resulting from the splitting of nodes in the base tree, and the global rebuilding (allowing deletions) used in the last section. The last use of amortization can be made worst case using the standard global rebuilding technique. The general idea is the following (see [105] for details): Instead of using $O(N \log_B N)$ I/Os all at once to rebuild the entire structure when the number of endpoints falls below $N/2$, we distribute the rebuilding over the next $1/3 \cdot N/2$ updates, that is, we still use and update the original structure and in parallel we build the new structure three times as fast (we perform something like $3 \log_B N$ work on the new structure at each update, keeping the update cost at $O(\log_B N)$). When the new structure is completed we still need to perform the $1/3 \cdot N/2$ updates that occurred after we started the rebuilding. But again this can be done in parallel during the next $1/3 \cdot (1/3 \cdot N/2)$ operations. This continues until we finally reach a state where both trees store the same set of segments (with at least $(1-(1/3+1/9+\cdots))N/2 \geq 1/2 \cdot N/2$ endpoints). Then we dismiss the old structure and use the new one instead. Because we finish the rebuilding before the structure contains $N/4$ endpoints we are always busy constructing at most one new structure.

This leaves us with the amortization introduced via the underflow structure and the splitting of nodes in the base tree. In Section 4.2 we repeat the definition of the "corner structure" [79] which was used as underflow structure, and show how to do updates on this structure in $O(1)$ I/Os in the worst case; that is, we show how to remove the amortization from Lemma 3. This allows us to remove the amortization from the "static" version of our interval tree summarized in Theorem 1. Recall that the amortization, apart from the amortization introduced in Lemma 3, in the update bounds of this theorem was introduced because of the rebuilding of the underflow structure—if an insertion resulted in the underflow structure containing B segments belonging to the same multislab list, or if a deletion resulted in the number of segments in a multislab list falling below $B/2$. To remove this amortization we do the rebuilding in a lazy way over the next $B/4$ updates that involve the multislab list in question. If the size of a multislab list falls to $B/2$ we start moving the segments to the underflow structure. When a new segment s is to be inserted in the list, we insert s in the underflow structure and move two of the $B/2$ segments from the multislab list to the underflow structure. On deletion of a segment s belonging to the multislab, we delete s (from the list or the underflow structure) and move two segments again. This way we are done moving the segments after at most $B/4$ updates, which means that when we are done there is between $\frac{1}{4}B$ and $\frac{3}{4}B$ segments belonging to the multislab in question stored in the node. As they are now all stored in the underflow structure we are back in a situation fulfilling the definition of an external interval tree. During the moving process we maintain the optimal space and query bound, even though there during the process may be a multislab list with

very few segments in it (still occupying one block of memory). This is because there is $\Theta(B)$ segments stored in the underflow structure that can "pay" for the memory-use and for the I/O a query will use to report these few segments. Similarly, if the number of segments in the underflow structure belonging to the same multislab reaches B, we create the multislab list and move the B segments during the next $B/4$ updates involving the list. In Section 4.2 it will become clear how we can easily find and delete the relevant segments in the underflow structure in this lazy way. The optimal space and query bounds follow as before.

Finally, we in Section 4.1 show how the splitting of a node can be done in $O(1)$ I/Os in the worst case instead of amortized, by showing how we can distribute the splitting, or rather the movement of segments associated with a split, over the updates that pass through the node before it splits again. Altogether this removes all the amortization and leads to our main result.

Theorem 4 *There exists an external interval tree that uses $O(N/B)$ blocks of external memory to store N segments such that stabbing queries can be answered in $O(\log_B N + T/B)$ I/Os and such that updates can be performed in $O(\log_B N)$ I/Os. All bounds are worst case.*

4.1 Splitting nodes lazily

In this section we how to split a node in the external interval tree in $O(1)$ I/Os in the worst case. Recall by referring to Figure 6 that we move two "types" of segments when v splits along b: Segments stored in v that contain b are moved to *parent*(v), and some of the segments stored in *parent*(v) that contain b are moved internally in *parent*(v). In Section 3 we used the fact stated in Theorem 2 that if we split a node on level l it has $O((\sqrt{B})^l B)$ endpoints below it (or segments stored in its secondary structures), and the same number of updates has to pass through it before it splits again. We showed that we could move the relevant segments in $O((\sqrt{B})^l)$ I/Os which lead to the $O(1)$, or rather $O(1/B)$, amortized splitting bound. In this section we show how we can move the necessary segments in a lazy way, such that when a node is split we use a constant number of I/Os on splitting the node during (some of) the next $O((\sqrt{B})^l B)$ updates (not $O((\sqrt{B})^l)$ as in Section 3) that access the node. In this way the update cost is kept at $O(\log_B N)$ I/Os in the *worst case*, and when a node splits we are sure that we have finished the necessary movements triggered by the last split of the node.

Basically our algorithm works just like in the amortized case, except that we do the movement lazily: When a node v needs to be split along a boundary b, we first insert b as a *partial* slab boundary in *parent*(v) and let the new reference created in this way point to v, that is, v plays the role of both v' and v'' on Figure 5. The boundary b remains partial in *parent*(v) until we have finished the movement of segments. As we can only be working on splitting a node along one boundary at a time, a partial slab boundary is always surrounded by two real boundaries, or equivalently, at most every second boundary in a node can be partial. As discussed in Section 3.1 this means that we can always split a node along a real boundary without violating the constraints on the base weight-balanced B-tree. We only split a node along a real boundary in order to keep different split processes from interfering with each other (as we shall see later). After creating the partial boundary in *parent*(v) we then like previously remove segments in v containing b by "splitting" each of the slab lists of v, and create two sorted lists L_l and L_r with these segments. Below we show how to do this in a lazy way using $O(1)$ I/Os over the next $O((\sqrt{B})^l)$ updates that access the node. We call this the *up phase*. Basically, we use an idea similar to global rebuilding. We rebuild the relevant

structures while we still query and update the "old" structures as normally. During this phase queries are done on *parent*(v) simply by ignoring the partial slab boundary b. When we after $O((\sqrt{B})^l)$ updates finish the up phase, we in $O(1)$ I/Os split v into v' and v'' by switching to the new structures and splitting the administration blocks. Next we move the segments in *parent*(v) that contain b in a lazy way over the next $O((\sqrt{B})^l)$ updates that access this node, again basically by rebuilding the relevant structures while we still update the old structures as normally. We call this the *rearrange phase* and describe it in detail below. If a query access the node during the rearrange-phase, we need to use as slightly modified query algorithm to report all the relevant segments. Consider Figure 6 again and a query x between b_i and b_{i+1} in *parent*(v). To correctly report the segments stored in *parent*(v) we first ignore the partial slab boundary b and query the node as previously. Then we in order to correctly report segments among the segments we have removed from v and inserted in the slab list of b (L_l and L_r) query the relevant of the two slab lists, that is, if x for example falls between b_i and b we query the left slab list of b (L_l). As we maximally do one such extra query on each level of the base tree—using $O(1)$ extra I/Os—our query I/O bound remains the optimal $O(\log_B N + T/B)$. Finally, when we after the $O((\sqrt{B})^l)$ updates finish the *rearrange phase*, we in $O(1)$ I/Os switch to the new structures and make b a normal slab boundary.

As explained above a node v that splits first trigger an up phase on itself and then a rearrange phase on *parent*(v). During these phases we will assume that only one such phase is working on a node at a time. This means that when we want to perform one of the phases on a node, we might need to wait for another phase to finish. Actually a number of phases may be waiting, and we may need to wait until all of them finish. Recall that our goal is to split v lazily in $O((\sqrt{B})^l B)$ I/Os, because we are then finished doing the split before the next split can occur. The up and rearrange phases only require $O((\sqrt{B})^l)$ updates accessing the node, so we can afford to wait for quite some time. Consider a node v that we want to split. If a rearrange-phase is not currently in progress on v we just start our up-phase. Otherwise, if a rearrange phase is in progress other such phases might be waiting and we will wait until all of them have finished. Now the number of waiting phases is bounded by the number of partial slab boundaries in v, which again is bounded by $\sqrt{B}$. This means that we maximally will have to wait for $\sqrt{B} \cdot O((\sqrt{B})^l)$ updates. We are willing to do so because this number is $O((\sqrt{B})^l B)$. While we are waiting, or while we are performing the up phase, new partial slab boundaries may be inserted in v because of a split of one of v's children, and new slab lists for such boundaries may be inserted when an up phase finishes. The latter will cause the need for a new rearrange phase which will be performed after we finish our (waiting) up phase. Note that inserting the new slab lists for a partial boundary b in the middle of another up or rearrange phase does not create any problems, because segments inserted this way cannot be involved in any such phase (because they all have both endpoints between b_i and b_{i+1} and because we do not split along partial boundaries). Also note that as the number of waiting processes is $O(\sqrt{B})$, information about them can be stored in one block. After finishing the up phase on v we might again need to wait for other phases to finish before we can do a rearrange phase on *parent*(v). But again we must wait for at most $\sqrt{B}$ rearrange phases which takes $O((\sqrt{B})^l)$ updates each, and for one up phase which takes $O((\sqrt{B})^{l+1})$ updates. This means that we can finish the split in the allowed $O((\sqrt{B})^l B)$ updates.

Before finally describing the up and rearrange phases we make a last modification of our structure. During updates we keep a *shadow* structure of every secondary structure, which is a precise copy of the original structure. We update a given of these shadow structures precisely as the original structure it is a copy of, and query the original structures as normally.

The shadow structures do not change the update or query I/O bounds. Using the shadow structures we can now do an up or rearrange phase in the following simple way using global rebuilding: We start a phase on v by "freezing" the shadow structures of v, that is, we stop doing updates on them and instead we just store the updates that arrive. Then we remove the necessary segments from the *shadow* structures and rebuild it using $O((\sqrt{B})^l)$ I/Os, just like when we rebuild the original structures in Section 3 (except that we use the new algorithms we will present in section 4.2 to manipulate the underflow structure). It is easy to realize that we can spread the rebuilding over the next $O((\sqrt{B})^l)$ update that access the node using $O(1)$ I/Os on each updates. There is one complication however, as some of the updates might actually update the structures of v, which means that we also have to update the shadow structure we are rebuilding. Therefore if an update arrives which needs to be performed on the shadow structure, we instead of $O(1)$ do $O(\log_B N)$ work on the rebuilding. This does not change the update I/O bound as we are already using $O(\log_B N)$ I/Os to perform the update on the original structures. Then we store the update and perform it lazily on the rebuild structures (again using $O(1)$ I/Os on each update that access the node) when we are completely done with the rebuilding. After $O((\sqrt{B})^l)$ I/Os we will be finished rebuilding our shadow structures, and finished performing all the updates that arrived during the rebuilding. We can also make a copy (a shadow) of the rebuild shadow structure in the same I/O bound. This finishes the phase and then we in $O(1)$ I/Os switch from the old to the new rebuild structure (and its shadow) as described previously.

4.2 Removing amortization from the corner structure

In this section we will show how to make the update bound on the "corner structure" in Lemma 3 [79] worst case. During this we will also remove the assumption about the internal memory being capable of holding $O(B)$ blocks.

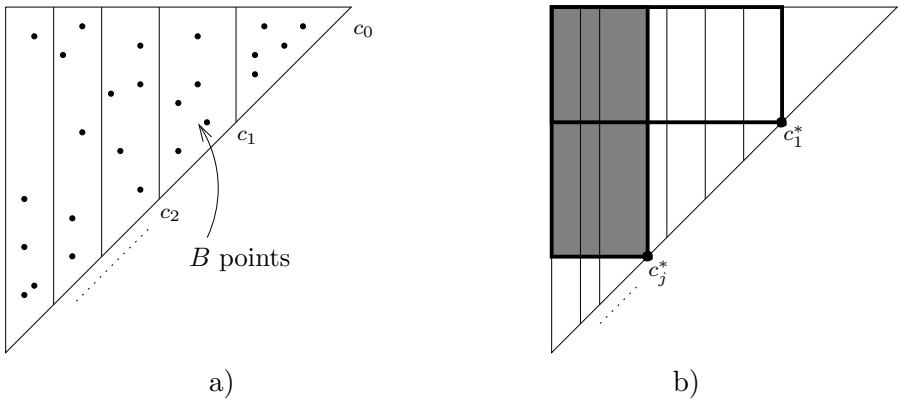


Figure 7: a) Vertical blocking. b) The set C^* (the marked points). The dark lines represent the boundaries of queries whose corners are points in C^*. One such query is shaded to demonstrate how they look like.

Recall from the introduction that by regarding an interval $[x, y]$ as the point (x, y) in the plane, stabbing queries reduce to diagonal corner queries. This property is used in the "corner structure" [79] which actually is a data structure that stores points in the plane above the $x = y$ line, such that diagonal corner queries can be answered efficiently. More precisely

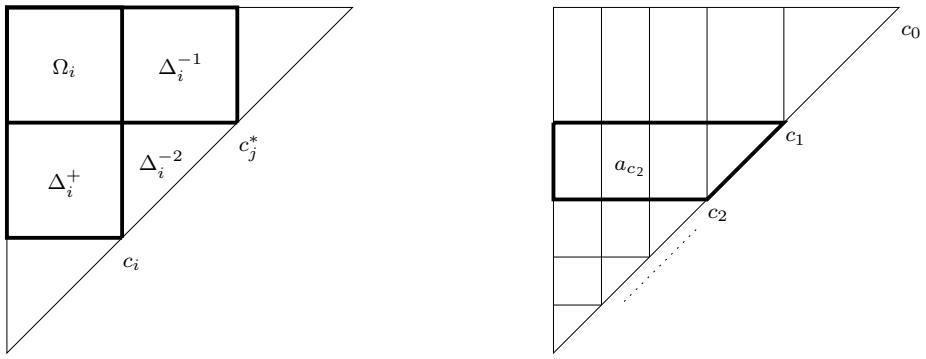


Figure 8: A) The sets Ω_i, Δ_i^{-1} and Δ_i^+. c_j^* is the last point that was added to C^* and c_i is the point being considered for inclusion. B) Definition of a_{c_i}.

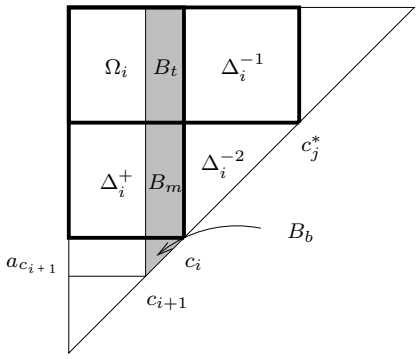
the corner structure is defined as follows: Initially, the set S of $k \leq B^2$ points (segments) is divided into k/B *vertical* oriented blocks as indicated in Figure 7a. Let C be the set of points at which the right boundaries of the regions corresponding to these blocks intersect the $y = x$ line. Then a subset C^* of these points is chosen and one or more blocks is used to explicitly represent the answer to each query that happens to have a corner $c \in C^*$. This is illustrated on Figure 7b. In order to decide which elements in C will become member of C^*, an iterative process is used. The first element in C^* is chosen to be at the intersection of the left boundary of the rightmost block in the vertical oriented blocking. This point is called c_1^*. To decide which other elements of C should be elements of C^*, one proceed along the line $x = y$ from the upper right to the lower left, considering each element in C in turn. Let c_j^* be the element of C most recently added to C^*; initially this is c_1^*. When considering $c_i \in C$ the sets Ω_i, Δ_i^{-1}, Δ_i^{-2} and Δ_i^+ are defined to be subsets of S as shown in Figure 8a. Let $\Delta_i^- = \Delta_i^{-1} \cup \Delta_i^{-2}$, and let $S_j^* = \Omega_i \cup \Delta_i^{-1}$ be the answer to a query whose corner is c_j^*. This was the last set of points that was explicitly blocked. Let $S_i = \Omega_i \cup \Delta_i^+$ be the answer to a query whose corner is c_i. c_i is added to C^* if and only if

$$|\Delta_i^-| + |\Delta_i^+| > |S_i|.$$

After having constructed C^* this way, the set of points S_i^* answering a diagonal corner query for each element $c_i^* \in C^*$ is explicitly stored *horizontally* blocked.

It is shown in [79] that after having defined C and C^* in the above way, and blocked the points as discussed, the total number of blocks used to store all the sets S_i^* is $O(k/B)$. This is then also a bound on the total number of blocks used by the structure. Then it is show how a query can be answered in $O(T/B + 1)$ I/Os using the representation, arriving at Lemma 2. We omit the details here and refer to [79]. Note however, that one key property is that the size of C (and thus C^*) is less than B, which means that we can hold the set in one block. This property makes it possible to find the point in C (and C^*) just above or below any query point on the $x = y$ line in $O(1)$ I/Os.

As described in the introduction the assumption about the internal memory being capable of holding $O(B^2)$ elements is in [79] used to make the structure dynamic with $O(1)$ I/O amortized update bounds (Lemma 3). This is done by maintaining a special update block



$$
\begin{aligned}
|\Delta_{i+1}^{-1}| &= |\Delta_i^{-1}| + |B_t| \\
|\Delta_{i+1}^{-2}| &= |\Delta_i^{-2}| + |B_m| + |B_b| \\
|\Omega_{i+1}| &= |\Omega_i| - |B_t| \\
|\Delta_{i+1}^{+}| &= |\Delta_i^{+}| - |B_m| + (a_{i+1} - |B_b|)
\end{aligned}
$$

Figure 9: Computation of C^*

and then rebuild the structure completely once B updates have been inserted in this block. The rebuilding is done in $O(k/B) = O(B)$ I/Os simply by loading the whole structure into internal memory, rebuild it, and write it back to external memory again. We will now show how to remove the assumption on the size of the internal memory and make the update bounds worst case. We do so by designing an algorithm for rebuilding the structure in $O(k/B)$ I/Os, without loading more than $O(1)$ blocks into internal memory at any time. It then turns out that this algorithm can be done incrementally over $O(k/B)$ updates, which means that we can use a variant of the global rebuilding technique to remove the amortization: We start a rebuilding of the structure once the number of updates in the update block reaches $B/2$ and do $O(1)$ I/Os on the rebuilding on each of the succeeding updates. We make sure that we do the rebuilding fast enough finish the rebuilding after $B/2$ updates, which means that we will be finished, an can switch structures, before we need to begin the next rebuilding.

In order to design our new (incremental) rebuilding algorithm we need to change the representation of the corner structure slightly. In addition to storing the points vertical blocked and storing the answers to a query for each element in C^*, we also store the point in two lists sorted according to x and y coordinate, respectively. This does not change the asymptotic space bound. Our rebuilding algorithm now consists of three main steps. First we compute and construct the new vertical blocking, that is, the set C. We then compute the set C^*, and finally we construct the blocking needed for each of the points in C^*.

In order to compute the new vertical blocking we simply merge the list of points sorted according to x coordinate with the (half) block of updates. This is easily done in $O(k/B)$ I/Os using $O(1)$ blocks of internal memory. Then we can easily compute C by scanning through the resulting sorted list, sampling the first point in each block. The vertical blocking can be constructed simply by making a copy of the list.

To compute C^* we start by merging the list of points sorted according to y coordinate with the points in the update block. Then we use the resulting list to compute for each pair of consecutive points c_i and c_{i+1} in the sorted sequence of points in C the number of points in S with y coordinate between the y coordinates of c_i and c_{i+1}. For the points c_i and c_{i+1} we call this number a_{c_i} as indicated in Figure 8b. We then compute C^* as previously by proceeding along the $x = y$ line from the top, and for each point $c_i \in C$ decide if it should be a member of C^* by checking if the equation $|\Delta_i^-| + |\Delta_i^+| > |S_i|$ is fulfilled. As the number of points in C is $\lceil k/B \rceil$ we can do this computation in $O(k/B)$ I/Os if we in $O(1)$ I/Os can decide to include a given point or not. We show how to do this by showing how we given $|\Omega_i|$, $|\Delta_i^+|$, $|\Delta_i^{-1}|$ and $|\Delta_i^{-2}|$ can compute $|\Omega_{i+1}|$, $|\Delta_{i+1}^+|$, $|\Delta_{i+1}^{-1}|$ and $|\Delta_{i+1}^{-2}|$ in $O(1)$ I/Os. Consider Figure 9

and assume that we know $|\Omega_i|$, $|\Delta_i^+|$, $|\Delta_i^{-1}|$ and $|\Delta_{i+1}^{-2}|$, the points c_j^*, c_i and c_{i+1}, and the value of a_{i+1} (we can load c_j^*, c_i, c_{i+1} and a_{i+1} in $O(1)$ I/Os, and the same is true for the sizes of the sets if we stored them after the computation of them). We now compute the sets B_t, B_m and B_b using one I/O, by loading the vertical block corresponding to c_i and distribute the points contained in this block in the three sets—using the y coordinate of c_i and c_j^*. $|\Omega_{i+1}|$, $|\Delta_{i+1}^+|$, $|\Delta_{i+1}^{-1}|$ and $|\Delta_{i+1}^{-2}|$ can now be computed without using further I/Os by the formulas given on Figure 9. If is easy to realize that we with minor modifications can use the same formulas if $c_i = c_j^*$, that is, if we just added c_i to C^*.

Finally, we for each point $c_j^* \in C^*$ need to horizontally block the set S_j^* of points in the answer to a query at c_j^*. We again handle the points (this time the ones in S^*) from the top one along the $x = y$ line, and create the horizontal blockings using the list of points sorted according to y coordinate. In order to produce the blocking of S_{j+1}^*, assuming that we have already produced the blocking of S_j^* and that we know the position of the last point in S_j^* (the one with the smallest y coordinate) in the list of all the points sorted according to y coordinate, we do the following (refer for example to Figure 7b): First we scan through the horizontal blocking of S_j^* and collect the points with x coordinate less than the x coordinate of c_j^*. These points (if any) will be the first points in the blocking of S_{j+1}^* and will automatically be horizontally blocked. Next we continue the scan in the list of all points sorted according to y coordinate from the position of the last point in S_j^* (using the pointer), and collect the rest of the points in C_{j+1}^*. Because we process the involved points in y order we will end up with the desired horizontal blocking. To produce S_{j+1}^* we use $O(\lceil |S_j^*|/B \rceil + \lceil |S_{j+1}^*|/B \rceil)$ I/O's, which means that the total number of I/Os used is $O(2\sum_{j \in 1..|S^*|} \lceil |S_j^*/B| \rceil)$. As previously mentioned it is proved in [79] that this number is $O(k/B)$.

This completes the description of the rebuilding algorithm. To realize that the algorithm can be used in an incremental way, that is, that it can be performed in $O(k/B)$ pieces of $O(1)$ I/Os each, just note that throughout the algorithm the current state of the algorithm can be represented by a constant number of pointers and values. Thus we can perform one "step" of the algorithm by loading the current state into internal memory using $O(1)$ I/Os, perform the step using $O(1)$ I/Os again, and finally use $O(1)$ I/Os to write the new state back to external memory. The following lemma then follows from the discussion in the beginning of this section.

Lemma 11 *A set of $k \leq B^2$ segments can be represented using $O(k/B)$ blocks of external memory such that a stabbing query can be answered in $O(T/B + 1)$ I/O operations and such that updates can be performed in $O(1)$ I/Os. The structure can be constructed in $O(k/B)$ I/Os. All bounds are worst case.*

This ends our description of how we remove the amortization from our external interval tree structure and proves Theorem 4.

5 External-memory segment tree

The ideas behind our external-memory interval tree can also be used to develop an external-memory segment tree-like structure with optimal update and query I/O bounds and a better space bound than all previous known such structures [28, 110]. In this section we sketch this structure.

In internal memory a segment tree consists of a binary base tree over the endpoints of segments stored in the tree, and a given segment is stored in the secondary structure of up to two nodes on each level of this tree. More precisely a segment is stored in all nodes v where the segment contains the interval X_v associated with v, but not the interval associated with $parent(v)$. Like our external interval tree our external segment tree has a weight-balanced B-tree (Theorem 2) as base tree, and the segments are stored in $O(B)$ secondary structures of each internal node. Like in the interval tree case an internal node in the base tree defines $\Theta(\sqrt{B})$ slabs, which again defines $\Theta(B)$ multislabs. By analogy with the internal segment tree, a segment s is stored in the secondary structures of a node v of the external segment tree if it spans one of the *slabs* associated with v, but not one of the slabs associated with $parent(v)$ (or equivalently, not the whole interval associated with v). Note the difference from the external interval tree, where we only store s in the highest node where s contains a slab *boundary*. As in the interval tree case we store s in a multislab list corresponding to the largest multislab it spans, but we do not store the "ends" of s in a slab list as we did previously. Instead the ends will (because of the way we defined where to store s) be stored "recursively" further down the tree. As an example consider segment s on Figure 3. This segment will be stored in v because it spans one of v's slabs, but not the whole interval of v. Like in the external interval tree case it will be stored in the multislab consisting of X_{v_2} and X_{v_3}, but the parts of the segment in X_{v_1} and X_{v_4} will not be stored in two slab lists, but will be passed down to v_1 and v_4 and stored recursively further down the tree.

Like previously we store segments from multislab lists containing $o(B)$ segments in an underflow structure. Furthermore, unlike in the interval tree case, we do not keep the multislab lists sorted, that is, we do not implement the lists using B-trees. Instead we, again in analogy with the internal case, maintain pointers between different copies of the same segments. More precisely a segment in node v has pointers to "itself" in the first nodes above and below v containing the segment. In order to be able to rebuild an underflow structure containing k segments in $O(k/B)$ I/Os, we furthermore keep a separate list of all segments stored in the underflow structure together with it. The segments in this list, and not the copy of the segments in the actual underflow structure, are the ones containing the pointers to other copies of the segments. This allows us to move all k segments in the underflow structure during a rebuilding without having to update k pointers using $O(k)$ I/Os. In this way the I/O-use of a rebuilding is kept at $O(k/B)$. In analogy with the internal segment tree it is easy to realize that a segment will at most be stored in two multislab lists on each level of the base tree, which means that we use $O((N/B)\log_B N)$ blocks to store N segments.

A *query* with x can be done in the optimal $O(\log_B N + T/B)$ I/Os, simply by searching down the tree for the leaf containing x, and in each node report the segments in the relevant multislab lists and query the underflow structure.

To *insert* a new segment s in our external segment tree we first insert the endpoints of s in the base tree, and then we do the actual insertion of s by searching down the base tree for the two endpoints of s, inserting s in the relevant multislab lists. As our multislab lists are unsorted, we can insert s in a multislab list in $O(1)$ I/Os, just by appending it to the list (insert it in the last block). It is also easy to maintain the pointers to the other copies of s. We handle the cases involving the underflow structure like in the external interval tree, except for the new list of all segments in the underflow structure mentioned earlier, which is manipulated as was it a multislab list. The actual insertion of s can therefore be done in $O(\log_B N)$ I/Os, but in order to obtain the same total (amortized) update I/O bound, we need to consider how to split a node on level l in $O((\sqrt{B})^l B)$ I/Os. To do so consider

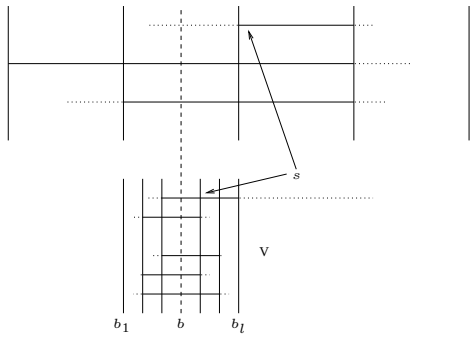


Figure 10: Splitting a segment tree node.

Figure 10. Segments which need to be moved when v splits are all stored in v (and perhaps also in *parent*(v)) and contain the split boundary b. They fall into two categories: segments that contain b_1 or b_l and segments that do not. The last set of segments only need to be moved within v. They need to be "cut in two" at b, that is, a given one of them needs to be deleted from one multislab list and inserted into two new ones. Removing a segment s from a multislab list in $O(1)$ I/Os is easily done by loading the block containing s and the last block in the list, delete s, and insert the last segment from the last block in the place formerly occupied by s. Also inserting s in the two new multislab lists and the update of the relevant pointers can easily be done in $O(1)$ I/Os. The first set of segments need to be moved inside v, as well as be inserted in the secondary structures of *parent*(v). Consider e.g segment s in Figure 10. s needs to be removed from the multislab list corresponding to b_l and the boundary just to the left of b, and be inserted in the multislab list corresponding to boundary b and the one to the left of it. This is done as before in $O(1)$ I/Os. The part of s between boundary b and b_l needs to be moved to *parent*(v), but in order to figure out which multislab list to insert it in we need to check if s is already stored somewhere in *parent*(v)—as it is in the example. If this is the case we need to "glue" the two pieces together to a bigger piece. The check is easily performed just by following the pointer in the copy of s stored in v to the copies of s in the node above v. Note that if that copy is in *parent*(v) we also get its position. Then it is (if needed) easy to remove s from the relevant multislab list in *parent*(v) and insert it in the new relevant one in $O(1)$ I/Os. Altogether we as desired only spend $O(1)$ I/Os on each of the $O((\sqrt{B})^l B)$ segments stored in v.

To delete a segment s we, as in the external interval tree case, use global rebuilding to remove the endpoints from the base tree structure. We use the pointer between segments to do the actual deletion of the segment from the relevant lists. In order to find the first (top) occurrence of a segment we maintain a separate B-tree with all the segments stored in the segment tree sorted according to right endpoint. In this tree a segment has a pointer to itself in the topmost node in the segment tree that stores the segment. This allows us to remove s by locating s in the B-tree, remove it, and then follow pointers to all the $O(\log_B N)$ occurrences of s in the external segment tree.

Finally, it is relatively easy to realize that we, as in the interval tree case, can do the splitting of a node lazily over $O((\sqrt{B})^l B)$ updates that pass the node, arriving at the following theorem:

Theorem 5 *There exists an external-memory segment tree that uses $O((N/B)\log_B N)$ blocks of external memory to store N segments such that stabbing queries can be answered in $O(\log_B N + T/B)$ I/Os and such that updates can be performed in $O(\log_B N)$ I/Os. All bounds are worst case.*

6 Conclusions and open problems

In this paper we have developed an I/O-optimal and disk space-optimal external memory data structure for the dynamic interval management problem, thus settling an important open problem in databases and I/O algorithms. Our data structure can be regarded as an external-memory version of the interval tree. We have also used our ideas to get an improved external version of the segment tree. Our data structures are the first external data structures with worst-case (rather than merely amortized) optimal update I/O bounds for a two- or higher-dimensional range searching problem. Our data structures work without need for the internal memory to hold $O(B^2)$ items. We have developed what can be regarded as a weight-balanced version of the well-known B-tree. This structure is of independent interest, since it can be used to remove amortization from external as well as internal data structures.

Several challenging problems remain open in the area of external range searching, such as, for example, how to develop optimal data structures for other variants of 2-dimensional range searching, as well as for range searching problems in higher dimensions.

Chapter 9

The I/O-Complexity of Ordered Binary-Decision Diagram Manipulation

The I/O-Complexity of Ordered Binary-Decision Diagram Manipulation*

Lars Arge†

BRICS‡
Department of Computer Science
University of Aarhus
Aarhus, Denmark

August 1996

Abstract

Ordered Binary-Decision Diagrams (OBDD) are the state-of-the-art data structure for boolean function manipulation and there exist several software packages for OBDD manipulation. OBDDs have been successfully used to solve problems in e.g. digital-systems design, verification and testing, in mathematical logic, concurrent system design and in artificial intelligence. The OBDDs used in many of these applications quickly get larger than the avaliable main memory and it becomes essential to consider the problem of minimizing the Input/Output (I/O) communication. In this paper we analyze why existing OBDD manipulation algorithms perform poorly in an I/O environment and develop new I/O-efficient algorithms.

1 Introduction

Many problems in digital-systems design, verification and testing, mathematical logic, concurrent system design and artificial intelligence can be expressed and solved in terms of boolean functions [33]. The efficiency of such solutions depends on the data structures used to represent the boolean functions, and on the algorithms used to manipulate these data structures. Ordered Binary-Decision Diagrams (OBDDs) [32, 33] are the state-of-the-art data structure for boolean function manipulation and they have been successfully used to solve problems from all of the above mentioned areas. There exist implementations of OBDD software packages for a number of sequential and parallel machines [17, 31, 101, 102]. Even though there exist very different sized OBDD representations of the same boolean function, OBDDs in real applications tend to be very large. In [17] for example, OBDDs of Gigabyte size are

*An extended abstract version of this paper was presented at the Sixth International Symposium on Algorithms and Computation (ISAAC'95).

†This work was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 7141 (project ALCOM II). Part of the work was done while a Visiting Scholar at Duke University.
Email: large@daimi.aau.dk.

‡Acronym for Basic Research in Computer Science, a Center of the Danish National Research Foundation.

manipulated in order to verify logic circuit designs, and researchers in this area would like to be able to manipulate OBDDs orders of magnitude larger. In such cases the Input/Output (I/O) communication becomes the bottleneck in the computation.

Until recently most research, both theoretical and practical, has concentrated on finding small OBDD representations of boolean functions appearing in specific problems [31, 33, 88, 112], or on finding alternative succinct representations while maintaining the efficient manipulation algorithms [64]. The limit on the size of the problem instances one has been able to solve in practice has generally been determined by the ability to find representations that fit in internal memory of the machine used to solve the problem. The underlying argument for concentrating on the problem of limiting the size of the OBDDs then seems to have been that as soon as they get to large—larger than the available main memory—generating a large number of page faults, resulting in dramatically increasing running times, is inevitable. Very recently however, researchers have instead begun to consider I/O issues arising when the OBDDs get larger than the available internal memory, and experimental results show that very large runtime speedups can be achieved with algorithms that try to minimize the access to external memory as much as possible [17, 102]. These speedups can be achieved because of the extremely large access time of external storage medias, such as disks, compared to the access time of internal memory. In the coming years we will be able to solve bigger and bigger problems due to the development of machines with larger and faster internal memory and due to increasing CPU speed. This will however just increase the significance of the I/O bottleneck since the development of disk technology lacks behind developments in CPU technology. At present, technological advances are increasing CPU speed at an annual rate of 40-60% while disk transfer rates are only increasing by 7-10% annually [113].

In this paper we analyze why existing OBDD manipulation algorithms perform poorly in an I/O environment and develop new I/O-efficient algorithms.

1.1 I/O Model and Previous Results

We will be working in the *parallel disk model* [5, 133] which models the I/O system of many existing workstations. The model has the following parameters:

$$\begin{aligned} N &= \#\text{ of elements in the problem instance;} \\ M &= \#\text{ of elements that can fit into main memory;} \\ B &= \#\text{ of elements per disk block;} \\ D &= \#\text{ of parallel disks,} \end{aligned}$$

where $M < N$ and $1 \leq DB \leq M/2$. Depending on the size of the data elements, typical values for workstations and file servers in production today are on the order of $M = 10^6$ or 10^7 and $B = 10^3$. Values of D range up to 10^2 in current disk arrays.

An I/O operation (or I/O) in the model is the process of simultaneously reading or writing D blocks of data, one block of B contiguous elements to or from each of the D disks. The I/O-complexity of an algorithm is simply the number of I/Os it performs. Internal computation is free, and we always assume that the N elements initially are stored in the first N/DB blocks of each disk. Thus reading all the input data requires N/DB I/Os. We will use the term *scanning* to describe the fundamental primitive of reading (or writing) all items in a set stored contiguously in external memory by reading (or writing) the blocks of the set in a sequential manner.

Early work on external-memory algorithms concentrated on sorting and permutation related problems [5, 47, 99, 100, 131, 133]. More recently researchers have designed external-memory algorithms for a number of problems in different areas. Most notably I/O-efficient algorithms have been developed for a large number of computational geometry [15, 67] and graph problems [42]. Other related papers are [122] and [58] that address the problem of computing the transitive closure of a graph under some restrictions on the size of the graph, and propose a framework for studying memory management problems for maintaining connectivity information and paths on graphs, respectively. Also worth noticing in this context is [97] that addresses the problem of storing graphs in a paging environment, but not the problem of performing computation on them, and [11] where a number of external (batched) dynamic data structures are developed. Finally, it is demonstrated in [41, 129] that the results obtained in the mentioned papers are not only of theoretical but also of great practical interest.

While N/DB is the number of I/Os needed to read all the input,[1] Aggarwal and Vitter [5] proved that $\Theta(\frac{N}{DB}\log_{M/B}\frac{N}{B}) = \Theta(\mathrm{sort}(N))$[2] is the number of I/Os needed to sort N elements. Furthermore, they proved that the number of I/Os needed to rearrange N elements according to a given permutation is $\Theta(\min\{N/D, \mathrm{sort}(N)\}) = \Theta(\mathrm{perm}(N))$.[2] They also developed algorithms with I/O bounds matching these lower bounds in the $D = 1$ model. Later the results have been extended to the general model [99, 98, 131, 133]. In [42] it was shown that the permutation lower bound also applies to a large number of fundamental graph problems.

Taking a closer look at the fundamental bounds for typical values of B and M reveals that because of the large base of the logarithm, $\log_{M/B}\frac{N}{B}$ is less than 3 or 4 for all realistic values of N, M and B. This means that the sorting bound in all realistic cases will be smaller than N/D, such that $\mathrm{perm}(N) = \mathrm{sort}(N)$. In practice the term in the bounds that really makes the difference is the DB-term in the denominator of the sorting bound. As typical values of DB are measured in thousands, going from a $\Omega(N)$ bound—as we shall see is the worst-case I/O performance of many internal-memory algorithms—to the sorting bound, can be really significant in practice.

1.2 OBDDs and Previous Results

An OBDD is a *branching program* with some extra constraints. A branching program is a directed acyclic graph with one root, whose leaves (sinks) are labeled with boolean constants. The non leaves are labeled with boolean variables and have two outgoing edges labeled 0 and 1, respectively. If a vertex is labeled with the variable x_i we say that it has index i. If f is the Boolean function represented by the branching program, an evaluation of $f(a_1, \ldots, a_n)$ starts at the root and follows for a vertex labeled x_i the outgoing edge with label a_i. The label of the sink reached in this way equals $f(a_1, \ldots, a_n)$. An OBDD is a branching program for which an ordering of the variables in the vertices is fixed. For simplicity we assume that this ordering is the natural one, $x_1, \ldots, x_n$, that is, if a vertex with label x_j is a successor of a vertex with label x_i, the condition $j > i$ has to be fulfilled. Figure 1 shows two examples of OBDDs. Note that an OBDD representing a boolean function of n variables can be of size 2^n, and that different variable orderings lead to representations of different size. There exist several algorithms (using heuristics) for choosing a variable ordering that minimizes the OBDD representation of a given function [88, 112].

[1] We refer to N/DB as the *linear* number of I/Os.

[2] For simplicity we write $\mathrm{sort}(N)$ and $\mathrm{perm}(N)$, suppressing M, B and D.

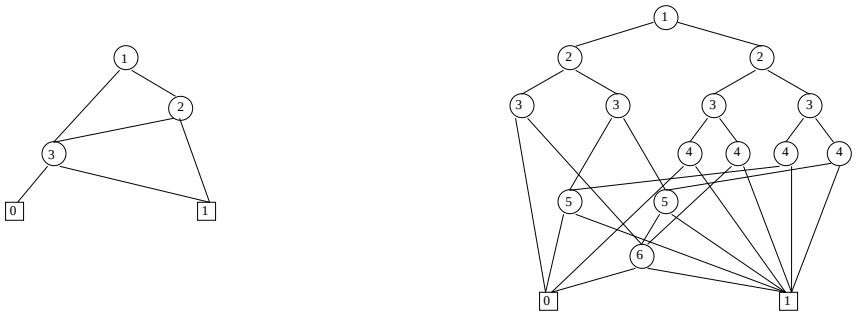


Figure 1: OBDD representations of the functions $x_1 \wedge x_2 \vee x_3$ and $x_1 \wedge x_4 \vee x_2 \wedge x_5 \vee x_3 \wedge x_6$. The left children are 0-successors and the right 1-successors. All edges are directed downwards.

In [32] Bryant proved that for a given variable ordering and a given boolean function there is (up to isomorphism) exactly one OBDD—called the reduced OBDD—of minimal size. Bryant also proved that iterated use of the following two reduction rules on an OBDD with at most one 0-sink and one 1-sink yields the reduced OBDD: 1) If the two outgoing edges of vertex v lead to the same vertex w, then eliminate vertex v by letting all edges leading to v lead directly to w. 2) If two vertices v and w labeled with the same variable have the same 1-successor and the same 0-successor, then merge v and w into one vertex. The OBDDs in Figure 1 are both reduced. Bryant [32] gave an algorithm for reducing an OBDD G with $|G|$ vertices in $O(|G| \log |G|)$ time. Later algorithms running in $O(|G|)$ time have been developed [33, 118].

The most fundamental operations on OBDDs are the following:

- Given an OBDD representing f, compute if f can be satisfied.
- Given two OBDDs representing f_1 and f_2, compute if $f_1 = f_2$.
- Compute from OBDDs representing f_1 and f_2 an OBDD for $f = f_1 \otimes f_2$, where $\otimes$ is some boolean operator.

The first two operations can easily be performed on *reduced* OBDDs. From a computational point of view the fundamental operations are therefore the reduce operation and the apply operation, as we shall call the operation which computes the reduced OBDD for the function obtained by combining two other functions by a binary operator. In [32] an $O(|G_1| \cdot |G_2|)$ time algorithm for using the apply operation on two OBDDs of size $|G_1|$ and $|G_2|$ is developed. This algorithm relies on a depth first traversal algorithm. In [101] a breadth first traversal algorithm with the same time bound is given.

Even though the I/O system (the size of the internal memory) seems to be the primary limitation on the size of the OBDD problems one is able to solve practically today [17, 31, 32, 33, 102], it was only very recently that OBDD manipulation algorithms especially designed to minimize I/O were developed. In [102] Ochi, Yasuoka and Yajima realized that the traditional depth first and breadth first apply algorithms do not perform well when the OBDDs are too large to fit in internal memory, and they developed alternative

algorithms working in a levelwise manner.[3] These algorithms were obtained by adding extra vertices to the representation (changing the OBDD definition) such that the index of successive vertices on any path in an OBDD differs by exactly one. This makes the previously developed breadth first algorithm [101] work in a levelwise manner. Implementing the algorithms they obtained runtime speedups of several hundreds compared to an implementation using depth first algorithms. Very recently Ashar and Cheong [17] showed how to develop levelwise algorithms without introducing extra vertices, and conducted experiments which showed that on large OBDDs their algorithms outperform all other known algorithms. As the general idea (levelwise algorithms) are the same in [101] and [17], we will only consider the latter paper here. Finally, it should be mentioned that Klarlund and Rauhe in [80] report significant runtime improvements when working on OBDDs fitting in internal memory and using algorithms taking advantage of the blocked transport of data between cache and main memory.

In the algorithms in [17] no explicit I/O control is used. Instead the algorithms use a virtual memory space and the I/O operations are done implicitly by the operation system. However, explicit memory management is done in the sense that memory is allocated in chunks/blocks that match the size of an I/O block, and a specific index is associated with each such block. Only vertices with this index are then stored in such a block. This effectively means that the OBDDs are stored in what we will call a level blocked manner. The general idea in the manipulation algorithms is then to try to access these level blocked OBDDs in such a way that the vertices are accessed in a pattern that is as levelwise as possible. On the other hand we in this paper assume that we can explicitly manage the I/O. This could seem to be difficult in practice and time consuming in terms of internal computation. However, as Vengroff and Vitter show in [129]—using the transparent parallel I/O environment (TPIE) developed by Vengroff [126]—the overhead required to manage I/O can be made very small.

1.3 Our Results

In this paper we analyze why the "traditional" OBDD manipulation algorithms perform poorly when the OBDDs get large, by considering their I/O performance in the parallel disk model. Furthermore we develop new I/O-efficient algorithms.

First we show that all existing *reduce* algorithms—including the algorithms developed with I/O in mind [17]—in the worst case use $\Omega(|G|)$ I/Os to reduce an OBDD of size $|G|$. We show that this is even the case if we assume that the OBDD is blocked in external memory in some for the algorithm "natural" or favorable way by the start of the algorithm—depth first, breadth first or level blocked.[4] Then we show that for a special class of algorithms, which includes all existing algorithms, $\Omega(\text{perm}(|G|))$ is a lower bound on the number of I/Os needed to reduce an OBDD of size $|G|$. We show that this is even the case if we assume one of the blockings mentioned above, and even if we assume another intuitively good blocking. Previous I/O lower bounds on graph problems all assume general blockings. Finally, we develop an $O(\text{sort}(|G|)$ I/O reduce algorithm. Thus our algorithm is asymptotically optimal in all realistic I/O-systems, among algorithms from the special class we consider.

[3] When we refer to depth first, breadth first and levelwise algorithms we refer to the way the apply algorithm traverse the OBDDs. By levelwise algorithm we mean an algorithm which processes vertices with the same index together. All known reduce algorithms work in a levelwise manner.

[4] When we refer to a blocking as e.g. a depth first blocking, we refer to a blocking where the vertices are assigned to blocks in the way they are met in a depth first traversal.

We then go on and analyze the existing *apply* algorithms in the parallel disk model. Again we show that in the worst case all existing algorithms use $\Omega(|G_1| \cdot |G_2|)$ I/Os, and that this also holds for natural blockings of the involved OBDDs. We also develop an $O(\text{sort}(|R|))$ I/O apply algorithm. Here $|R|$ denotes the size of the resulting un-reduced OBDD. Our algorithm is thus asymptotically optimal in all realistic I/O-systems assuming that we have to do a reduction step after the use of the apply algorithm.

We believe that the developed algorithms could be of enormous practical value, as the constants in the asymptotic I/O bounds are all small. As mentioned in [17] large runtime improvements open up the possibility of creating OBDDs for verifying very large portions of chips, something considered impossible until now.

The rest of the paper is organized with a section for each of the OBDD manipulation algorithms. For simplicity we only consider the one disk model in these two sections. In Section 4 we then discuss extension of our results to the general D-disk model. We end the paper with a concluding section.

2 The Reduce Operation

Our discussion of the reduce operation is divided into three main parts. In Subsection 2.1 we present the existing reduce algorithms in order to be able to analyze their I/O-behavior in Subsection 2.2. For natural reason these two subsection will be rather discussing, and not very mathematically strict as Subsection 2.3 where we prove a lower bound on the number of I/Os needed to reduce a given OBDD. Finally, we in Subsection 2.4 present our new I/O-efficient reduce algorithm.

In our discussions of reduce algorithms—and in the rest of this paper—we assume that an OBDD is stored as a number of vertices and that the edges are stored implicitly in these. We also assume that each vertex knows the indices (levels) of its two children. The same assumptions are made in the existing algorithms. This means that the fundamental unit is a vertex (e.g., an integer—we call it the *id* of the vertex) with an index, and an id and an index (and maybe a pointer) for each of the two children. The vertices also contain a few other fields used by the apply and reduce algorithms.

2.1 Reduce Algorithms

All reduce algorithms reported in the literature basically works in the same way. In order to analyze their I/O behavior, we in this section sketch the basic algorithm and the different variations of it.

The basic reduce algorithm closely follows an algorithm for testing whether two trees are isomorphic [6]. The algorithm processes the vertices levelwise from the sinks up to the root, and tries to use the two reduction rules discussed previously on each vertex. When the root is reached the reduced OBDD has been obtained, as the reduction rules cannot be used on any of the vertices in the OBDD. More precisely the algorithm assigns an integer label to each vertex in the OBDD such that a unique label is assigned to each unique sub-OBDD: First, two distinct labels are assigned to the sink vertices—one to the 1-sinks and one to the 0-sinks—and then the vertices are labeled level by level (index by index). Assuming that all vertices with index greater than i have been processed, a vertex v with index i is assigned a label equal to that of some other vertex that has already been relabeled, if and only if one of two conditions is satisfied (one of the two reduction rules can be used). First, if the labels of

v's children are equal the vertex is redundant and it is assigned a label equal to that of the children (R1). Secondly, if there exists some already processed vertex w with index i whose left and right children have the same labels as the left and right children of v, respectively, then the sub-OBDDs rooted in v and w is isomorphic, and v is assigned the same label as w (R2). When all vertices have been relabeled the OBDD consisting of precisely one vertex for each unique label is the reduced OBDD corresponding to the original one.

The reduction algorithm comes in three variants (disregarding I/O issues for now), and the main difference between them is the way they decide if reduction rule R2 can be used on a given vertex. When processing vertices with index i Bryant's original algorithm [32] sorts the vertices according to the labels of their children such that vertices which should have assigned the same label end up next to each other in the sorted sequence. The time complexity of the algorithm is dominated by the time used to sort the vertices, such that the algorithm runs in time $O(|G| \log |G|)$. Later, Bryant [33] gave an algorithm which instead of sorting the vertices maintain a (hash) table with an entry for each unique vertex seen so far. When processing a vertex a lookup is made in the table to see if an isomorphic vertex has already been labeled. If not the vertex is given a new unique label and inserted in the table. The implementations reported in [17, 31, 101, 102] all use this general idea. From a theoretical point of view the table uses a lot of space, namely $O(n \cdot |G|^2)$, but using "lazy initialization" [6] the running time of the algorithm can be kept at $O(|G|)$, as that is the number of entries in the table which is actually used. Finally, the algorithm by Sieling and Wegener [118] also sorts the vertices with a given index according to the labels of the children, but uses the bounded size of the label domain to do so with two phases of the well-known bucket sort algorithm. First, the vertices are partitioned according to the label of the 0-successor, and in a second bucket sort phase the non-empty buckets are partitioned according to the 1-successor. Vertices that end up in the same bucket is then assigned the same label. The algorithm runs in $O(|G|)$ time.

2.2 The I/O behavior of Reduce Algorithms

The basic reduce algorithm by Bryant [32] starts by doing a depth first traversal of the OBDD in order to collect the vertices in lists according to their indices (levels). If one does not assume anything about the way the vertices are blocked—which probably is most realistic, at least if one works in a virtual memory environment and uses pointers to implement the OBDDs—an adversary can force the algorithm to use $\Omega(|G|)$ I/Os just to do this traversal: If we call the vertex visited as number i in a depth first traversal for v_i, the adversary simply groups vertex $v_1, v_{M+1}, v_{2M+1}, \ldots, v_{(B-1)M+1}$ together into the first block, $v_2, v_{M+2}, v_{2M+2}, \ldots, v_{(B-1)M+2}$ into the second block, and so on. This results in a page fault every time a new vertex is visited. Even if one assumes that the OBDD is blocked in a breadth first manner, or even in a level manner, it is fairly easy to realize that the depth first traversal algorithm causes $\Omega(|G|)$ page faults in the worst case.

So let us assume that the OBDD is blocked in a depth first manner, such that the traversal can be performed in $O(|G|/B)$ I/Os. At first it seems that the algorithm still uses $\Omega(|G|)$ I/Os, as it during the traversal outputs the vertices to n different lists (one for each index), and as an adversary can force it never to output two consecutive vertices in the depth first order to the same list. However, *in practice* this would not be to bad, as we typically have that $n \ll |G|$ and that n actually is smaller than M/B, which means that we can reserve a block in internal memory for each of the n lists and only do an output when one of these blocks

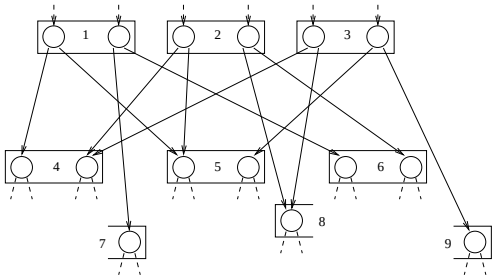


Figure 2: I/O behavior of reduce algorithms ($B = 2$).

runs full. Then we only use $O(|G|/B)$ I/Os to produce the lists. In general an algorithm which scans through the OBDD and distribute the vertices to one of n lists will perform well in practice. As we will discuss below this is precisely the idea used in the algorithm by Ashar and Cheong [17].

So let us then assume that we have produced the index lists (and thus a level blocking of the OBDD) in a acceptable number of I/Os, and analyze how the different variations of the basic algorithm then perform. Recall that all the variations basically sort the vertices with a given index according to the labels of their children. This means that when processing vertices with a given index the children have to be "visited" in order to obtain their labels. Assuming noting about the order in which this is done, it is not difficult to realize that an algorithm can be forced to do an I/O every time it visits a child. Actually, this holds whatever blocking one has—depth first, breadth first or level blocking—mainly because the vertices can have large fan-in. As an example of this, consider the part of an OBDD in Figure 2. We assume that the OBDD is level blocked and that $B = 2$ and $M/B = 3$, that is, that the main memory can hold 3 blocks. Now consider the process of visiting the children of each vertex in block 1 through 3, assuming that a least recently used (LRU) like paging strategy is used. First we load block 1 and start to visit the children of the leftmost vertex. To do so we load block 4 and then block 5. Then we continue to visit the children of the second vertex in block 1. To do so we have to make room for block 7, so we flush block 4 from internal memory. Then we can load block 7 and continue to load block 6, flushing block 5. This process continues, and it is easy to realize that we do an I/O every time we visit a vertex. Similar examples can be given for depth first and breadth first blockings.

The above problem is also realized in [17], and in order to avoid some of the randomness in the memory access the algorithm presented there visits the children in level order. This is accomplished by scanning through the vertices, distributing each of them to two of n lists according to the index (level) of their children. As discussed above this can be done I/O-efficient in practice. Then these lists are processed one at a time and the desired labels are obtained. The algorithm follows the general philosophy mentioned earlier that as the vertices are stored levelwise they should also be accessed levelwise. But still it is not hard to realize that also this algorithm could be forced to do an I/O every time a child is accessed, because there is no correlation between the order in which the children on a given level are visited and the blocks they are stored in. For example using the strategy on the OBDD in Figure 2 still results in a page fault every time a child is visited. To summarize, all variations of the basic algorithm use $\Omega(|G|)$ I/O in the worst case to obtain the labels of the children—even the algorithm designed with I/O in mind. Furthermore, there seems to be no simple blocking strategy that avoids this.

Finally, there is the actual sorting step. It is difficult to say how many I/Os the basic algorithm uses on this task, as a number of different sorting algorithms could be used and as some of them might actually perform reasonably in an I/O environment. It is however easy to realize that the (hash) table approaches [17, 33] perform poorly on large OBDDs, as there is no regular pattern in the access to the table. Also the bucket approach used in [118] performs poorly because of the random pattern in which the (large number of) buckets are accessed.

To summarize, all known algorithms use $\Omega(|G|)$ I/Os in the worst case to reduce an OBDD of size $|G|$. As mentioned, this number could be very large compared to the linear or the sorting I/O bounds. There are several reasons why the algorithm in [17] still performs so relatively well in practice. We believe that the main reason is that the OBDDs used in the experiments in [17], even though they are large, still are small enough to allow one level of the OBDD to fit in internal memory. This, together with the intuitively correct levelwise blocking and access to the table, results in the large runtime speedups compared to other algorithms. A main reason is also to be found in the apply algorithm which we discuss in Section 3.

2.3 I/O Lower Bound on the Reduce Operation

After analyzing the I/O performance of existing reduce algorithms, we will now prove a lower bound on the number of I/Os we have to use in order to reduce an OBDD. As mentioned in the introduction, Aggarwal and Vitter [5] proved a lower bound on the number of I/Os needed to permute N elements. They used a counting argument where they counted the number of permutations one can produce with a given number of I/Os and compared this to $N!$ We will use the same kind of technique to prove a lower bound on the reduce operation. However, while the current permutation is well defined throughout an algorithm for the permutation problem, this is generally not the case in graph computations like the reduce operation. In the permutation case one can regard the main and external memory as one big memory, and it is then easy to define the current permutation as the N elements are all present somewhere in this memory throughout the algorithm. On the contrary elements (vertices) may disappear and new ones may be created during a reduce operation. In the extreme case all the vertices of an input OBDD are removed by a reduce operation and replaced by one (sink) vertex. In order to prove permutation-like bounds on graph problems, that is, bounds expressing the fact that in the I/O-model it is in general hard to rearrange elements according to a given permutation, we thus restrict our attention to a specific class of reduce algorithms. Intuitively, the class consists of all algorithms that work by assigning labels to vertices, and check if the reduction rules can be used on a vertex by checking the labels of its children. The assumption is that the children of a vertex are loaded into internal memory (if they are not there already) when their new label is checked. All known reduction algorithms belong to this class and the one we develop in Section 2.4 does as well. In order to define the class precisely, we in Section 2.3.1 define a pebble game played on a graph. We also discuss its relation to I/O-complexity. Then we in Section 2.3.2 consider a specific graph and prove a lower bound for playing the game on this graph. This result is then used to prove an I/O lower bound on the reduce operation. We prove the lower bound for a number of specific blockings.

2.3.1 The (M, B)-Blocked Red-Blue Pebble Game

In [72] Hung and Kung defined a red-blue pebble game played on directed acyclic graphs in order to define I/O-complexity. In their game there were no notion of blocks. Here we define a game which is also played on directed graphs with red and blue pebbles, but otherwise is rather different form the Hung and Kung game. Among other things our game takes blocks into account.

An mentioned our (M, B)-blocked red-blue pebble game is played on a graph. During the game the vertices of the graph hold a number of pebbles colored red or blue. The blue pebbles contain an integer each, called the *block index*. Also the edges of the graph will be colored blue or red. A *configuration* is a pebbled graph with colored edges. In the *start configuration* all vertices contain precisely one blue pebble and all edges are blue. Furthermore, precisely B pebbles have the block index 1, precisely B have block index 2, and so on up to V/B (we assume without loss of generality that B divides the number of vertices V). Throughout the game at most $M - B$ pebbles may be red. A *terminal configuration* is one where all pebbles are blue and all edges red. The rules of the game are the following:

Rule 1: (Input) Blue pebbles with the same block index may be colored red.

Rule 2: (Output) Up to B red pebbles may be colored blue and given the same block index, while all other pebbles with that block index are removed from the game.

Rule 3: New red pebbles may be placed on any vertex with a red pebble.

Rule 4: The edge (v_i, v_j) may be colored red if both vertices v_i and v_j contain at least one red pebble.

A *transition* in the game is an ordered pair of configurations, where the second one follows from the first one by using one of the above rules. A *calculation* is a sequence of transitions of which the first configuration is the start configuration. A calculation is *complete* if it ends with the terminal configuration. We define the *pebble I/O-complexity* of a complete calculation to be the number of transitions in the calculation defined by the use of rule one or two.

Playing the pebble game on a graph models an I/O algorithm with the graph as input, and pebble I/O-complexity corresponds to I/O-complexity as defined in the introduction. Blue pebbles reflect vertices stored on disk and red pebbles vertices stored in main memory. In the start configuration the graph is stored in the first V/B blocks on disk. Rule one and two then correspond to an input and output respectively, while rule three allows copying of vertices in internal memory (and thus storing of the same vertex in different blocks on disk). Finally, rule four—together with the definition of terminating configuration—defines the class of algorithms we want to consider, namely algorithms where for every edge (v_i, v_j) in the graph, the algorithm at some point in the computation holds both vertices v_i and v_j in main memory at the same time.

Note that in the pebble game the external memory is divided into what is normally called *tracks*, as we read and write blocks of elements to or from a block of external memory with a unique block index. However, lower bounds proved in the pebble model also hold in the model discussed in the introduction, as an I/O reading or writing a portion of two tracks can be simulated with a constant number of I/Os respecting track boundaries.

2.3.2 Pebble I/O Lower Bound on the Reduce Operation

In [42] the following generalization of the permutation lower bound from [5] is proved:

Lemma 1 *Let A be an algorithm capable of performing $(N!)^{\alpha}N^{c}$ different permutations on an input of size N, where $0 < \alpha \leq 1$ and c are constant. Then at least one of these permutations requires $\Theta(\text{perm}(N))$ I/Os.*

Using this lemma an $\Omega(\text{perm}(N))$ lower bound can be shown on the number of I/Os needed to solve the *proximate neighbors problem* [42]. The proximate neighbors problem is defined as follows: Initially, we have N elements in external memory, each with a key that is a positive integer $k \leq N/2$. Exactly two elements have each possible value of k. The problem is to permute the elements such that, for every k, both elements with k are in the same block. In [42] the proximate neighbors problem is used to prove lower bounds on a number of important graph problems. We define a variant of the proximate neighbors problem called the *split proximate neighbors problem* (SPN). This problem is defined similar to the proximate neighbors problem, except that we require that the keys of the first $N/2$ elements in external memory (and consequently also the last $N/2$ elements) are distinct. Furthermore, we require that the keys of the first $N/2$ elements are sorted. Following the proof of the lower bound on the proximate neighbors problem we can prove the following:

Lemma 2 *Solving SPN requires $\Omega(\text{perm}(N))$ I/Os in the worst case, that is, there exists an instance of SPN requiring $\Omega(\text{perm}(N))$ I/Os.*

Proof: There are $(N/2)!$ distinct split proximate neighbors problems. We define a block permutation to be an assignment of elements to blocks. For each of the distinct problems an algorithm will do some permutation in order to reach a block permutation that solves it. We want to estimate how many distinct problems one block permutation can be solution to. Consider the first block of a given block permutation. This block contains $B/2$ elements from the first part of the split proximate neighbors problem and $B/2$ elements from the last part. The elements have precisely $B/2$ different keys $k_1, k_2, \ldots, k_{B/2}$. Now let $i_1, i_2, \ldots, i_{B/2}$ be the indices of the elements from the last half of the problem, that is, the positions of the elements in the input configuration. The block permutation in hand can only be a solution to problems in which the keys $k_1, k_2, \ldots, k_{B/2}$ are distributed among the elements with indices $i_1, i_2, \ldots, i_{B/2}$ in the start configuration. This can be done in $(B/2)!$ different ways. This holds for all the N/B blocks in the block permutation, and therefore $((B/2)!)^{N/B}$ is an upper bound on the number of distinct problems one block permutation can be a solution to. Thus we have that $\frac{(N/2)!}{((B/2)!)^{N/B}}$ is a lower bound on the number of block permutations an algorithm solving SPN must be able to perform. As $\frac{(N/2)!}{((B/2)!)^{N/B}} = \Omega\left(\frac{(N!)^{1/3}}{((B/2)!)^{N/B}}\right)$, and as we can rearrange the elements within each block of a block permutation in an additional N/B I/Os, the algorithm could produce $(N!)^{1/3}$ permutations. The bound then follows from Lemma 1. □

Using SPN we can now prove a lower bound on the number of pebble I/Os needed to complete a specific pebble game. Lemma 2 tells us that there exists at least one SPN instance X of size N which requires $\Omega(\text{perm}(N))$ I/Os. We can now obtain an algorithm for X from a pebble game by imagining that the elements in X are written on some of the pebbles in a specific graph. Figure 3 shows how we imagine this encoding. The marked vertices are

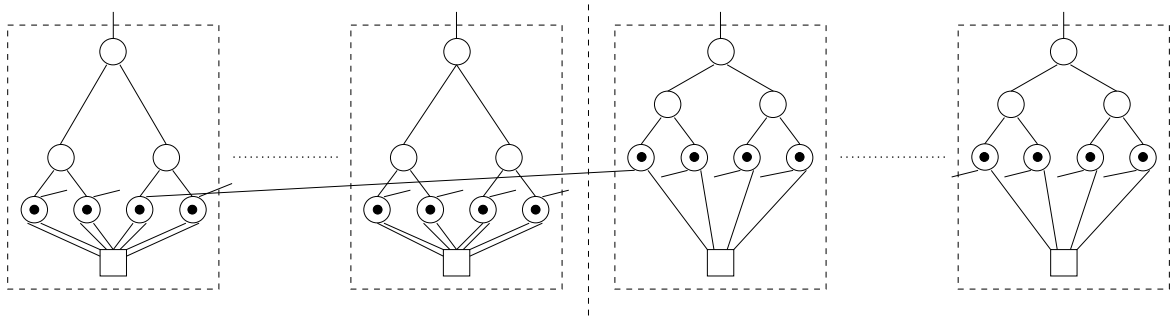


Figure 3: Graph used to obtain SPN algorithm from pebble game ($B = 8$).

the ones containing elements from X, and the vertices to the left of the vertical dotted line contain the first half of the elements. Vertices containing elements with the same key are connected with an edge. In the start configuration the pebbles in the dotted boxes (blocks) have the same block identifier.[5] We can now prove the following:

Lemma 3 *Completing the pebble game on the graph in Figure 3 takes at least* $\Omega(\text{perm}(N))$ *pebble I/Os.*

Proof: Any sequence of transitions S that completes the game can be used to construct an I/O algorithm for the hard SPN instance X (or rather a sequence of I/Os that solve X). The I/O algorithm first scans through X to produce a configuration where the first block contains the first $B/2$ input elements, the next the next $B/2$ elements and so on. This is done in $O(N/B)$ I/Os. Then it simulates the pebble I/Os in S, that is, every time a rule one transition is done in S involving vertices used in the encoding of X, the algorithm performs a similar input, and every time a rule two transition involving vertices used in the encoding is done it performs a similar output. Now every time an edge between vertices used in the encoding is colored red in the game (using a rule four transition), two elements with the same key in X are in main memory. When this happens the I/O algorithm puts the two elements in a special block in main memory. We have such a spare block because the pebble game was designed only to use $M - B$ internal memory. The block is written back to disk by the algorithm when it runs full. Thus when the pebble game is complete the I/O algorithm will have produced a set of blocks on disk solving the SPN problem on instance X. However there is one complication, as the pebble game allows copying of elements. We did not allow copying of elements in the SPN lower bound proof, so the solution to X should consist of the original elements on not of copies. But given the sequence of I/Os solving X produced above, we can easily produce another sequence solving X which do not copy elements at all, simply by removing all elements except for those constituting the final solution. As the number of I/Os performed by the algorithm is bounded by $O(N/B) + O(|S|)$ the lemma follows. □

Having proved a lower bound on the number of pebble I/Os needed to complete a (M, B)-blocked pebble game on the graph in Figure 3, we can now easily prove a lower bound on the number of pebble I/Os needed by a reduce operation. First we build a complete tree on top of the *base blocks* as we shall call the blocks in Figure 3. The tree is blocked as pictured in Figure 4a), and we obtain a breadth first blocked OBDD containing $O(N)$ vertices. In

[5] We assume without loss of generality that 2 and B divide N.

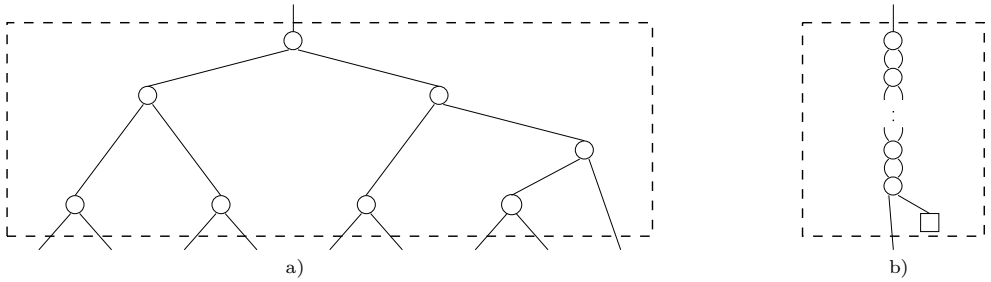


Figure 4: a) One block of top blocking in the breadth first blocking lower bound ($B = 8$). b) One of the "fill blocks" in proof of Lemma 6 and 7.

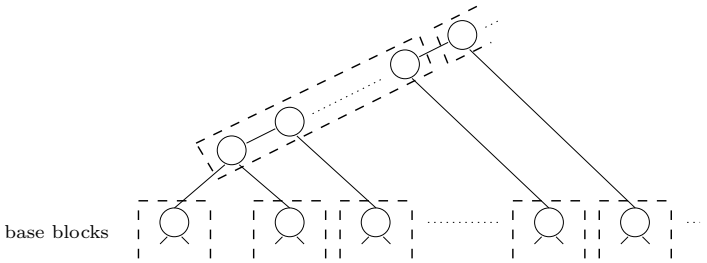


Figure 5: Top-blocking in the depth first blocking lower bound.

a similar way we can obtain a depth first blocked OBDD of size $O(N)$ by building the tree pictured in Figure 5 on top of the base blocks. As Lemma 3 also holds for these extensions of the graph in Figure 3, and as reducing such a graph completes the pebble game on the graph, we get the following.

Lemma 4 *Reducing a breadth first or depth first blocked OBDD with $|G|$ vertices requires $\Omega(\mathrm{perm}\,|G|))$ pebble I/Os in the worst case.*

Recall that the OBDDs in [17] are level blocked. It is easy to repeat the above proof for a level blocked OBDD and thus Lemma 4 also holds for such blockings. But proving these lower bounds does not mean that we could not be lucky and be able to reduce an OBDD in less I/Os, presuming that it is blocked in some other smart way.[6] However, when we later consider the apply operation it turns out that the blockings we have considered (depth, breadth and level) are in fact the natural ones for the different algorithms for this operation. Intuitively however, the best blocking strategy for the class of reduce algorithms we are considering, would be a blocking that minimizes the number of pairs of vertices connected by an edge which are not in the same block—what we will call a *minimal-pair blocking*. But as we will prove next, a slightly modified version of the breadth first blocking we just considered is actually such a minimal-pair blocking. Thus the lower bound also holds for minimal-pair blockings. The modification consists of inserting a layer of the blocks in Figure 4b) between the base blocks and the blocked tree on top of them. The blocks are inserted purely for "proof-technical" reasons, and the effect of them is that every path of length less than B between a marked vertex in the left half and a marked vertex in the right half of the base blocks must contain one of the edges between marked vertices.

[6] We would then of course also have to worry about maintaining such a blocking between operations.

In order to prove that the blocking of the graph G in Figure 3 and 4 is indeed a minimal-pair blocking, we first state the following lemma which follows directly from the fact that every vertex in G has at least in-degree one.

Lemma 5 *For all blockings of G every block must have at least one in edge.*

Now intuitively the blocking in Figure 3 and 4 is a minimal-pair blocking because all blocks, except for the base blocks, have one in edge, and because the vertices in the base blocks cannot be blocked in a better way than they are, that is, they must result in at least half as many edges between vertices in different blocks as there are marked vertices—we call such edges *pair breaking* edges. We will formalize and prove this in the next series of lemmas.

Call the first $N/2$ marked vertices in Figure 3 for a-vertices, and the last $N/2$ marked vertices for b-vertices. These vertices are the "special" vertices, because they have edges to each other and to the multi fan-in sink vertices (we call the sink vertices c-vertices). We now consider a block in an arbitrary blocking of G containing one or more of these special vertices.

Lemma 6 *Let K be a block containing a_1 a-vertices and their corresponding c-vertices and a_2 a-vertices without their corresponding c-vertices, together with b_1 b-vertices with their c-vertices and b_2 b-vertices without their c-vertices, such that a_1, a_2, b_1 and b_2 are all $\leq B/2$, and such that at least one of the a_i's and one of the b_i's are non zero. K has at least $a_1 + a_2 + b_1 + b_2 + k$ pair breaking edges, where k is the number of a_1, a_2, b_1, b_2 that are non zero.*

Proof: First we assume without loss of generality that all a_1 a-vertices are from the same base block. This is the best case as far as pair breaking edges are concerned, because they are all connected to the same c-vertex. Similarly, we assume that all b_1 b-vertices are from the same base block. Finally, we make the same assumption for the a_2 a-vertices and the b_2 b-vertices without their corresponding c-vertices. Making these assumptions we are left with vertices from at most four base blocks. If we can prove the lemma for this particular configuration it also holds for all other configurations.

We divide the proof in cases according to the value of a_1 and b_1:

- $a_1 = 0$

  In this configuration every a-vertex accounts for at least two pair breaking edges—namely the two edges to the c-vertices corresponding to the a_2 a-vertices. But then it is easy to realize that a and b vertices accounts for $a_2 + b_1 + b_2$ pair breaking edges. One of the edges of the a vertices accounts for a_2 of them. For every b-vertex its corresponding a-vertex is either not one of the a_2 a-vertices, in which case the b-vertex itself accounts for a pair-breaking edge, or it is one of the a-vertices, in which case the other of the a-vertices pair breaking edges with a c vertex can be counted. Finally, because of the "fill blocks" above the base blocks (Figure 4), we cannot connect the base blocks except with edges between a and b vertices. This means that every base block must contribute with one pair breaking edge not counting a-b edges. This gives the extra k pair breaking edges.

- $a_1 \geq 1$,$b_1 \geq 1$

  To hold a and b-vertices and the two c-vertices we use $a_1 + a_2 + b_1 + b_2 + 2$ of K's capacity of B vertices. The number of pair breaking edges in a block consisting of *only*

these vertices is $a_1 + 2(B/2 - a_1) + 3a_2 + b_1 + (B/2 - b_1) + 2b_2 + |(a_1 + a_2) - (b_1 + b_2)|$. The first two terms correspond to the a_1 a-vertices and the third term to the a_2 a-vertices—not counting edges between a and b vertices. Similarly, the next three terms correspond to the b-vertices. The last term counts the minimal number of pair breaking edges corresponding to edges between a and b vertices in the block (assuming that as many as possible are "paired").

Now we add vertices one by one in order to obtain the final block K. We add them in such a way that when adding a new vertex, it has an edge to at least one vertex already in the block (if such a vertex exists). Because we cannot connect the base blocks except with a-b edges (again because of the "fill blocks"), one added vertex can at most decrease the number of pair breaking edges by one. Assuming that every added vertex decreases the number of pair breaking edges by one, we end up with a block with $a_1 + 2(B/2 - a_1) + 3a_2 + b_1 + (B/2 - b_1) + 2b_2 + |(a_1 + a_2) - (b_1 + b_2)| - (B - (a_1 + a_2 + b_1 + b_2 + 2)) = 3a_2 + 2b_2 + B/2 - a_1 + |(a_1 + a_2) - (b_1 + b_2)| + a_1 + a_2 + b_1 + b_2 + 2$ pair breaking edges. We want to prove that this number is at least $a_1 + a_2 + b_1 + b_2 + k$, which is indeed the case if $3a_2 + 2b_2 + B/2 - a_1 + |(a_1 + a_2) - (b_1 + b_2)| \geq k - 2$. This again trivially holds because $a_1 \leq B/2$ and because k is only greater than 2 if a_2 and/or b_2 is non zero.

- $a \geq 1$,$b_1 = 0$

  As $b_1 = 0$, b_2 must be non zero. Doing the same calculations as in the previous case we find that the lemma holds if $3a_2 + 2b_2 - a_1 + |a_1 + a_2 - b_2| \geq k - 1$.

  Now if $a_1 + a_2 \geq b_2$ we get that $4a_2 + b_2$ should be grater than or equal to $k - 1$, which is trivially fulfilled as $k = 2$ if $a_2 = 0$ and 3 otherwise.

  If $a_1 + a_2 \leq b_2$ we get that $2a_2 + 3b_2 - 2a_1$ should be greater than or equal to $k - 1$. This is again trivially fulfilled (under the assumption $a_1 + a_2 \leq b_2$).

□

Now we want to remove the $a_1, a_2, b_1, b_2 \leq B/2$ assumption from Lemma 6. In the proof the assumption was used to easily be able to bound the number of pair breaking edges between c-vertices in K and a and b-vertices outside the block. Note that if one of the variables is greater than $B/2$ then the others must be less than $B/2$.

Lemma 7 *Let K be a block containing a_1 a-vertices and their corresponding c-vertices and a_2 a-vertices without their corresponding c-vertices, together with b_1 b-vertices with their c-vertices and b_2 b-vertices without their c-vertices, such that at least one of the a_i's and one of the b_i's are non zero. K has at least $a_1 + a_2 + b_1 + b_2 + k$ pair breaking edges, where k is the number of a_1, a_2, b_1, b_2 that are non zero.*

Proof: As in the proof of Lemma 6 we assume that all (or as many as possible) a and b-vertices of the same type are in the same base block. Taking a closer look at the proof of Lemma 6 quickly reveals that the proof works even if a_2 or b_2 is greater than $B/2$. The proof also works if $a_1 = 0$, so we are left with the following two cases:

- $b_1 > B/2$ (and $1 \leq a < B/2$)

  The block consisting of only a and b-vertices and the corresponding c-vertices have at least $a_1 + 2(B/2 - a_1) + 3a_2 + b_1 + (B/2 - (b_1 - B/2)) + 2b_2 + |(a_1 + a_2) - (b_1 + b_2)| =$

$3a_2 + 2b_2 + 2B - a_1 + |(a_1 + a_2) - (b_1 + b_2)|$ pair-breaking edges. Assuming that the rest $(B - (a_1 + a_2 + b_1 + b_2 + 3))$ of the vertices in the block all brings the number of pair breaking edges down by one, the lemma follows from the fact that the following inequality is trivially satisfied $3a_2 + 2b_2 + B - a_1 + |(a_1 + a_2) - (b_1 + b_2)| \geq k - 3$.

- $a_1 > B/2$

  We divide the proof in two:

  - $b_1 \geq 1$

    The number of pair breaking edges "produced" by a and b-vertices is at least $a_1 + 2(B/2 - (a_1 - B/2)) + 3a_2 + b_1 + (B/2 - b_1) + 2b_2 + |(a_1 + a_2) - (b_1 + b_2)|$ leading to the following inequality $3a_2+2b_2+3/2B-a_1+|(a_1+a_2)-(b_1+b_2)| \geq k-3$. This inequality is trivially satisfied as $a_1 < B$.

  - $b_1 = 0$ (and $b_2 \geq 1$)

    Using the same argument the satisfied inequality is $3a_2 + 2b_2 + B - a_1 + |(a_1 + a_2) - (b_1 + b_2)| \geq k - 3$.

□

We can now prove the main lemma:

Lemma 8 *The breadth first blocking of G in Figure 3 and 4 is a minimal-pair blocking.*

Proof: First note that Lemma 7 also holds (trivially) if a block only contains vertices of one type. It then follows that every a and b-vertex must result in at least one pair breaking edges. Furthermore, every c-vertex (or rather every base block) must result in at least one additional pair breaking edge. Thus the blocking in Figure 3 obtains the minimal number of pair breaking edges. From Lemma 5 we furthermore know that every other block must have at least one in edge. The lemma then follows from the fact that the blocking in Figure 4 only has one in edge. □

Now we have proved that the intuitively best blocking for the reduction algorithm, as well as all the intuitively best blockings for the apply algorithms, all result in a lower bound of $\Omega(\text{perm}(N))$ I/Os on the reduce operation. The results can be summarized as follows.

Theorem 1 *Reducing an OBDD with $|G|$ vertices—minimal-pair, level, depth first or breadth first blocked—requires $\Omega(\text{perm}(|G|))$ pebble I/Os in the worst case.*

2.4 I/O-Efficient Reduce Algorithm

Recall that one of the main problems with existing reduce algorithms with respect to I/O is that when they process a level of the OBDD they perform a lot of I/Os in order to get the labels of the children of vertices on the level. Our solution to this problem is simple—when a vertex is given a label we "inform" all its immediate predecessors about it in a "lazy" way using an external priority queue developed in [11]. On this priority queue we can do a sequence of N insert and deletemin operations in $O(\text{sort}(N))$ I/Os in total. After labeling a vertex we thus insert an element in the queue for all predecessors of the vertex, and we order the queue such that we on higher levels simply can perform deletemin operations to obtain the required labels.

In order to describe our algorithm precisely we need some notation. We refer to a vertex and its label with the same symbol (e.g. v). The index or level of a vertex is referred to as $id(v)$, and the 0-successor and 1-successor are referred to as $low(v)$ and $high(v)$, respectively. In order to make our reduce algorithm I/O-efficient, we start the algorithm by creating two sorted lists of the vertices in the OBDD we want to reduce. The first list (L1) contains the vertices sorted according to index and secondary according to label—that is, according to $(id(v), v)$. Put another way, the list represents a level blocking of the OBDD. The second list (L2) contains two copies of each vertex and it is sorted according to the index of their children and secondarily according to the labels of their children. That is, we have two copies of vertex v ordered according to $(id(low(v)), low(v))$ and $(id(high(v)), high(v))$, respectively. To create L1—for *any* blocking of the OBDD—we just scan through the vertices in the OBDD, inserting them in a priority queue ordered according to index and label, and then we repeatedly perform deletemin operations to obtain L1. Thus we use $O(\text{sort}(N))$ I/Os to produce L1. L2 can be produced in the same number of I/Os in a similar way.

We are now ready to describe our new reduce algorithm. Basically it works like all the other algorithms. We process the vertices from the sinks up to the root and assign a unique label to each unique sub-OBDD root. We start by assigning one label to all 0-sinks and another to all 1-sinks. This is done just by scanning through L1 until all sinks have been processed. During this process—and the rest of the algorithm—we output one copy of each unique vertex to a result list. After labeling the sink vertices we insert an element in the priority queue for each vertex that has a sink as one of its children. The elements contain the label of the relevant child (sink), and the queue is ordered according to level and label of the "receiving" vertex (the vertex having a sink as child). This is accomplished by merging the list of sinks with the appropriate (first) elements in L2. Now assume that we have processed all vertices above level i and want to process level i. In the priority queue we now have one or two elements for each vertex that has a child on a lower level than i. In particular we have two elements for every vertex on level i. Because the elements in the priority queue are ordered according to $(id(v), v)$, we can thus just do deletemin operations on the queue until we have obtained the elements corresponding to vertices on level i. At the same time we merge the elements with L1 in order to "transfer" the labels of the children to the relevant vertices in L1. Then we proceed like Bryant [32]; we sort the vertices according to the labels of the children (with an I/O optimal sorting algorithm [5, 133]), and use the reduction rules to assign new labels. Then we sort the vertices back into their original order, merge the resulting list with L2, and insert the appropriate elements (vertices with a child on level i) in the priority queue—just like after assigning labels to the sinks. When we reach the root we have obtained the reduced OBDD.

In order to analyze the I/O use of the algorithm, note that a linear number of operations in the size of the OBDD is performed on the priority queue. Thus we in total use $O(\text{sort}(|G|))$ I/Os to manipulate the queue. The I/O use of the rest of the algorithm is dominated by the sorting of the elements on each level, that is, by $O(\text{sort}(|G|))$ in total. We then have:

Theorem 2 *An OBDD with $|G|$ vertices can be reduced in $O(\text{sort}(|G|))$ I/Os.*

3 The Apply Operation

We divide our discussion of the apply operation into subsections on existing algorithms, their I/O-behavior, and on our new I/O-efficient algorithm.

3.1 Apply Algorithms

The basic idea in all the apply algorithms reported in the literature is to use the formula

$$f_1 \otimes f_2 = \overline{x_i} \cdot (f_1|_{x_i=0} \otimes f_2|_{x_i=0}) + x_i \cdot (f_1|_{x_i=1} \otimes f_2|_{x_i=1})$$

to design a recursive algorithm. Here $f|_{x_i=b}$ denotes the function obtained from f when the argument x_i is replaced by the boolean constant b. Using this formula Bryant's algorithm [32] works as follows: Consider two functions f_1 and f_2 represented by OBDDs with roots v_1 and v_2. First, suppose both v_1 and v_2 are sinks. Then the resulting OBDD consists of a sink having the boolean value $value(v_1) \otimes value(v_2)$. Otherwise, suppose that at least one of the vertices is not a sink vertex. If $id(v_1) = id(v_2) = i$ the resulting OBDD consists of a root vertex with index i, and with the root vertex in the OBDD obtained by applying the apply operation on $low(v_1)$ and $low(v_2)$ as 0-child and with the root vertex in the OBDD obtained by applying the apply operation on $high(v_1)$ and $high(v_2)$ as 1-child. Thus a vertex u with index i is created and the algorithm is used recursively twice to obtain $low(u)$ and $high(u)$. Suppose on the other hand (and without loss of generality) that $id(v_1) = i$ and $id(v_2) > i$. Then the function represented by the OBDD with root v_2 is independent of x_i (because of the fixed variable ordering), that is, $f_2|_{x_i=0} = f_2|_{x_i=1} = f_2$. Hence, a vertex u with index i is created, and the algorithm is recursively applied on $low(v_1)$ and v_2 to generate the OBDD whose root becomes $low(u)$, and on $high(v_1)$ and v_2 to generate the OBDD whose root becomes $high(u)$. This is basically the algorithm except that in order to avoid generating the OBDD for a pair of sub-OBDDs more than once—which would result in exponential (in n) running time—dynamic programming is used: During the algorithm a table of size $|G_1| \cdot |G_2|$ is maintained. The xy'th entry in this table contains the result (the label of the root vertex) of using the algorithm on the vertex in the OBDD for f_1 with label x and the vertex in the OBDD for f_2 with label y, *if* it is already computed. Before applying the algorithm to a pair of vertices it is checked whether the table already contains an entry for the vertices in question. If that is the case the result already computed is just returned. Otherwise, the algorithm continues as described above and adds the root vertex to the table. It is straightforward to realize that Bryant's algorithm runs in $O(|G_1| \cdot |G_2|)$ time (the size of the dynamic programming table). Note that it is proved in [32] that there actually exist reduced OBDDs representing functions f_1 and f_2 such that the size of $f_1 \otimes f_2$ is $\Theta(|G_1| \cdot |G_2|)$.

Due to the recursive structure Bryant's algorithm works in a depth first manner on the involved OBDDs. In [101] an algorithm algorithm working in a breadth first manner is developed in order to perform OBDD manipulation efficiently on a CRAY-type supercomputer. This algorithm works like Bryant's, except that recursive calls (vertices which need to have their children computed—we call them *requests*) are inserted in a queue (the *request* queue) and computed one at a time. This leads to a breadth first traversal of the involved OBDDs. Also this algorithm uses dynamic programming and runs in $O(|G_1| \cdot |G_2|)$ time.

As previously discussed, I/O issues are then taken into account in [17] and an $O(|G_1| \cdot |G_2|)$ time algorithm working in a levelwise manner is developed. As in the case of the reduce algorithm it is assumed that the OBDDs are stored levelwise and the general idea is then to work as levelwise as possible on them. Basically the algorithm works like the breadth first algorithm, but with the request queue split up into n queues—one for each level of the OBDDs. When a new request is generated it is placed in the queue assigned to the level of the vertex corresponding to the request. The queues are then processed one at a time from the queue corresponding to the top level and down. This way the OBDDs are traversed in a

levelwise manner. Also *before* a new request is inserted in a queue it is checked if a duplicate request has already been inserted in the queue. This effectively means that the dynamic programming table and the request queues are "merged" into one structure. Finally, much like the way the reduce algorithm in [102] obtains the new labels of the children of a vertex in level order, the requests on a given level are handled in sorted order according to the levels of the requests issued as a consequence of them. As previously the motivation for this is that it assures that lookups for duplicate requests are done in a levelwise manner.

In order to obtain a canonical OBDD all the presented algorithms run the reduce algorithm after constructing a new OBDD with the apply operation. It should be noted that Bryant in [33] has modified his depth first algorithm such that the reduction is performed as an integrated part of the apply algorithm. The algorithm simply tries to use the reduction rules after returning from the two recursive calls. While it is easy to check if R1 can be applied, a table of the already generated vertices is used to check if R2 can be applied. The advantage of this modified algorithm is that redundant vertices, which would be removed in the following reduction step, is not created and thus space is saved. The algorithms not working in a depth first manner [17, 101, 102] cannot perform the reduction as an integrated part of the apply algorithm.

3.2 The I/O-Behavior of Apply Algorithms

Like we in Section 2.2 analyzed the existing reduce algorithms in the parallel disk model, we will now analyze the different apply algorithms in the model. In the following $|R|$ will be the size of the un-reduced OBDD resulting from a use of the apply algorithm on two OBDDs of size $|G_1|$ and $|G_2|$, respectively. We first analyze why the depth first [32] and breadth first [101] algorithms perform so poorly in an I/O-environment, and then we take a closer look at the algorithms developed with I/O in mind.

As in the case of the reduce algorithm it is not difficult to realize that assuming nothing about the blocking (which is probably the most realistic in practice) it is easy for an adversary to force both the depth first and the breadth first algorithm to do an I/O every time a new vertex in G_1 or G_2 is visited. This results in an overall use of $\Omega(|R|)$ I/Os, that is, $O(|G_1| \cdot |G_2|)$ I/Os in the worst case. It is equally easy to realize that breadth first and level blockings are just as bad for the depth first algorithm [32], and depth first and level blockings just as bad for the breadth first algorithm [101]. So let us assume that the OBDDs are blocked in some "good" way with respect to the used traversal scheme. Still the algorithms perform poorly because of the lack of locality of reference in the lookups in the dynamic programming table. To illustrate this we take a closer look at the depth first algorithm [32] assuming that the OBDDs are depth first blocked. Again, if we do not assume anything about the blocking of the dynamic programming table, it is easy to realize that in the worst case every access to the table results in a page fault. If we were able to block the table as we like, the only obvious way to block it would be in a depth first manner (Figure 6a): Assume that the algorithm is working on vertex v_1 in G_1 and v_2 in G_2. The algorithm now makes one of the following recursive calls: v_1 and $low(v_2)$, $low(v_1)$ and v_2, or $low(v_1)$ and $low(v_2)$. Before doing so it makes a lookup in the dynamic programming table. Thus the table should be blocked as indicated in Figure 6a) as we would like the corresponding part of the table to be in internal memory. Note that with the blocking in Figure 6a) the algorithm would at least make a page fault on every $\sqrt{B}$ lookup operation. But actually it is much worse than that, which can be illustrated with the example in Figure 6b). Here a depth first blocking of an OBDD is

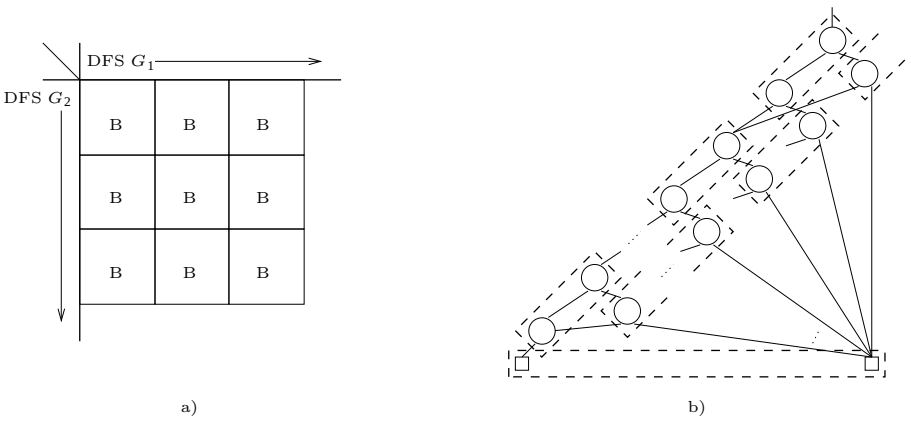


Figure 6: a) Dynamic programming table. b) I/O performance of algorithm in [17].

indicated. It is also indicated how the 0-children of the "right" vertices can be chosen in an almost arbitrary way. This in particular means that an adversary can force the algorithm to make a lookup page fault every time one of these vertices is visited. As the number of such vertices is $\Theta(|G|)$ the algorithm could end up making a page fault for almost every of the $|R|$ vertices in the new OBDD.

After illustrating why the breadth first and depth first algorithms perform poorly, let us shortly consider the algorithm especially designed with I/O in mind [17]. Recall that this algorithm maintains a request queue for each level of the OBDDs, which also functions as the dynamic programming table divided into levels, and that it processes one level of requests in the order of the levels of the requests issued as a consequence of them. It is relatively easy to realize that in the worst case a page fault is generated every time one of the request queues is accessed as dynamic programming table. The reason is precisely the same as in the case of the depth first algorithm, namely that there is no nice pattern in the access to the table—not even in the access to one level of it. As previously, we therefore cannot block the queues efficiently and we again get the $\Omega(|R|)$ worst-case I/O behavior. The natural question to ask is of course why experiments then show that this approach can lead to the mentioned runtime speedups. The answer is partly that the traditional depth first and breadth first algorithms behave so poorly with respect to I/Os that just considering I/O issues, and actually try to block the OBDDs and access them in a "sequential" way, leads to large runtime improvements. Another important reason is the previously mentioned fact that in practical examples n is much smaller than M/B, which means that a block from each of the n queues fits in internal memory. However, we believe that one major reason for the experimental success in [17] is that the OBDDs in the experiments roughly are of the size of the internal memory of the machines used. This means that one level of the OBDDs actually fits in internal memory, which again explains the good performance because the worst case behavior precisely occurs when one level does not fit in internal memory.

3.3 I/O-Efficient Apply Algorithm

The main idea in our new apply algorithm is to do the computation levelwise as in [17], but use a priority queue to control the recursion. Using a priority queue we do not need a queue for each level as in [17]. Furthermore, we do not check for duplicates when new requests

are issued, but when they are about to be computed. Recall that the main problem with the previous levelwise algorithm precisely is the random lookups in the queues/tables when these checks are made. Instead of checking for duplicates when new requests are issued, we just insert them in a priority queue and perform the checks when removing requests from the queue. We do so simply by ordering the queue such that identical requests will be returned by consecutive deletemin operations. This way we can in a simple way ignore requests that have already been computed.

In order to make our algorithm work efficiently we need the vertices of each of the OBDDs sorted according to level and secondary according to label. This representation can easily be constructed in $O(\text{sort}(|G_1|) + \text{sort}(|G_2|))$ I/Os in the same way as we constructed the list L1 in the reduce algorithm. For convenience we now use four priority queues to control the recursion (instead of one). They all contain requests represented by a pair of vertices, one from G_1 and one from G_2. The first queue V contains pairs (v, w) where $id(v) < id(w)$, and is ordered according to the level and label of v. Thus V contains requests which should be processed at level $id(v)$ and which can be processed without obtaining new information about w. Similarly, the second queue W contains pairs where $id(v) > id(w)$, ordered according to $(id(w), w)$. The last two priority queues E_V and E_W contain pairs where $id(v) = id(w)$, and are ordered according to $(id(v), v)$ and $(id(w), w)$, respectively.

We do the apply in a levelwise manner starting from the root vertices. We label the vertices in the resulting OBDD R with pairs of labels from G_1 and G_2. The algorithm starts by comparing the indices of the two root vertices v and w, and creates the root vertex (v, w) of R with index equal to the lowest of the two indices. If $id(v) < id(w)$ it then makes two new vertices $(low(v), w)$ and $(high(v), w)$ with indices $\min(id(low(v)), id(w))$ and $\min(id(high(v)), id(w))$, respectively, and "connects" (v, w) to these two vertices. Similarly, if $id(v) > id(w)$ it makes the vertices $(v, low(w))$ and $(v, high(w))$, and if $id(v) = id(w)$ the vertices $(low(v), low(w))$ and $(high(v), high(w))$. Now the algorithm makes two recursive calls in order to construct the OBDDs rooted in the two newly created vertices. As the recursion is controlled by the priority queues this is done by inserting the vertices/requests in these queues. The level on which a given vertex/request (u_1, u_2) is to be processed is determined by $\min(id(u_1), id(u_2))$. Therefore (u_1, u_2) is inserted in V if $id(u_1) < id(u_2)$ and in W if $id(u_1) > id(u_2)$. If $id(u_1) = id(u_2)$ it is inserted both in E_V and in E_W.

Now assume that the algorithm has processed all levels up to level $i-1$. In order to process level i we do the following: We do deletemin operations on V in order to obtain all requests in this queue that need to be processed on level i. As discussed above we only process one copy of duplicate requests. As all requests (u_1, u_2) in V have $id(u_1) < id(u_2)$ all new requests generated as a consequence of them are of the form $(low(u_1), u_2)$ or $(high(u_1), u_2)$. Thus we do not need any new information about u_2 to issue the new requests. During the process of deleting the relevant requests from the queue we therefore simply "merge" the requests with the representation of G_1 in order to obtain the information needed. We process level i requests in W in a similar way. Finally, we process level i requests in E_V and E_W. We know that all vertices in these queues have $id(u_1) = id(u_2) = i$, which means that they will create requests of the form $(low(u_1), low(u_2))$ and $(high(u_1), high(u_2))$. Therefore we need to obtain new information from both G_1 and G_2. Thus we do deletemin operations on E_V and "merge" the result with the representation of G_1, collecting the information we need from this OBDD. During this process we also insert the resulting vertices in E_W. Finally, we do deletemin operations on E_W and "merge" with G_2 to obtain the information we need to issue the relevant new requests.

When the above process terminates we will have produced R, and the analysis of the I/O-complexity of the algorithm is easy: Each vertex/request is inserted in and deleted from a priority queue a constant number of times. Thus we directly obtain the following from the I/O bounds of the priority queue operations.

Theorem 3 *The apply operation can be performed in* $O(\mathrm{sort}(|R|))$ *I/O operations.*

4 Extension of Results to D-disk Model

As promised we should make a few comments about our results in the D-disk model. As far as the lower bound is concerned, we can of course just divide our bound in the one-disk model by D and the obtained bound will then be a lower bound in the parallel disk model. It turns out that this bound can actually be matched, that is, we can obtain a speedup proportional to the number of disks.

To obtain the upper bounds in this paper we only used three basic "primitives"; scanning, sorting and priority queues. Scanning through N elements can easily be done in $O(N/DB)$ I/Os in the parallel disk model and as already mentioned we can also sort optimally in the model. Furthermore, it is proved in [11] that the priority queue can also take full advantage of parallel disks. Both the sorting algorithms and the priority queue on parallel disks work under the (non-restrictive in practice) assumption that $4DB \leq M - M^{1/2+\beta}$ for some $0 < \beta < 1/2$. Thus with the same assumption all the results obtained in this paper holds in the parallel disk model.

5 Conclusion and Open Problems

In this paper we have demonstrated how all the existing OBDD manipulation algorithms in the worst case make on the order of the number of memory accesses page faults. This is the reason why they perform poorly in an I/O environment. We have also developed new OBDD manipulation algorithms and proved their optimality under some natural assumptions.

We believe that the developed algorithms are not only of theoretical but also of practical interest—especially if we make a couple of modifications. If we represent the OBDDs in terms of edges instead of vertices (where each edge "knows" the level of both source and sink) and block them in the way they are used by the apply algorithm, it can be realized that our apply algorithm automatically produce the blocking used by the (following) reduce algorithm. The reduce algorithm can then again produce the blocking used by the (next) apply algorithm. This can be done without extra I/O use, basically because the apply algorithm works in a top-down manner while the reduce algorithm works in a bottom-up manner. With this modification the algorithms are greatly simplified and we save the I/Os used to create the special representations of the OBDDs used in the reduce and apply algorithms. Furthermore, it is also easy to realize that we can do with only one priority queues in the apply algorithm. As the constants in the I/O bounds on the priority queue operations are all small, the constants in the bounds of the developed OBDD manipulation algorithms are also small. Also it is demonstrated in [129] that the overhead required to explicitly manage I/O can be made very small, and therefore we believe that our algorithms could lead to large runtime speedups on existing workstations. We hope in the future to be able to implement the priority queue data

structure in the Transparent Parallel I/O Environment (TPIE) developed by Vengroff [126] in order to verify this.

A couple of questions remains open, namely if it is possible to prove an $O(\mathrm{perm}(N))$ I/O lower bound on the reduce operation assuming *any* blocking, and if it is possible to prove a lower bound on the apply operation without assuming that a reduce step is done after the apply.

Acknowledgments

I would especially like to thank Peter Bro Miltersen for comments on drafts of this paper and for comments that lead to the definition of the blocked red-blue pebble game. I would also like to thank Allan Cheng for introducing me to OBDDs and Darren Erik Vengroff for inspiring discussions.

Chapter 10

A General Lower Bound on the I/O-Complexity of Comparison-based Algorithms

A General Lower Bound on the I/O-Complexity of Comparison-based Algorithms*

Lars Arge, Mikael Knudsen and Kirsten Larsen

Department of Computer Science
University of Aarhus
Aarhus, Denmark†

August 1992

Abstract

We show a general relationship between the number of comparisons and the number of I/O-operations needed to solve a given problem. This relationship enables one to show lower bounds on the number of I/O-operations needed to solve a problem whenever a lower bound on the number of comparisons is known. We use the result to show lower bounds on the I/O-complexity of a number of problems where known techniques only give trivial bounds. Among these are the problems of removing duplicates from a multiset, a problem of great importance in e.g. relational data-base systems, and the problem of determining the mode — the most frequently occurring element — of a multiset. We develop non-trivial algorithms for these problems in order to show that the lower bounds are tight.

1 Introduction

In the studies of complexity of algorithms, most attention has been given to bounding the number of primitive operations (for example comparisons) needed to solve a given problem. However, when working with data materials so large that they will not fit into internal memory, the amount of time needed to transfer data between the internal memory and the external storage (the number of I/O-operations) can easily dominate the overall execution time.

In this paper we work in a model introduced by Aggarwal and Vitter [5] where an I/O-operation swaps B records between external storage and the internal memory, capable of holding M records. An algorithm for this model is called an I/O-algorithm. Aggarwal and Vitter [5] consider the I/O-complexity of a number of specific sorting-related problems, namely sorting, fast Fourier transformation, permutation networks, permuting and matrix transposition. They give asymptotically matching upper and lower bounds for these problems. The

*This paper is a slightly revised version of DAIMI PB-407. The paper was presented at the Third Workshop on Algorithms and Data Structures (WADS'93).

†This work was partially supported by the ESPRIT II Basic Research Actions Program of the EC under contract No. 7141 (project ALCOM II). Email communication to large@daimi.aau.dk.

lower bounds are based on routing arguments, and in general no restrictions are made on the operations allowed in internal memory, except that records are considered to be atomic and cannot be divided into smaller parts. Only when the internal memory is extremely small, the comparison model is assumed.

In this paper we shall use the same model of computation, except that in general we will limit the permitted operations in the internal memory to comparisons. The main result of this paper is the following: Given an I/O-algorithm that solves an N-record problem P_N using $I/O(\overline{x})$ I/O's on the input $\overline{x}$, there exists an ordinary comparison algorithm that uses no more than $N \log B + I/O(\overline{x}) \cdot T_{\text{merge}}(M-B, B)$ comparisons on input $\overline{x}$. $T_{\text{merge}}(n, m)$ denotes the number of comparisons needed to merge two sorted lists, of size n and m respectively. While [5] shows lower bounds for a number of specific problems, our result enables one to show lower bounds on the number of I/O-operations needed to solve a problem for any problem where a lower bound on the number of comparisons needed is known. Among these is sorting where we obtain the same lower bound as in [5]. We use the result to show lower bounds on a number of problems not formerly considered with respect to I/O-complexity. Among these are the problems of removing duplicates from a multiset, a problem of great importance in e.g. relational data-base systems, and the problem of finding the most frequently occurring element in a multiset. We develop non-trivial algorithms for these problems in order to show that the lower bounds are tight.

The basic idea in the lower bound proofs in [5] is to count how many permutations can be generated with a given number of I/O-operations and to compare this to the number of permutations needed to solve a problem. This technique, however, is not generally applicable. For example only trivial lower bounds can be shown on the problems we consider in this paper using this technique. We use a different information theoretical approach, where we extend normal comparison trees with I/O-nodes. The proof of the main result then corresponds to giving an algorithm that transfers an I/O-tree that solves a given (I/O-) problem to a normal comparison tree with a height bounded by a function of the number of I/O-nodes in the original tree. An important property of the result is that it not only can be used to show worst-case I/O lower bounds, but that e.g an average lower bound on the number of comparisons needed also induces an average I/O lower bound.

In the next section, we formally define the model we will be working in. We also define the I/O-tree on which the main result in section 3 is based. In section 4, we discuss the generality of the I/O-tree model, and in section 5, we give optimal algorithms for two problems concerning multisets, namely determining the mode and removing duplicates. Finally, some open problems are discussed in section 6.

2 Definition of the model and the I/O-tree

We shall consider N-record problems, where in any start configuration the N records — $x_1, x_2, \ldots, x_N$ — reside in secondary storage. The number of records that can fit into internal memory is denoted M and the number of records transferable between internal memory and secondary storage in a single block is denoted B ($1 \leq B \leq M < N$). The internal memory and the secondary storage device together are viewed as an extended memory with at least $M+N$ locations. The first M locations in the extended memory constitute the internal memory — we denote these $s[1], s[2], \ldots, s[M]$ — and the rest of the extended memory constitute secondary storage. The k'th *track* is defined as the B contiguous locations $s[M+(k-1)B+1], s[M+$

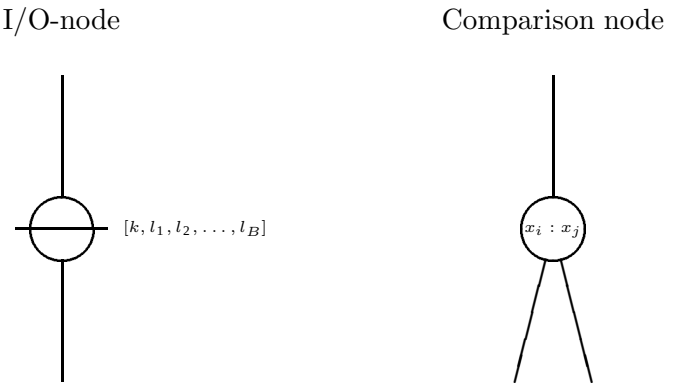


Figure 1: Node-types: An *I/O-node* swaps the B records $s(l_1), \ldots, s(l_B)$ with the B records in the k'th track, as denoted by the I/O-vector $[k, l_1, l_2, \ldots, l_B]$, where $l_1, l_2, \ldots, l_B \in \{1, \ldots M\}$ and are pairwise different, and $k \in \{1, 2, \ldots\}$. A *comparison node* compares x_i with x_j. x_i and x_j must be in internal memory.

$(k-1)B+2], \ldots, s[M+kB]$ in extended memory, $k = 1, 2, \ldots$. An I/O-operation is now an exchange of B records between the internal memory and a track in secondary storage.

An I/O-tree is a tree with two types of nodes: comparison nodes and I/O-nodes. Comparison nodes compare two records x_i and x_j in the *internal memory* using $<$ or $\leq$. i and j refer to the initial positions of the records and *not* to storage locations. A comparison node has two outgoing edges, corresponding to the two possible results of the comparison. An I/O-node performs an I/O-operation, that is, it swaps B (possibly empty) records in the internal memory with B (possibly empty) records from secondary storage. The B records from secondary storage must constitute a track (see Figure 1).

To each I/O-node, we attach a predicate Q and two functions π and π'. The predicate Q contains information about the relationship between the x_i's. We define the predicate recursively: First we attach a predicate P_k to each edge from a comparison node k. If the node made the comparison $x_i < x_j$ the predicate $x_i < x_j$ is attached to the left edge, and $x_i \geq x_j$ to the right edge. Similarly with $\leq$. We now consider a path S where we number the I/O-nodes $s_0, s_1, s_2, \ldots$ starting in the root and ending in the leaf.
Q_{s_i} is then defined as follows: $Q_{s_0} = True$
$Q_{s_i} = Q_{s_{i-1}} \wedge P_1 \wedge P_2 \wedge \ldots \wedge P_l$
where P_1, $P_2 \ldots P_l$ are the predicates along the path from s_{i-1} to s_i (see Figure 2).

The π's contain information about where in the extended storage the original N records — $x_1, x_2, \ldots, x_N$ — are placed. More formally, we have: $\pi : \{1, 2, \ldots, N\} \rightarrow \{1, 2, \ldots\}$, where $\pi(i) = j$ means that x_i is in the jth cell in the extended memory. Note that π is one-to-one. π' is the result of performing an I/O-operation in a configuration described by π, i.e., a track, consisting of B records, is swapped with B records from the internal memory (as denoted by the $(B+1)$-vector in Figure 1). More formally, $\pi' = \pi$ except for the following:

$$\begin{aligned}
\pi'(\pi^{-1}(l_1)) &= M + (k-1)B + 1 \\
\pi'(\pi^{-1}(M + (k-1)B + 1)) &= l_1 \\
&\vdots \\
\pi'(\pi^{-1}(l_B)) &= M + kB \\
\pi'(\pi^{-1}(M + kB)) &= l_B
\end{aligned}$$

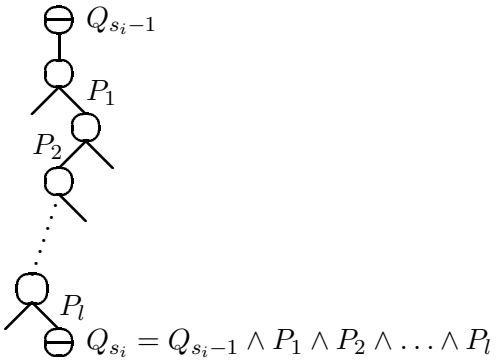


Figure 2: The predicate Q_{s_i} is defined recursively from the predicate $Q_{s_{i-1}}$ and the predicates along the path between the two I/O-nodes.

π in an I/O-node is, of course, equal to π' in its closest ancestor, i.e., $\pi_{s_i} = \pi'_{s_{i-1}}$.

Definition 1 *An I/O-tree is a tree consisting of comparison and I/O-nodes. The root of the tree is an I/O-node where $\pi_{root}(i) = M + i$, i.e. corresponding to a configuration where there are N records residing first in secondary storage and the internal memory is empty. The leaves of the tree are I/O-nodes, again corresponding to a configuration where the N records reside first in secondary storage (possibly permuted with respect to the start configuration) and the internal memory is empty. This means that $\pi'_{leaf}(i) \in \{M+1, M+2 \ldots M+N\}$ for all i.*

Definition 2 *If T is an I/O-tree then* $\mathrm{path}_T(\overline{x})$ *denotes the path $\overline{x}$ follows in T.* $|\mathrm{path}_T(\overline{x})|$ *is the number of nodes on this path.*

We split the problems solvable by I/O-trees into two classes: decision problems and construction problems. Decision problems are problems where we, given a predicate Q_P and a vector $\overline{x}$, want to decide whether or not $\overline{x}$ satisfies Q_P. Construction problems are problems where we are given a predicate Q_P and a vector $\overline{x}$, and want to make a permutation ρ, such that $\rho(\overline{x})$ satisfies Q_P.

Definition 3 *An I/O-tree T solves a decision problem P, if the following holds for every leaf l:*

$$\begin{array}{lll} (\forall \overline{x} : Q_l(\overline{x}) & \Rightarrow & Q_P(\overline{x})) \quad \vee \\ (\forall \overline{x} : Q_l(\overline{x}) & \Rightarrow & \neg Q_P(\overline{x})) \end{array}$$

An I/O-tree T solves a construction problem P, if the following holds for every leaf l:

$$\forall \overline{x} : Q_l(\overline{x}) \Rightarrow Q_P(x_{\pi_l'^{-1}(M+1)}, x_{\pi_l'^{-1}(M+2)}, \ldots, x_{\pi_l'^{-1}(M+N)})$$

It is important to note that an I/O-tree reduces to an ordinary comparison tree solving the same problem, if the I/O-nodes are removed. This is due to the fact that the comparison nodes refer to records (numbered with respect to the initial configuration) and not to storage locations.

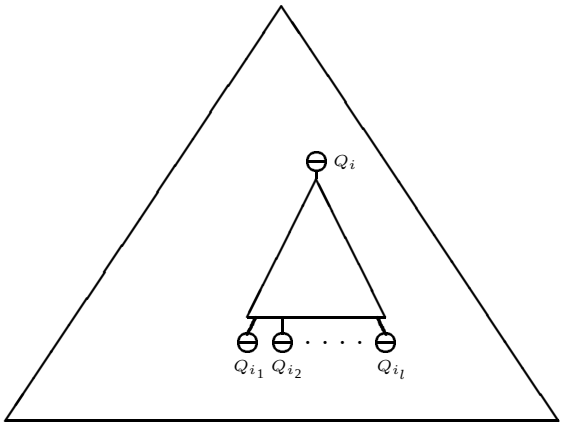


Figure 3: Comparison subtree

3 The Main Result

Theorem 1 *Let P_N be an N-record problem, T be an I/O-tree solving P_N and let $I/O_T(\overline{x})$ denote the number of I/O-nodes in* $\text{path}_T(\overline{x})$*. There exists an ordinary comparison tree T_c solving P_N such that the following holds:*

$$|\text{path}_{T_c}(\overline{x})| \leq N \log B + I/O_T(\overline{x}) \cdot T_{\text{merge}}(M-B, B)$$

where $T_{\text{merge}}(n, m)$ denotes the number of comparisons needed to merge two sorted lists of length n and m, respectively.

Proof: We will prove the theorem by constructing the comparison tree T_c, but first we want to construct another I/O-tree T' that solves P_N from the I/O-tree T.

We consider a *comparison subtree* of T — an inner comparison tree of T with an I/O-node as the root and its immediately succeeding I/O-nodes as the leaves (see figure 3).

A characteristic of this tree is that, except from the I/O-nodes in the root and in the leaves, it only contains comparison nodes that compare records in the internal memory, i.e. comparisons of the form $x_i < x_j$ $(x_i \leq x_j)$ where $\pi(i), \pi(j) \in \{1, .., M\}$. In other words $Q_{i_1}, Q_{i_2}, \ldots, Q_{i_l}$ must be of the form $Q_i \wedge (x_{i_1} < x_{j_1}) \wedge (x_{i_2} \leq x_{j_2}) \wedge \ldots$ where $\pi(i_m), \pi(j_m) \in \{1, .., M\}$. Moreover, one and only one of the predicates $Q_{i_1}, Q_{i_2}, \ldots, Q_{i_l}$ is true for any $\overline{x}$ that satisfies Q_i.

We now build T' from T by inductively building comparison subtrees in T' from comparison subtrees in T starting with the "uppermost" comparison subtree: The root of the new comparison subtree is the same as the root of the original comparison subtree. The internal comparison nodes are replaced with a tree that makes all the comparisons needed for a total ordering of records in internal memory. Finally, the leaves are I/O-nodes selected among the l I/O-nodes in the original subtree in the following way: If R is the predicate "generated" on the path from the root of T' to a leaf in the new subtree, the I/O-node with the predicate Q_{i_j} such that $R \Rightarrow Q_{i_j}$ is used. The choice of I/O-node is well-defined because the predicate R implies exactly one of the Q_{i_j}'s. If any of the leaves in the original comparison subtree are also roots of comparison subtrees, i.e., they are not the leaves of T, we repeat the process for each of these subtrees. Note that any of them may appear several times in T'. It

should be clear that when T' is constructed in this way, it solves P_N. Furthermore, for all $\overline{x}$, $\text{path}_T(\overline{x})$ and $\text{path}_{T'}(\overline{x})$ contain the same I/O-nodes. This means that if the height of the comparison subtrees in T' is at most h, then the number of comparison nodes on $\text{path}_{T'}(\overline{x})$ is at most $h \cdot I/O_T(\overline{x})$. But then there exists an ordinary comparison tree T_c solving P_N, such that $|\text{path}_{T_c}(\overline{x})| \leq h \cdot I/O(\overline{x})$, namely the comparison tree obtained from T' by removing the I/O-nodes.

It is obvious that our upper bound on $|\text{path}_{T_c}(\overline{x})|$ improves the smaller an h we can get. This means that we want to build a comparison tree, that after an I/O-operation determines the total order of the M records in internal memory with as small a height as possible. After an I/O-operation we know the order of the $M - B$ records that were not affected by the I/O-operation — this is an implicit invariant in the construction of T'. The problem is, therefore, limited to placing the B "new" records within this ordering. If we, furthermore, assume that we know the order of the B records, then we are left with the problem of merging two ordered lists, this can be done using at most $T_{merge}(M - B, B)$ comparisons. We cannot in general assume that the B records are ordered, but because the I/O-operations always are performed on tracks and because we know the order of the records we write to a track, the number of times we can read B records that are not ordered (and where we must use $B \log B$ comparisons to sort them) cannot exceed $\frac{N}{B}$.
Finally, we get the desired result:

$$|\text{path}_{T_c}(\overline{x})| \leq \tfrac{N}{B} B \log B + I/O_{T'}(\overline{x}) \cdot T_{\text{merge}}(M - B, B)$$
$$\Downarrow$$
$$|\text{path}_{T_c}(\overline{x})| \leq N \log B + I/O_T(\overline{x}) \cdot T_{\text{merge}}(M - B, B)$$

□

Two lists of length n and m (where $n > m$) can be merged using *binary merging* [82] in $m + \lfloor \frac{n}{2^t} \rfloor - 1 + t \cdot m$ comparisons where $t = \lfloor \log \frac{n}{m} \rfloor$. This means that $T_{\text{merge}}(M - B, B) \leq B \log(\frac{M-B}{B}) + 3B$ which gives us the following corollary:

Corollary 1

$$|\text{path}_{T_c}(\overline{x})| \leq N \log B + I/O_T(\overline{x}) \cdot \left(B \log(\frac{M - B}{B}) + 3B \right)$$

□

It should be clear that the corollary can be used to prove lower bounds on the number of I/O-operations needed to solve a given problem. An example is sorting, where an $N \log N - O(N)$ worst-case lower bound on the number of comparisons is known. In other words, we know that for any comparison tree (algorithm) T_c that sorts N records there is an $\overline{x}$ such that $|\text{path}_{T_c}(\overline{x})| \geq N \log N - O(N)$. From the corollary we get $N \log N - O(N) \leq N \log B + I/O_T(\overline{x}) \cdot \left(B \log(\frac{M-B}{B}) + 3B \right)$, hence the worst-case number of I/O-operations needed to sort N records is at least $\frac{N \log \frac{N}{B} - O(N)}{B \log(\frac{M-B}{B}) + 3B}$.
Note that no matter what kind of lower bound on the number of comparisons we are working with - worst-case, average or others - the theorem applies, because it relates the number of I/O's and comparisons for *each instance* of the problem.

4 Extending the Model

The class of algorithms for which our result is valid comprises algorithms that can be simulated by our I/O-trees. This means that the only operations permitted are binary comparisons and transfers between secondary storage and internal memory. It should be obvious that a tree, using ternary comparisons and swapping of records in internal memory, can be simulated by a tree with the same I/O-height, that only uses binary comparisons and no swapping (swapping only effects the $\pi's$). Therefore, a lower bound in our model will also be a lower bound in a model where swapping and ternary comparisons are permitted. Similarly, we can permit algorithms that use integer variables, if their values are implied by the sequence of comparisons made so far, and we can make branches according to the value of these variables. This is because such manipulations cannot save any comparisons.

The differences between our model and the model presented in [5] are, apart from ours being restricted to a comparison model, mainly three things. Firstly, Aggarwal and Vitter only assume that a transfer involves B contiguous records in secondary storage, whereas we assume that the B records constitute a track. Reading/writing across a track boundary, however, can be simulated by a constant number of "standard" I/O's. Hence, lower bounds proved in our model still apply asymptotically. Secondly, their I/O's differ from ours in the sense that they permit copying of records, i.e. writing to secondary storage without deleting them from internal memory. It can be seen that the construction in the proof of our theorem still works, if we instead of one I/O-node have both an I-node and an O-node that reads from, respectively writes to, a track. Therefore, our theorem still holds when record copying is permitted. Finally, they model parallelism with a parameter P that represents the number of blocks that can be transferred concurrently. It should be clear that we can get lower bounds in the same model by dividing lower bounds proved in our model by P.

It is worth noting that this parallel model is not especially realistic. A more realistic model was considered in [133] in which the secondary storage is partitioned into P distinct disk drives. In each I/O-operation, each of the P disks can simultaneously transfer one block. Thus, P blocks can be transferred per I/O, but only if no two blocks come from the same disk. Of course lower bounds in the Aggarwal and Vitter model also apply in the more realistic model. As using multiple disks is a very popular way of speeding up e.g. external sorting, extensive research has recently been done in this area [99] [131].

5 Optimal Algorithms

Aggarwal and Vitter [5] show the following lower bound on the I/O-complexity of sorting:

$$\Omega\left(\frac{N\log\frac{N}{B}}{B\log\frac{M}{B}}\right)$$

They also give two algorithms based on mergesort and bucketsort that are asymptotically optimal. As mentioned earlier our result provides the same lower bound.

An almost immediate consequence of the tight lower bound on sorting is a tight lower bound on set equality, set inclusion and set disjointness, i.e., the problems of deciding whether $A = B$, $A \subseteq B$ or $A \cap B = \emptyset$ given sets A and B. It can easily be shown (see e.g. [22]) that a lower bound on the number of comparisons for each of these problems is $N \log N - O(N)$.

An optimal algorithm is, therefore, to sort the two sets independently, and then solving the problem by "merging" them.

In the following, we will look at two slightly more difficult problems for which our theorem gives asymptotically tight bounds.

5.1 Duplicate Removal

We wish to remove duplicates from a file in secondary storage — that is, make a set from a multiset. Before removing the duplicates, N records reside at the beginning of the secondary storage and the internal memory is empty. The goal is to have the constructed set residing first in the secondary storage and the duplicates immediately after. Formally this corresponds to the following predicate:

$$\begin{aligned} Q_P(\overline{y}) \quad = \exists k: \quad & (\forall i,j \;\; 1 \leq i,j \leq k \;\wedge\; i \neq j : y_i \neq y_j) \wedge \\ & (\forall i \;\; k < i \leq N : \exists j \;\; 1 \leq j \leq k : y_i = y_j) \end{aligned}$$

A lower bound on the number of comparisons needed to remove duplicates is $N \log N - \sum_{i=1}^{k} N_i \log N_i - O(N)$, where N_i is the multiplicity of the ith record in the set. This can be seen by observing that after the duplicate removal, the total order of the original N records is known. Any two records in the constructed set must be known not to be equal, and because we compare records using only $<$ or $\leq$, we know the relationship between them. Any other record (i.e. one of the duplicates) equals one in the set. As the total order is known, the number of comparisons made must be at least the number needed to sort the initial multiset. A lower bound on this has been shown [93] to be $N \log N - \sum_{i=1}^{k} N_i \log N_i - O(N)$.

Combining a trivial lower bound of $\frac{N}{B}$ (we have to look at each record at least once) with an application of our theorem to the above comparison lower bound, we obtain:

$$I/O(\text{Duplicate}-\text{Removal}_N) \in \Omega\left(\max\left\{\frac{N \log \frac{N}{B} - \sum_{i=1}^{k} N_i \log N_i}{B \log \frac{M}{B}}, \frac{N}{B}\right\}\right)$$

We match this lower bound asymptotically with an algorithm that is a variant of merge sort, where we "get rid of" duplicates as soon as we meet them. We use a block (of B records) in internal memory to accumulate duplicates, transferring them to secondary storage as soon as the block runs full.

The algorithm works like the standard merge sort algorithm. We start by making $\lceil \frac{N}{M-B} \rceil$ runs; we fill up the internal memory $\lceil \frac{N}{M-B} \rceil$ times and sort the records, removing duplicates as described above. We then repeatedly merge c runs into one longer run until we only have one run, containing k records. $c = \lfloor \frac{M}{B} \rfloor - 2$ as we use one block for duplicates and one for the "outgoing" run. It is obvious that there are less than $\log_c(\lceil \frac{N}{M-B} \rceil) + 1$ phases in this merge sort, and that we in a single phase use no more than the number of records being merged times $\frac{2}{B}$ I/O-operations.

We now consider records of type x_i. In the first phase we read all the N_i records of this type. In phase j there are less than $\frac{\lceil N/(M-B) \rceil}{c^{j-1}}$ runs and we therefore have two cases:

$\frac{\lceil N/(M-B) \rceil}{c^{j-1}} \geq N_i$: There are more runs than records of the type x_i, this means that in the worst case we have not removed any duplicates, and therefore all the N_i records contribute to the I/O-complexity.

$\frac{\lceil N/(M-B)\rceil}{c^{j-1}} < N_i$: There are fewer runs than the original number of x_i's. There cannot be more than one record of the type x_i in each run and therefore the recordtype x_i contributes with no more than the number of runs to the I/O-complexity.

The solution to the equation $\frac{\lceil N/(M-B)\rceil}{c^{j-1}} = N_i$ with respect to j gives the number of phases where all N_i records might contribute to the I/O-complexity. The solution is $j = \log_c(\frac{\lceil N/(M-B)\rceil}{N_i}) + 1$, and the number of times the recordtype x_i contributes to the overall I/O-complexity is no more than:

$$N_i \left(\log_c(\frac{\lceil N/(M-B)\rceil}{N_i}) + 1 \right) + \sum_{j=\log_c(\frac{\lceil N/(M-B)\rceil}{N_i})+2}^{\log_c(\lceil N/(M-B)\rceil)+1} \frac{\lceil N/(M-B)\rceil}{c^j}$$

Adding together the contributions from each of the k records we get the overall I/O-complexity:

$$\begin{aligned}
I/O &\leq \frac{2}{B}\left[N + \sum_{i=1}^{k} \left(N_i \left(\log_c(\frac{\lceil N/(M-B)\rceil}{N_i}) + 1 \right) + \sum_{j=\log_c(\frac{\lceil N/(M-B)\rceil}{N_i})+2}^{\log_c(\lceil N/(M-B)\rceil)+1} \frac{\lceil N/(M-B)\rceil}{c^j} \right) \right] \\
&= \frac{2}{B}\left[N + N\log_c(\lceil N/(M-B)\rceil) - \sum_{i=1}^{k} N_i \log_c N_i + N + \right. \\
&\quad \left. \sum_{i=1}^{k} \lceil N/(M-B)\rceil \cdot \left(\sum_{j=0}^{\log_c(\lceil N/(M-B)\rceil)+1} (c^{-1})^j - \sum_{j=0}^{\log_c(\frac{\lceil N/(M-B)\rceil}{N_i})+1} (c^{-1})^j \right) \right] \\
&= \frac{2}{B}\left[N + N\log_c(\lceil N/(M-B)\rceil) - \sum_{i=1}^{k} N_i \log_c N_i + N + \frac{N-k}{c^2-c} \right] \\
&= 2\frac{N\log(\lceil N/(M-B)\rceil) - \sum_{i=1}^{k} N_i \log N_i}{B\log(\lceil \frac{M}{B}\rceil - 2)} + \frac{4N}{B} + \frac{2(N-k)}{B(c^2-c)} \\
&\in O\left(\max\left\{ \frac{N\log\frac{N}{B} - \sum_{i=1}^{k} N_i \log N_i}{B\log\frac{M}{B}}, \frac{N}{B} \right\} \right)
\end{aligned}$$

5.2 Determining the Mode

We wish to determine the mode of a multiset, i.e. the most frequently occurring record. In a start configuration, the N records reside at the beginning of the secondary storage. The goal is to have an instance of the most frequently occurring record residing first in secondary storage and all other records immediately after. Formally this corresponds to the following predicate:

$$Q_P(\overline{y}) = \forall j \;\; 1 \leq j \leq N : |\{i \mid y_i = y_1, 1 \leq i \leq N\}| \geq |\{i \mid y_i = y_j, 1 \leq i \leq N\}|$$

Munro and Raman [93] showed that $N\log\frac{N}{a} - O(N)$ is a lower bound on the number of ternary comparisons needed to determine the mode, where a denotes the frequency of the mode. This must also be a lower bound on the number of binary comparisons, thus, our theorem, again combined with a trivial lower bound of $\frac{N}{B}$, gives the following lower bound on

the number of I/O-operations:

$$I/O(\text{mode}_N) \in \Omega\left(\max\left\{\frac{N\log\frac{N}{aB}}{B\log\frac{M}{B}}, \frac{N}{B}\right\}\right)$$

The algorithm that matches this bound is inspired by the distribution sort algorithm presented by Munro and Spira [94]. First, we divide the multiset into c disjoint segments of roughly equal size (a segment is a sub-multiset which contains all elements within a given range). We then look at each segment and determine which records (if any) have multiplicity greater than the segment size divided by a constant l (we call this an l-majorant). If no segments contained an l-majorant, the process is repeated on each of the segments. If, on the other hand, there were any l-majorants, we check whether the one among these with the greatest multiplicity has multiplicity greater than the size of *the largest segment* divided by l. If it does, we have found the mode. If not, we continue the process on each of the segments as described above.

We now argue that both the division into segments and the determination of l-majorants can be done in a constant number of sequential runs through each segment.

To determine l-majorants we use an algorithm due to Misra and Gries [91]. First, $l-1$ candidates are found in a sequential run through the segment in the following way: For each record it is checked whether it is among the present $l-1$ candidates (initially each of the $l-1$ candidates are just "empty"). If it is, this candidates multiplicity is incremented by $l-1$. If not, all the candidates multiplicities are decremented by 1, unless any of the candidates had multiplicity 0 (or was "empty"), in which case the record becomes a candidate with multiplicity $l-1$ instead of one with multiplicity 0. When the run is completed, if there were any l-majorants, they will be among the candidates with positive multiplicity. This is checked in another sequential run, where the actual multiplicities of the candidates are determined. Note that l must be less than $M-B$ because the l candidates have to be in internal memory.

The division of a segment into c disjoint segments of roughly equal size is done by first finding c pivot elements, and then distributing the records in the original segment into the segments defined by the pivot elements in a sequential run. We use one block in internal memory for the ingoing records, and c blocks, one for each segment, for the outgoing records (this means $c \leq \lfloor\frac{M}{B}\rfloor - 1$). Aggarwal and Vitter [5] describe an algorithm to find $\sqrt{\frac{M}{B}}$ pivot elements in a *set* in $O(\frac{N}{B})$ I/O's. The algorithm satisfies the following: If $d_1, d_2, \ldots, d_{\sqrt{\frac{M}{B}}}$ denote the pivot elements and K_i denotes the elements in $[d_{i-1}, d_i[$ then $\frac{N}{2\sqrt{\frac{M}{B}}} \leq |K_i| \leq \frac{3N}{2\sqrt{\frac{M}{B}}}$ ($\star$). We now wish to argue that a slightly modified version of the algorithm can be used to find pivot elements in a *multiset* so that either $|K_i| \leq \frac{3N}{\sqrt{\frac{M}{B}}}$ or else all the records in K_i are equal.

We start by using the algorithm by Aggarwal and Vitter. This algorithm depends almost exclusively on the *k-selection* algorithm that finds the kth smallest element in a multiset in linear time [29]. This means that if we implicitly assume an order of equal records, namely the order in which we meet them, the resulting pivot elements define segments that satisfy ($\star$). Some of the pivot elements might be equal, and we therefore use some slightly different elements. If $d_{i-1} \neq d_i = d_{i+1} = \ldots = d_{i+k} \neq d_{i+k+1}$, we use the elements $d_{i-1}, d_i, succ(d_{i+k})$ and d_{i+k+1}, where $succ(d)$ is the successor to d in the record order (in the case where $succ(d_{i+k}) = d_{i+k+1}$, we only choose the three pivot elements d_{i-1}, d_i and $succ(d_{i+k})$). The segments now either consist of all equal records, or else they are no more than double the size of the segments

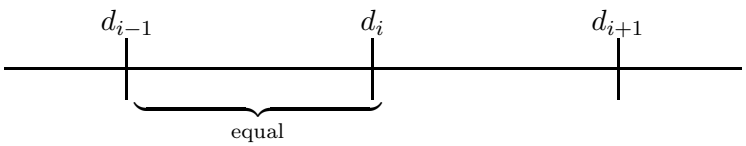


Figure 4: "The worst case": $d_{i-1} \neq d_i \neq d_{i+1}$ and $[d_{i-1}, d_i[$ contains one element whereas $[d_i, d_{i+1}[$ is double the intended size.

we get, assuming that all records are distinct. This can be seen by looking at the "worst case" which is when $d_{i-1} \neq d_i \neq d_{i+1}$ and all records in $]d_{i-1}, d_i[$ are equal, and consequently $[d_i, d_{i+1}[$ and $]d_{i-1}, d_{i+1}[$ contain the same elements (see Figure 4).

We have argued that the number of I/O-operations at each level is proportional to N/B, and the analysis therefore reduces to bounding the number of levels. An upper bound on the size of the largest segment on level j must be $\frac{N}{(\frac{1}{3}\sqrt{M/B})^j}$. It follows that the algorithm can run no longer than to a level j where $\frac{N}{(\frac{1}{3}\sqrt{M/B})^j}/l \leq a$. Solving this inequality with respect to j, we find that no matter what value of l we choose in the range $\{B, \ldots, M - B\}$, we get a bound on the number levels of $O\left(\frac{\log \frac{N}{aB}}{\log \frac{M}{B}}\right)$. This gives us the matching upper bound of $O\left(\frac{N \log \frac{N}{aB}}{B \log \frac{M}{B}}\right)$.

6 Remarks and Open Problems

In the previous section we showed tight bounds on the problems of removing duplicates from a multiset, and determining the mode of a multiset. As mentioned in the introduction, previously known techniques [5] give only trivial lower bounds on these problems. On the other hand our theorem is also limited in the sense that there are problems for which it is useless. One example is the problem of permuting N records according to a given permutation π. An interesting and important problem "lying between" duplicate removal and permuting is multiset sorting. This problem is analyzed in [81], and lower bounds are given, both using our theorem and a (reduction-) variant of the technique from [5]. The obtained lower bounds are quite good, but we believe there is room for improvement.

Another interesting problem is to extend the model in which the lower bounds apply. Especially it would be interesting to extend our theorem to an I/O version of algebraic decision trees - thus allowing arithmetic. This would probably give interesting bounds on e.g. a number of computational geometry problems.

Acknowledgments

The authors thank Gudmund S. Frandsen, Peter Bro Miltersen and Erik Meineche Schmidt for valuable help and inspiration. Special thanks go to Peter Bro Miltersen and Erik Meineche Schmidt for carefully reading drafts of this paper and providing constructive criticism.

Bibliography

[1] A. Aggarwal, B. Alpern, A. K. Chandra, and M. Snir. A model for hierarchical memory. In *Proc. ACM Symp. on Theory of Computation*, pages 305–314, 1987.

[2] A. Aggarwal and A. K. Chandra. Virtual memory algorithms. In *Proc. ACM Symp. on Theory of Computation*, pages 173–185, 1988.

[3] A. Aggarwal, A. K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 204–216, 1987.

[4] A. Aggarwal and G. Plaxton. Optimal parallel sorting in multi-level storage. *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 659–668, 1994.

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

[6] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, MA, 1974.

[7] B. Alpern, L. Carter, and E. Feig. Uniform memory hierarchies. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 600–608, 1990.

[8] R. J. Anderson and G. L. Miller. A simple randomized parallel algorithm for list-ranking. *Information Processing Letters*, 33:269–273, 1990.

[9] D. S. Andrews, J. Snoeyink, J. Boritz, T. Chan, G. Denham, J. Harrison, and C. Zhu. Further comparisons of algorithms for geometric intersection problems. In *Proc. 6th Int'l. Symp. on Spatial Data Handling*, 1994.

[10] ARC/INFO. *Understanding GIS—the ARC/INFO method*. ARC/INFO, 1993. Rev. 6 for workstations.

[11] 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.

[12] 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.

[13] 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.

[14] L. Arge and P. B. Miltersen. On the indivisibility assumption in the theory of external-memory algorithms. In preparation.

[15] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proc. Annual European Symposium on Algorithms, LNCS 979*, pages 295–310, 1995. A full version is to appear in special issue of Algorithmica.

[16] L. Arge and J. S. Vitter. Optimal dynamic interval management in external memory. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, 1996.

[17] P. Ashar and M. Cheong. Efficient breadth-first manipulation of binary decision diagrams. In *Proc. IEEE International Conference on CAD*, 1994.

[18] M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: A technique for designing parallel algorithms. *SIAM Journal of Computing*, 18(3):499–532, 1989.

[19] R. D. Barve, E. F. Grove, and J. S. Vitter. Simple randomized mergesort on parallel disks. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, 1996.

[20] H. Baumgarten, H. Jung, and K. Mehlhorn. Dynamic point location in general subdivisions. *Journal of Algorithms*, 17:342–380, 1994.

[21] R. Bayer and E. McCreight. Organization and maintenance of large ordered indizes. *Acta Informatica*, 1:173–189, 1972.

[22] M. Ben-Or. Lower bounds for algebraic computation trees. In *Proc. ACM Symp. on Theory of Computation*, pages 80–86, 1983.

[23] J. L. Bentley. Algorithms for klee's rectangle problems. Dept. of Computer Science, Carnegie Mellon Univ., unpublished notes, 1977.

[24] J. L. Bentley. Multidimensional divide and conquer. *Communications of the ACM*, 23(6):214–229, 1980.

[25] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Transactions on Computers*, C-28(9):643–647, 1979.

[26] J. L. Bentley and D. Wood. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers*, 29:571–577, 1980.

[27] G. Blankenagel and R. Güting. XP-trees — External priority search trees. Technical report, FernUniversität Hagen, Informatik-Bericht Nr. 92, 1990.

[28] G. Blankenagel and R. Güting. External segment trees. *Algorithmica*, 12:498–532, 1994.

[29] M. Blum, R. W. Floyd, V. Pratt, R. L. Rievest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

[30] N. Blum and K. Mehlhorn. On the average number of rebalancing operations in weight-balanced trees. *Theoretical Computer Science*, 11:303–320, 1980.

[31] S. K. Brace, R. L. Rudell, and R. E. Bryant. Efficient implementation of a BDD package. In *Proc. ACM/IEEE Design Automation Conference*, 1990.

[32] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[33] R. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3), 1992.

[34] 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.

[35] T. M. Chan. A simple trapezoid sweep algorithm for reporting red/blue segment intersections. In *Proc. of 6th Canadian Conference on Computational Geometry*, 1994.

[36] B. Chazelle. Triangulating a simple polygon in linear time. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, 1990.

[37] B. Chazelle and H. Edelsbrunner. An optimal algorithm for intersecting line segments in the plane. *Journal of the ACM*, 39:1–54, 1992.

[38] B. Chazelle, H. Edelsbrunner, L. J. Guibas, and M. Sharir. Algorithms for bichromatic line-segment problems and polyhedral terrains. *Algorithmica*, 11:116–132, 1994.

[39] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[40] Y.-J. Chiang. *Dynamic and I/O-Efficient Algorithms for Computational Geometry and Graph Problems: Theoretical and Experimental Results*. PhD thesis, Brown University, August 1995.

[41] Y.-J. Chiang. Experiments on the practical I/O efficiency of geometric algorithms: Distribution sweep vs. plane sweep. In *Proc. Workshop on Algorithms and Data Structures, LNCS 955*, pages 346–357, 1995.

[42] 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.

[43] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of IEEE, Special Issue on Computational Geometry*, 80(9):362–381, 1992.

[44] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 383–391, 1996.

[45] A. Cockcroft. *Sun Performance and Tuning. SPARC & Solaris*. Sun Microsystems Inc., 1995.

[46] R. Cole and U. Vishkin. Deterministic coin tossing with applications to optimal list-ranking. *Information and Control*, 70(1):32–53, 1986.

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

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

[49] 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.

[50] T. H. Cormen and L. F. Wisniewski. Asymptotically tight bounds for performing BMMC permutations on parallel disk systems. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 130–139, 1993.

[51] D. Cormer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[52] R. F. Cromp. An intellegent information fusion system for handling the archiving and querying of terabyte-sized spatial databases. In *S. R. Tate ed., Report on the Workshop on Data and Image Compression Needs and Uses in the Scientific Community, CESDIS Technical Report Series, TR–93–99*, pages 75–84, 1993.

[53] H. Edelsbrunner. A new approach to rectangle intersections, part I. *Int. J. Computer Mathematics*, 13:209–219, 1983.

[54] H. Edelsbrunner. A new approach to rectangle intersections, part II. *Int. J. Computer Mathematics*, 13:221–229, 1983.

[55] H. Edelsbrunner and M. Overmars. Batched dynamic solutions to decomposable searching problems. *Journal of Algorithms*, 6:515–542, 1985.

[56] P. Ferragina and R. Grossi. A fully-dynamic data structure for external substring search. In *Proc. ACM Symp. on Theory of Computation*, pages 693–702, 1995.

[57] P. Ferragina and R. Grossi. Fast string searching in secondary storage: Theoretical developments and experimental results. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 373–382, 1996.

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

[59] R. W. Floyd. Permuting information in idealized two-level storage. In *Complexity of Computer Calculations*, pages 105–109, 1972. R. Miller and J. Thatcher, Eds. Plenum, New York.

[60] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, 1990.

[61] A. Fournier and D. Y. Montuno. Triangulating simple polygons and equivalent problems. *ACM Trans. on Graphics*, 3(2):153–174, 1984.

[62] 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.

[63] G. R. Ganger, B. L. Worthington, R. Y. Hou, and Y. N. Patt. Disk arrays. high-performance, high-reliability storage subsystems. *IEEE Computer*, 27(3):30–46, 1994.

[64] J. Gergov and C. Meinel. Frontiers of feasible and probabilistic feasible boolean manipulation with branching programs. In *Symposium on Theoretical Aspects of Computer Science, LNCS 665*, 1993.

[65] D. Gifford and A. Spector. The TWA reservation system. *Communications of the ACM*, 27:650–665, 1984.

[66] J. Gil and A. Itai. Packing trees. In *Proc. Annual European Symposium on Algorithms, LNCS 979*, pages 113–127, 1995.

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

[68] O. Günther. The design of the cell tree: An object-oriented index structure for geometric databases. In *Proc. of the fifth Int. Conf. on Data Engineering*, pages 598–605, 1989.

[69] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM Conf. on Management of Data*, pages 47–57, 1985.

[70] L. M. Haas and W. F. Cody. Exploiting extensible dbms in integrated geographic information systems. In *Proc. of Advances in Spatial Databases, LNCS 525*, 1991.

[71] A. Henrich, H.-W. Six, and P. Widmayer. Paging binary trees with external balancing. In *Proc. Graph-Theoretic Concepts in Computer Science, LNCS 411*, pages 260–276, 1989.

[72] J. W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proc. ACM Symp. on Theory of Computation*, pages 326–333, 1981.

[73] S. Huddleston and K. Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982.

[74] C. Icking, R. Klein, and T. Ottmann. Priority search trees in secondary memory. In *Proc. Graph-Theoretic Concepts in Computer Science, LNCS 314*, pages 84–93, 1987.

[75] B. Jiang. Traversing graphs in a paging environment BFS or DFS? *Information Processing Letters*, 37:143–147, 1991.

[76] B. Jiang. I/O and CPU-optimal recorgnition of strongly connected components. *Information Processing Letters*, 45:111–115, 1993.

[77] B. H. H. Juurlink and H. A. G. Wijshoff. The parallel hierarchical memory model. In *Proc. Scandinavian Workshop on Algorithms Theory, LNCS 824*, pages 240–251, 1993.

[78] P. C. Kanellakis, G. Kuper, and P. Revesz. Constraint query languages. In *Proc. ACM Symp. Principles of Database Systems*, pages 299–313, 1990.

[79] P. C. Kanellakis, S. Ramaswamy, D. E. Vengroff, and J. S. Vitter. Indexing for data models with constraints and classes. In *Proc. ACM Symp. Principles of Database Systems*, 1993. Invited to special issue of JCSS on Principles of Database Systems (to appear). A complete version appears as technical report 90-31, Brown University.

[80] N. Klarlund and T. Rauhe. BDD algorithms and cache misses. Technical Report RS-96-26, BRICS, University of Aarhus, 1996.

[81] M. Knudsen and K. Larsen. I/O-complexity of comparison and permutation problems. Master's thesis, Aarhus University, November 1992.

[82] D. Knuth. *The Art of Computer Programming, Vol. 3 Sorting and Searching.* Addison-Wesley, 1973.

[83] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. IEEE Symp. on Parallel and Distributed Processing*, 1996.

[84] R. Laurini and A. D. Thompson. *Fundamentals of Spatial Information Systems.* A.P.I.C. Series, Academic Press, New York, NY, 1992.

[85] C. E. Leiserson, S. Rao, and S. Toledo. Efficient out-of-core algorithms for linear relaxation using blocking covers. In *Proc. IEEE Symp. on Foundations of Comp. Sci.*, pages 704–713, 1993.

[86] D. Lomet and B. Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, 1990.

[87] H. G. Mairson and J. Stolfi. Reporting and counting intersections between two sets of line segments. In *R. Earnshaw (ed.), Theoretical Foundation of Computer Graphics and CAD, NATO ASI Series, Vol. F40*, pages 307–326, 1988.

[88] S. Malik, A. R. Wang, R. K. Brayton, and A. Sangiovanni-Vincentelli. Logic verification using binary decision diagrams in a logic synthesis environment. In *Proc. IEEE International Conference on CAD*, 1988.

[89] E. McCreight. Priority search trees. *SIAM Journal of Computing*, 14(2):257–276, 1985.

[90] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry.* Springer-Verlag, EATCS Monographs on Theoretical Computer Science, 1984.

[91] J. Misra and D. Gries. Finding repeated elements. *Science of Computer Programming*, 2:143–152, 1982.

[92] K. Mulmuley. *Computational Geometry. An introduction through randomized algorithms.* Prentice-Hall, 1994.

[93] J. I. Munro and V. Raman. Sorting multisets and vectors in-place. In *Proc. Workshop on Algorithms and Data Structures, LNCS 519*, pages 473–479, 1991.

[94] J. I. Munro and P. M. Spira. Sorting and searching in multisets. *SIAM Journal of Computing*, 5(1):1–8, 1976.

[95] J. Nievergelt, H. Hinterberger, and K. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):257–276, 1984.

[96] J. Nievergelt and E. M. Reingold. Binary search tree of bounded balance. *SIAM Journal of Computing*, 2(1), 1973.

[97] M. H. Nodine, M. T. Goodrich, and J. S. Vitter. Blocking for external graph searching. *Algorithmica*, 16(2):181–214, 1996.

[98] M. H. Nodine and J. S. Vitter. Large-scale sorting in parallel memories. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 29–39, 1991.

[99] 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.

[100] 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.

[101] H. Ochi, N. Ishiura, and S. Yajima. Breadth-first manipulation of sbdd of boolean functions for vector processing. In *Proc. ACM/IEEE Design Automation Conference*, 1991.

[102] H. Ochi, K. Yasuoka, and S. Yajima. Breadth-first manipulation of very large binary-decision diagrams. In *Proc. IEEE International Conference on CAD*, 1993.

[103] J. Orenstein. Spatial query processing in an object-oriented database system. In *Proc. ACM Conf. on Management of Data*, pages 326–336, 1986.

[104] M. Overmars, M. Smid, M. de Berg, and M. van Kreveld. Maintaining range trees in secundary memory. Part I: Partitions. *Acta Informatica*, 27:423–452, 1990.

[105] M. H. Overmars. *The Design of Dynamic Data Structures*. Springer-Verlag, LNCS 156, 1983.

[106] L. Palazzi and J. Snoeyink. Counting and reporting red/blue segment intersections. In *Proc. Workshop on Algorithms and Data Structures, LNCS 709*, pages 530–540, 1993.

[107] Y. N. Patt. The I/O subsystem — a candidate for improvement. *Guest Editor's Introduction in IEEE Computer*, 27(3):15–16, 1994.

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

[109] S. Ramaswamy and P. Kanellakis. OOBD indexing by class division. In *A.P.I.C. Series, Academic Press, New York*, 1995.

[110] S. Ramaswamy and S. Subramanian. Path caching: A technique for optimal external searching. In *Proc. ACM Symp. Principles of Database Systems*, 1994.

[111] J. Robinson. The K-D-B tree: A search structure for large multidimensional dynamic indexes. In *Proc. ACM Conf. on Management of Data*, pages 10–18, 1984.

[112] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *Proc. IEEE International Conference on CAD*, 1993.

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

[114] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison Wesley, MA, 1989.

[115] H. Samet. *The Design and Analyses of Spatial Data Structures*. Addison Wesley, MA, 1989.

[116] J. E. Savage. Space-time tradeoffs in memory hierarchies. Technical Report CS-93-08, Brown University, 1993.

[117] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multidimensional objects. In *Proc. IEEE International Conf. on Very Large Databases*, 1987.

[118] D. Sieling and I. Wegener. Reduction of obdds in linear time. *Information Processing Letters*, 48, 1993.

[119] M. Smid. *Dynamic Data Structures on Multiple Storage Media*. PhD thesis, University of Amsterdam, 1989.

[120] M. Smid and M. Overmars. Maintaining range trees in secundary memory. Part II: Lower bounds. *Acta Informatica*, 27:453–480, 1990.

[121] 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.

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

[123] V. K. Vaishnavi and D. Wood. Rectilinear line segment intersection, layered segment trees, and dynamization. *Journal of Algorithms*, 3:160–176, 1982.

[124] M. van Kreveld. Geographic information systems. Utrecht University, INF/DOC–95–01, 1995.

[125] J. van Leeuwen. *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*. Elsevier, 1990.

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

[127] D. E. Vengroff. Private communication, 1995.

[128] D. E. Vengroff. Private communication, 1996.

[129] D. E. Vengroff and J. S. Vitter. I/O-efficient scientific computation using TPIE. In *Proc. IEEE Symp. on Parallel and Distributed Computing*, 1995. Appears also as Duke University Dept. of Computer Science technical report CS-1995-18.

[130] D. E. Vengroff and J. S. Vitter. Efficient 3-d range searching in external memory. In *Proc. ACM Symp. on Theory of Computation*, pages 192–201, 1996.

[131] J. S. Vitter. Efficient memory access in large-scale computation (invited paper). In *Symposium on Theoretical Aspects of Computer Science, LNCS 480*, pages 26–41, 1991.

[132] J. S. Vitter and M. H. Nodine. Large-scale sorting in uniform memory hierarchies. *Journal of Parallel and Distributed Computing*, 17:107–114, 1993.

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

[134] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory, II: Hierarchical multilevel memories. *Algorithmica*, 12(2–3):148–169, 1994.

[135] J. D. Watson. The human genome projekt: Past, present, and future. *Science*, 248:44–49, 1990.

[136] D. Willard and G. Lueker. Adding range restriction capability to dynamic data structures. *Journal of the ACM*, 32(3):597–617, 1985.

[137] B. Zhu. Further computational geometry in secondary memory. In *Proc. Int. Symp. on Algorithms and Computation*, pages 514–522, 1994.

Recent Publications in the BRICS Dissertation Series

DS-96-3 **Lars Arge.** ***Efficient External-Memory Data Structures and Applications.*** **August 1996. Ph.D. thesis. xii+169 pp.**

DS-96-2 **Allan Cheng.** ***Reasoning About Concurrent Computational Systems.*** **August 1996. Ph.D. thesis. xiv+229 pp.**

DS-96-1 **Urban Engberg.** ***Reasoning in the Temporal Logic of Actions — The design and implementation of an interactive computer system.*** **August 1996. Ph.D. thesis. xvi+222 pp.**