Improved Algorithms and Data Structures for Solving Graph Problems in External Memory *

Vijay Kumar
Eric J. Schwabe

Department of EECS
Northwestern University
Evanston, IL 60208

Phone: (847) 467–2298
Fax: (847) 467-4144
Email: {vijay, schwabe}@eecs.nwu.edu

Keywords: Input/output efficiency, external-memory algorithms, graph algorithms, data structures.

*This research was supported in part by the National Science Foundation under grant CCR–9309111. A preliminary version of this work will appear in the Proceedings of the Eighth IEEE Symposium on Parallel and Distributed Processing, October 1996.

Abstract

Recently, the study of I/O-efficient algorithms has moved beyond fundamental problems of sorting and permuting and into wider areas such as computational geometry and graph algorithms. With this expansion has come a need for new algorithmic techniques and data structures. In this paper, we present I/O-efficient analogues of well-known data structures that we show to be useful for obtaining simpler and improved algorithms for several graph problems. Our results include improved algorithms for minimum spanning trees, breadth-first and depth-first search, and single-source shortest paths. The descriptions of these algorithms are greatly simplified by their use of well-defined I/O-efficient data structures with good amortised performance bounds. We expect that I/O-efficient data structures such as these will be a useful tool for the design of I/O-efficient algorithms.

1 Introduction

The design of I/O-efficient algorithms has received increasingly greater attention in recent years. This has been because of a disparity in the growth rates of CPU speeds and disk transfer rates: improvements in CPU speeds have consistently outpaced the rate at which the speed of communication between main and external memory has grown. According to one recent estimate, advances in technology have yielded an annual growth of 40 to 60 percent in CPU speeds, while disk transfer rates have been increasing only at about 7 to 10 percent annually [1]. As it is, communication between internal and external memory has for some time been a bottleneck in many large-scale computations. The increasing disparity in CPU speeds and disk transfer rates implies that the significance of this bottleneck continues to grow. This has made it difficult to take full advantage of the power of the CPU in many large-scale computations. The bottleneck is all the more significant in parallel computing and multiprocessing. Consequently, it has become important to design algorithms that minimise the transfer of data between internal and external memory.

Problems that are too large to be solved in internal memory are encountered in many areas, ranging from numerical computing to computational geometry. A lot of work has been done in many of these areas to design I/O-efficient algorithms. In graph algorithms, some instances of such problems are large circuit layout problems and modeling of large phone, communication or other networks.

In the area of graph algorithms, most of the effort has gone into devising I/O-efficient algorithms for individual problems. The common approach is to re-design the algorithm on one hand and the arrangement of data on the other, to extract such I/O-efficiency as the nature of the problem would allow. The design of data structures that facilitate the task has only recently been widely taken up [2, 3]. We feel that the design of general-purpose I/O-efficient data structures is very important. The I/O efficiency of a range of algorithms could be improved simply by replacing the data structures employed by their I/O-efficient versions, if such versions were to be available. Algorithms that use such general-purpose data structures are also likely to be simpler to describe.

1.1 Model of Computation

The computational model that we employ was introduced by Aggarwal and Vitter [4]. It consists of a single processor with a small local memory connected to a large external memory. We use the following parameters:

N = the size of the problem in external memory,

M = the size of the internal memory, and

B = the number of items in one block of data.

An extension of the model [5] incorporates another parameter:

D = the number of disks in the memory system.

It is assumed that $M < N$ and $1 \ll DB < M/2$.

The basic I/O operation [5] consists of the reading of one block of data from each of the D disks (or writing of one block of data to each) — a transfer of a total of DB items between main memory and external memory. The address space of the external memory is *striped* block-wise across the D disks: the rth block overall of the N/B blocks of data is the $(\lfloor r/D \rfloor)$th block on disk number $r \bmod D$. How many such basic I/O operations an algorithm uses would be our measure of its performance.

We will work with the single-disk model ($D = 1$). The extendibility of our results to the D-disk model is discussed in Section 1.3.

After Chiang et al. [6], we define the useful quantities $scan(x) = \frac{x}{DB}$ and $sort(x) = \frac{x}{DB} \log_{\frac{M}{B}} \frac{x}{B}$. Up to constant factors, $scan(x)$ is the number of I/Os needed to read x consecutive items stored on the D disks (the items being stored in the striped fashion described above), and $sort(x)$ is number of I/Os required to sort x items stored consecutively on the D disks, using a comparison-based algorithm [7, 8, 5]. In the single-disk model, these expressions reduce to $scan(x) = \frac{x}{B}$ and $sort(x) = \frac{x}{B} \log_{\frac{M}{B}} \frac{x}{B}$.

In addition, in the context of a graph problem, let V denote the number of vertices in the graph being considered and let E be the number of edges. A graph will be represented as a sequence of V adjacency lists, where each list consists of a fixed amount of information

about a vertex followed by a list of its incident edges. Assuming such a representation, we have $N = \Theta(V + E)$.

1.2 Previous Results

The problem of designing I/O-efficient algorithms was first taken up by Aggarwal and Vitter [4], who gave several algorithms for basic problems such as sorting, permuting, and matrix operations. Their results assumed a weaker computational model in which the external memory consists of a single large disk containing all N items, and the basic I/O step can transfer *any* k blocks of B items each between main memory and external memory. Optimal sorting algorithms were obtained for the more general D-disk model by Vitter and Shriver [5] (using a randomised algorithm), and by Nodine and Vitter [7] (using a deterministic algorithm). Nodine and Vitter [8] also extended their sorting results to apply to a variety of memory hierarchies. However, until recently, research in I/O-efficient algorithms centered on the fundamental problems of sorting, permuting, and the like.

Goodrich et al. [9] were the first to develop external-memory techniques that applied to a class of algorithms — algorithms for computational geometric problems. A lot of effort has since gone into the design of I/O-efficient computational geometry algorithms [9, 10]. Later, Chiang et al. [6] focused on the design of external-memory graph algorithms, producing algorithms for a number of problems including list ranking, expression tree evaluation, PRAM simulation, 3-colouring of cycles, and depth-first search.

In most of this work, the data structures used by the algorithms were motivated by the particular problems that were being considered. Arge [2] improved several of the results in these earlier papers by introducing the I/O-efficient *buffer tree*: the first I/O-efficient data structure to incorporate an amortised analysis for batched operations. It led to simple algorithms for sorting and some graph problems and to the generalisation of some of the results of Chiang et al.[6]. Arge also designed a similar data structure called the *segment tree* which he used to develop external-memory algorithms for many problems in computational geometry [10].

1.3 Our Results

We strive to develop new general-purpose I/O-efficient data structures, and illustrate further how they can assist in the design of I/O-efficient algorithms. The primary motivation here is to solve problems on large graphs, but we expect that our I/O-efficient versions will find applications in other areas and be useful in the design of a variety of I/O-efficient algorithms.

The two data structures we present are I/O-efficient versions of a *heap* and a *tournament tree*. We present an external-memory generalisation of a binary heap that can achieve amortised performance of $O(\frac{1}{B}\log_{\frac{M}{B}}\frac{N}{B})$ I/Os per operation (insert, delete, or deletemin) over a sequence of operations on a heap containing at most N elements. This data structure resembles Arge's buffer-tree [2] in its use of a buffering technique for batched maintenance of the data structure, and yields the same performance bounds. Arge's buffer-tree is an external-memory generalisation of a balanced search tree, just as our data structure is derived from the binary heap data structure. We avoid performing complicated tree-balancing operations to achieve some simplicity of description for our data structure. Our I/O-efficient heap immediately yields a heapsort with optimal I/O performance, illustrating how I/O-efficient data structures can simplify the task of describing algorithms.

Our I/O-efficient tournament tree requires $O(\frac{1}{B}\log_2\frac{N}{B})$ I/Os per operation (delete, deletemin, or update) amortised over a sequence of operations on a tournament tree of N elements. In addition to general deletions and deletemins, the tournament tree also supports an *update* operation that, given an element and a new key value, changes the element's key value if and only if the current key value is greater than the new one.

These data structures, together with some new algorithmic techniques, lead to algorithms with improved I/O performance for several fundamental graph problems. Our results include:

- New algorithms for finding minimum spanning trees with nearly-optimal I/O performance — $O(\log B \cdot sort(E) + \log V \cdot scan(E))$ I/Os; previously the best known algorithm for this problem [6] required $O(\min\{sort(V^2),\ \log\frac{V}{M}\cdot sort(E)\})$.
- New algorithms for breadth-first search and single-source shortest paths that require $O(V + \frac{E}{B}\log_2\frac{E}{B})$ I/Os.
- A new algorithm for depth-first search that requires $O(V\log_2 V + \frac{E}{B}\log_2\frac{E}{B})$ I/Os.

- A generalised deterministic algorithm for list ranking that requires $O(sort(N))$ I/Os.

Throughout the paper, we will work in the single-disk external-memory model — that is, for the case when $D = 1$. All of our proofs generalise in a straightforward way to external memories consisting of more than one disk if the number of disks is no more than $O((\frac{M}{B})^{\alpha})$ for some $\alpha < 1$, using a technique known as *striping*. Striping treats the D disks as a single disk with a block size of DB, and the basic I/O operation is the reading or writing of one of these larger blocks of data. We believe that the results are extendible to the more general case too, but we have not proved this.

The paper is organised as follows. In Section 2, we present our I/O-efficient data structures and establish amortised bounds on their performance. Section 3 contains several improved algorithms for graph problems that use our I/O-efficient data structures. Finally, in Section 4 we discuss our conclusions and mention some open problems.

2 Data Structures for External-Memory Algorithms

In this section, we present the I/O-efficient versions of two common data structures.

2.1 An I/O-Efficient Heap

2.1.1 Description of the Data Structure

A *priority queue* is a data structure encountered very frequently in algorithms of all kinds. It is used to maintain a set S of elements, where each element has a number associated with it, called the *key* of that element. The following operations can be performed:

1. *insert(S,x)*: inserts element x into S.
2. *minimum(S)*: Returns the element with the smallest key among all the elements of S.
3. *deletemin(S)*: Returns the element in S with the smallest key and removes it from S.

One of the most common implementations of a priority queue is a data structure called *heap*. A heap is a complete binary tree of elements with a certain number of its rightmost leaves removed. The key of any non-leaf element is smaller than the key of either of its children.

Numerous variants of the basic heap design are encountered in various algorithms, but they are not I/O-efficient and not conducive to use in external-memory algorithms. We present an I/O-efficient data structure called the *I/O-efficient heap*, the design of which is described below.

An *I/O-efficient heap* is a tree with the following properties:

- Each node contains a *list* of elements in sorted order. The number of elements in the list is between $\sqrt{MB}/2$ and $2\sqrt{MB}$.
- Each non-leaf node has $\sqrt{M/B}$ children.
- Each node has an unordered *buffer* that can contain up to M elements.
- Any element contained in a node's list is smaller than any element contained in its buffer or in any of its descendant nodes.

The only exception to these rules could be a heap with just one node, or the nodes at the level above the leaves, which may not always have $\sqrt{M/B}$ children.

The *cardinality* of a node in the heap is the number of elements contained in the sub-heap rooted at it.

2.1.2 Maintenance of the data structure

To achieve I/O efficiency, we develop a scheme of lazy updates for this data structure. It is to make lazy updates possible that we allow a node to hold a variable number of elements. The objective is to perform maintenance on this data structure in a *batched* fashion. To allow inserted elements to move down the heap in a batched fashion we provide for a *buffer* at every node with capacity M. For a node v, elements that are to be sent down to the

descendants of v are accumulated in its buffer. When the buffer becomes full, its contents are transferred to the children of v. These features allow us to avoid having to undertake expensive I/O to update a node and its children every time an element is to move up from or down to it. Such updates can now be batched together to significantly reduce the I/O cost per update.

In addition to *insert* and *deletemin* described above, the operation *delete(S,x)*, which removes a specified element x from the heap S, is supported. The I/O-efficient implementation of these operations is described below.

- *insert*: the *insert* operation adds a new element to the root node. If the element is small enough to be held in the root's list it is inserted there. This happens when the element to be inserted is smaller than the largest element in the root's list. Otherwise, it is added to the root's buffer and is expected to trickle down to some node where its presence does not violate the heap order. When thus moving down from a node v, an element goes to the root node of the sub-heap that has the smallest cardinality among all the sub-heaps rooted at children of v. This helps preserve the height balance of the tree structure.

- *deletemin*: the *deletemin* operation removes and returns the smallest element contained in the root node.

- *delete*: the *delete* operation creates a duplicate of the element to be deleted and inserts it in the heap. Whenever two copies of the same element encounter each other in the heap, they are annihilated. That is to say, the two copies can not exist in the same node's list. Annihilation is assured because to be returned by *deletemin*, one of the two copies must become the smallest element in the heap. But in that case it would have reached the root's list, as would the other copy because it has the same key. This will lead to annihilation. Although this implementation of *delete* implies that only distinct elements can be kept in the heap, this restriction can be circumvented if desired by adding another field to the elements that can be used to make elements distinct even when they have the same key value. This extra field can be used as a secondary key in comparisons so that duplicate elements inserted by the *delete* operation are not kept apart by other elements which have the same key.

In order to implement the heap operations, the following primitives are used:

- *emptylist*: called when a node's list contains more than $2\sqrt{MB}$ elements. It moves excess elements to the buffer.
- *fillup*: called to replenish a node whose list contains less than $\sqrt{MB}/2$ elements. It moves elements up from the node's children into the node's list.
- *emptybuffer*: called when a buffer overflows. Moves the contents of the buffer down to the child nodes.

2.1.3 Implementation of the primitives

The primitives are implemented as follows:

emptylist is straightforward: it takes all except the smallest $\sqrt{MB}$ elements in the list and puts them in the node's buffer.

emptybuffer is used to empty a node's buffer by distributing its contents among its child nodes. If the buffer contains x elements, up to $x/\sqrt{MB}$ of the node's children are potential recipients of the buffer's contents. The distribution is done as follows: A *set of recipients* is maintained. Initially, the set of recipients contains the child node with the smallest cardinality. At any stage in the process, all the elements in the set of recipients have equal cardinality. A step in the distribution process consists of setting aside B elements from the buffer being emptied for each of the recipients, whose cardinality is then assumed to have increased by B. If the cardinality of the nodes in the set of recipients becomes equal to the cardinality of any non-recipient child node, that node is added to the set of recipients if doing so does not make the number of recipients more than $x/\sqrt{MB}$. In the end, the elements set aside for a particular recipient are actually given to it. The recipient's list is read into memory, and if there are any elements among its share that belong inside the list in sorted order, they are added to the list, while the rest are output to its buffer. If either the list or the buffer overflows, it is in turn emptied.

In case of a leaf, *emptybuffer* creates child nodes and gives $\sqrt{MB}$ elements to each. If a node has fewer than $\sqrt{M/B}$ children, *emptybuffer* first creates enough children to raise

the number of children to $\sqrt{M/B}$, and then distributes the remaining contents of the buffer among the children as described above.

In *fillup*, first we empty the node's buffer, and then extract enough elements from the child nodes to raise the number of elements in the list to $\sqrt{MB}$. Treating the children's lists as a collection of lists we have to merge, we obtain a merged list of the required size and append it to the list to be refilled. As efficient merging schemes are known (see for instance [4]), we omit the details.

2.1.4 The I/O Complexity of Heap Operations

Although the heap can be unbalanced by deletions, insertions serve to balance it, as new elements are added to the smaller sub-heaps. So when the heap grows bigger than it has ever been upto that point, it is a balanced heap. It is easy to see that

Lemma 1 *The height of a heap that never contains more than N elements is $O(\log_{\frac{M}{B}} \frac{N}{B})$.*

Proof: Follows from the fact that $\log_{\frac{M}{B}} \frac{N}{\sqrt{MB}}$ is $O(\log_{\frac{M}{B}} \frac{N}{B})$. ■

Let l denote the height of such a heap, with the root at level l and the leaves at level 0.

Our objective is to obtain an amortised bound on the total number of I/Os in terms of the number of heap operations performed. The lower bound of $\Omega(\frac{N}{B} \log_{\frac{M}{B}} \frac{N}{B})$ on sorting [4] together with the bound on heap height implies that a cost of $O(l/B)$ per heap operation would be optimal.

The primitives *emptybuffer*, *emptylist*, and *fillup* perform all the I/O operations, each one moving a collection of elements one level down or up the heap. These functions are I/O-efficient — that is, when they move x elements up or down one level, they use x/B I/Os to do so. We will account for the I/O cost of these primitives by charging the various heap operations.

When some elements are moved from a node's list to its buffer, we will ignore the I/O cost. This cost is no more than that of later moving those elements out of the buffer, which

is the next operation to be performed on these elements. If we can bound the latter cost, the same bound will apply to the former. As there is no movement of elements from a buffer to the corresponding list, we are effectively considering all movement of elements to be between two adjacent levels in the heap.

In the following, we derive amortised cost bounds for the various heap operations. First we take up *deletemin*.

Lemma 2 *The amortised cost of a deletemin is $O(l/B)$ I/Os.*

Proof: Let every *deletemin* 'pay' the amount l/B in dollars to the root node. As the root must lose $\frac{\sqrt{MB}}{2}$ elements before a fillup is needed, it will have amassed $\frac{l}{2}\sqrt{\frac{M}{B}}$ dollars by the time a *fillup* takes place.

Let us inductively show (using induction on distance from the root) that when a node at level i needs a *fillup*, it has $\frac{i}{2}\sqrt{\frac{M}{B}}$ dollars to pay for it. It is true of the root, as we saw. Let it be true of a node at level $i+1$. Thus when a node at level $i+1$ needs a *fillup*, it possesses $\frac{i+1}{2}\sqrt{\frac{M}{B}}$ dollars. The I/O complexity of extracting $\frac{1}{2}\sqrt{MB}$ elements from the children is $\frac{1}{2}\sqrt{\frac{M}{B}}$, if the operation is to take $1/B$ I/Os per element moved. Let us use $\frac{1}{2}\sqrt{\frac{M}{B}}$ dollars to pay for that. The remaining $\frac{i}{2}\sqrt{\frac{M}{B}}$ dollars will be used as follows: if x elements have been extracted from a child, let $\frac{xi}{B}$ dollars be paid to that child. As $\frac{1}{2}\sqrt{MB}$ elements have been extracted in all, this will cost $\frac{i}{2}\sqrt{\frac{M}{B}}$ dollars.

Now a node at level i must lose $\frac{1}{2}\sqrt{MB}$ elements to its parent in order to need a *fillup*, and for that the parent will have paid it $\frac{i}{2}\sqrt{\frac{M}{B}}$ dollars, which completes the induction. Therefore the number of dollars paid by *deletemin* is enough to cover the resulting I/O cost. Thus the amortised cost of a *deletemin* is $O(l/B)$. ∎

Now let us look at *inserts*. A *delete* is analogous to an *insert* (it is performed by inserting a duplicate of the element to be deleted), and will not be treated separately.

Lemma 3 *The amortised cost of an* insert *or a* delete *is $O(\frac{l}{B})$ I/Os.*

Proof: Let every *insert* pay $\frac{l}{B}$ roubles to the root. We will use these roubles to pay for all the *emptybuffer* operations (remember that we have decided to ignore the cost of all the

emptylist operations). We will inductively show that when a node needs its buffer emptied, it has enough roubles to pay for it.

Let the root buffer contain x elements when it needs to be emptied. At least x insertions have been made since the last *emptybuffer*. So the root has acquired $\frac{lx}{B}$ roubles through these insertions. We will inductively show that a node at level i has amassed $\frac{ix}{B}$ roubles since the last *emptybuffer*, if its buffer now contains x elements.

Assume it to be true of nodes at level $i+1$. So when such a node with x elements in its buffer needs an *emptybuffer* operation, it has at least $\frac{(i+1)x}{B}$ roubles.

The *emptybuffer* operation requires $O(\frac{x}{B})$ I/Os. Let $\frac{x}{B}$ roubles be used to pay for that. Spend the remaining $\frac{ix}{B}$ roubles as follows: If a child node receives j elements during the process, let it be paid $\frac{ij}{B}$ roubles.

So if an element at level has y elements its buffer, it has since its last *emptybuffer* received at least y elements from its parent, which means it has received $\frac{iy}{B}$ roubles as well. This completes the induction. As the number of roubles spent by the *inserts* is sufficient to cover the I/O cost, the amortised I/O cost of *inserts* is $O(\frac{l}{B})$ per operation. ■

We have shown that we can pay for all the I/Os using the credits that we obtained by charging the *insert* [and *delete*] and *deletemin* operations $\frac{l}{B}$ per operation. That means the amortised cost of each of these operations is $O(\frac{l}{B})$ I/Os. Using the bound on l, the number of levels, we obtain the following theorem:

Theorem 1 *On an I/O-efficient heap with at most N elements, a sequence of k operations (*insert*, *delete*, and* deletemin*) requires at most $O(\frac{k}{B}\log_{\frac{M}{B}}\frac{N}{B})$ I/Os.* ■

These performance bounds for the I/O-efficient heap lead immediately to an optimal heap sorting algorithm for N elements, which consists of just N insertions followed by N deletemins:

Corollary 1 *On an I/O-efficient heap, Heapsort sorts N elements using $O(sort(N))$ I/Os.* ■

(Arge [2] achieved the same bound for sorting with a different data structure.)

2.2 An I/O-Efficient Tournament Tree

Next we describe a data structure called the *tournament tree*, which is similar to the heap in some ways, but holds additional information.

The tournament tree is a useful data structure. It offers some advantages over a heap as there is a natural mapping of the numbers $1, \ldots, N$ to the N items in a tournament tree, and this mapping is implicit in the way a tournament tree is organised. The additional information is often useful.

Imagine a knock-out tournament to be played among $N = 2^k$ players. It would involve k rounds and $2 \cdot N - 1$ matches. In round i, 2^{k-i} matches would be played, each resulting in the elimination of the player that loses. The tournament can be represented using a binary tree, where each leaf represents one player, and each internal node holds the winner of a match played between its two child nodes. The root contains the winner of the tournament. The resulting data structure is called the tournament tree. The completed chart of the Wimbledon draw (with the winner of each match filled in), for instance, is a tournament tree with 128 leaves. Tournament trees are useful in many comparison-based problems, where the elements are the 'players', and contests represent comparisons. For example, a tournament tree can be used to implement a priority queue.

An I/O-efficient version of the tournament tree is presented below. The I/O-efficient version differs from the basic tournament tree described above in the following respects: just as in the I/O-efficient heap, each node contains a number of elements, and has an associated buffer; and each element is contained in just one node. Later, we will demonstrate applications to the single-source shortest paths problem and breadth-first and depth-first search that illustrate the usefulness of this data structure.

2.2.1 Description of the Data Structure

The I/O-efficient tournament tree houses N elements, each of which is a pair of the form (x, y). The x values, ranging from 1 to N, identify the elements. The structure is organised according to comparisons on the y values of the elements. The y value of an element will also be known as its *key*; all elements initially have a key value of infinity. More formally:

An *I/O-efficient tournament tree* is a binary tree with the following properties:

- It is a complete binary tree, except for the fact that some k rightmost leaves may be missing.
- It has N/M leaves, where N is the number of *elements* in the tree. Each element is a pair of values, called the ID and the key of the element, respectively. Each element has a unique ID, and the IDs range from 1 to N. The element with ID x will be referred to as element x. Out of two elements, the one with the smaller key will be referred to as the smaller element.
- There is a mapping of element IDs to leaves. Elements with IDs in the range $(i-1)M+1$ to iM map to the i^{th} leaf. An element is either contained in the leaf that it maps to, or in some ancestor thereof.
- A node may contain between $M/2$ and M elements. If a node v contains x elements, they are the smallest x out of all the elements that map to leaves that are descendants of v.
- Each non-leaf node has an associated *buffer* of size M.

The structure supports the following three operations:

- *deletemin* returns the element with the smallest key in the tree and removes it from the tree.
- $delete(x)$ changes the key value of element x to infinity, and removes (x, y) from internal nodes on the path from the root to x's leaf, where y is the old key value of x.
- $update(x, y)$ changes the key value of element x to y if and only if y is smaller than the key value of x.

The tree is balanced and consists of $\log_2 \frac{N}{M}$ levels. Every node at any level except the lowest has two children. The x value of any element in the left child is smaller than that of any element in the right child.

2.2.2 Maintenance of the data structure

Initially, all elements have a key value of infinity. When an operation is performed, a *delete* or *update signal* is generated and inserted in the root. When a signal is inserted in a node, it may affect some element held in that node. The key of that element may have to be changed, and the signal may have to be propagated down to the appropriate child node. This is done in the following (lazy) way.

A *delete* or an *update* signal propagates down till it meets the targeted element in some node. In case of a *delete*, the element is removed, the signal is converted into an *update* signal with a key value of infinity and it continues to propagate down from that node. In the case of an *update*, if the new value is larger than the old value, the signal is discarded. If the new value is smaller than the old value, it replaces the old value. The signal does not propagate any further down from the node containing the targeted element. If an *update* signal with a key value of infinity reaches a leaf and the targeted element is not present in the leaf, it is inserted in the leaf with a key value of infinity.

When a signal is inserted into the root (which is held in internal memory), the appropriate changes are made to elements held in the root and the signal is added to the root buffer. When a buffer contains M signals we *empty* it as follows: The signals in the buffer are distributed between the two child nodes, each signal being sent to that child node which is to contain the element targeted by that signal. Then we apply to each child node its share of the signals. For this, the child nodes have to be read into memory one after the other. The signals that have to propagate further down the tree are then added to the respective child nodes' buffers. It is easy to see that

Observation 1 *An* empty *operation requires* $O(\frac{M}{B})$ *I/Os.* ■

When the number of undeleted elements in a node is reduced to $\frac{M}{2}$, we have to replenish that node. The following routine is called when the number of elements in a node is reduced to less than $\frac{M}{2}$:

fillup(v) restores the strength of node v to M elements. It examines the child nodes of v to find the M elements with the smallest key values contained therein. Before this can be

done, however, the child nodes must be updated with all the signals in the parent's buffer. For this purpose, the buffer of v is first emptied. If during the *fillup* all the elements from a child node are used up, that node is filled up before the parent can be filled up. At the end we check if either child node is depleted to a strength of less than $\frac{M}{2}$ elements: if so, that node is in turn filled up. Not counting the cost of recursive calls to fill up child nodes,

Observation 2 *The cost of a* fillup *operation is* $O(\frac{M}{B})$ *I/Os.* ■

2.2.3 The I/O Complexity of Tournament Tree Operations

To begin with, since a tournament tree with N elements is a balanced binary tree with $O(\frac{N}{M})$ leaves, we have

Observation 3 *The height of a tournament tree containing* N *elements is* $O(\log_2 \frac{N}{M})$. ■

It may be noted that $\log_2 \frac{N}{M}$ is $O(\log_2 \frac{N}{B})$.

We note that the data structure contains elements and signals. We also make the following observations:

Observation 4 *Elements only move upwards in the tree, while signals only move downwards. All such movement is along the path between the root and the leaf to which the targeted element is mapped.* ■

Observation 5 *All the I/O done in the* empty *and* fillup *operations either moves signals down or moves elements up the tree.* ■

Observation 6 *An operation on the tournament tree generates no more than* $O(1)$ *signals.* ■

Recall that both *fillup* and *empty* cost $O(\frac{M}{B})$ I/Os each. We will now establish bounds on the amortised I/O cost of tournament tree operations and in the process account for the I/O cost of calls to *fillup* and *empty*.

Lemma 4 *A sequence of tournament tree operations containing k deletions on a tournament tree containing N elements requires $O(\frac{k}{B} \cdot \log_2 \frac{N}{B})$ I/Os for calls to* fillup.

Proof: A *fillup* operation is required as a result of depletion of a node. It is evident that this depletion is caused solely by deletions. Note that a deletion can potentially cause up to h nodes (one at each level in the tree) to lose an element each (either directly or when the node's parent needs to be filled up), where h is the height of the tree. So when a deletion causes a node to lose an element, allow that node to charge a cost of $O(\frac{1}{B})$ to that deletion. So overall a deletion can be charged up to $O(\frac{h}{B})$. As a node must lose $\frac{M}{2}$ elements to require a *fillup*, it will have accumulated $O(\frac{M}{B})$ credits by charging *delete* operations, which can then be used to pay for the $O(\frac{M}{B})$ cost of the fillup.

As the height h is known to be $O(\log_2 \frac{N}{B})$, the lemma follows. ■

What about the total cost of all the *empty* operations? These operations move signals down the tree. We will charge this cost to the various operations.

Lemma 5 *The calls to the subroutine empty made by a sequence of k tournament tree operations require no more than $O(\frac{k}{B} \cdot \log_2 \frac{N}{B})$ I/O steps.*

Proof: A call to *empty* can be made either when a buffer is full, or when a node requires a *fillup*. In the latter case, the $O(\frac{M}{B})$ cost of *empty* is included in the $O(\frac{M}{B})$ overall cost of a *fillup*. The $O(\frac{M}{B})$ cost in the former case can be looked upon as a cost of $O(\frac{1}{B})$ I/Os for each signal moved one level down. Let us charge this cost to the tournament tree operation that generated the signal. Obviously, each operation can be charged at most once by each of the h levels, so the cost charged to each operation is no more than $O(\frac{h}{B})$ I/Os. ■

The lemmas above directly yield

Theorem 2 *On an I/O-efficient tournament tree with N elements, a sequence of k operations (delete, deletemin, and update) requires at most $O(\frac{k}{B} \log_2 \frac{N}{B})$ I/Os.* ■

The ability of the I/O-efficient tournament tree to perform updates to the key values of entries without knowing their current values will be used in Section 3 to help design new I/O-efficient algorithms for breadth-first and depth-first search and single-source shortest paths.

3 Improved Algorithms for Graph Problems

Conventional graph algorithms often do not lend themselves to adaptation for the external-memory setting. Graph edges can be looked upon as 'connections' between 'items' that are not necessarily proximate. Following an edge to a neighbouring vertex therefore means using an I/O step with the objective of accessing a single item, an operation that is not I/O-efficient. One way to derive I/O-efficient algorithms is to construct them using two kinds of building blocks: I/O-efficient data structures, and techniques like sorting, list ranking and merging for which I/O-efficient algorithms are available.

In this section we present improved I/O-efficient algorithms for several basic graph problems that use the data structures presented in Section 2.

3.1 Improved Minimum Spanning Tree Algorithms

There are well-known $O(V \log V)$-time algorithms for the minimum spanning tree problem. Those techniques, however, are not suitable for external-memory computation. An algorithm is presented here that uses $O(\log B \cdot sort(E) + \log V \cdot scan(E))$ I/Os to compute a minimum spanning tree.

Initially, the *size* of an adjacency list is the number of edges in it. When two adjacency lists are merged, the size of the resulting list is the sum of the sizes of the two lists. An adjacency list is *small* if its size is less than B, and *large* otherwise. A vertex is a *large*-degree vertex if its adjacency list is large. Otherwise it is a *small*-degree vertex.

First we present an algorithm for a restricted case in which there are no small-degree vertices in the graph. The skeleton of the algorithm is familiar [11]: it consists of a number of phases, and in every phase it uses the smallest-weight-edge on every vertex to form a set of connected components. Vertices in a component are then grouped into pairs for merging. The edges that are shrunk are designated MST edges, the adjacency lists of component vertices are merged to obtain the adjacency list of the new vertex, and internal edges are removed from the combined adjacency list.

As we shall see, the number of pairs of vertices that are shrunk into single vertices in a phase is a constant fraction of the number of vertices in that phase. Therefore, each phase

reduces the number of vertices in the graph by a constant factor. This implies that the number of phases in the algorithm is $O(\log V)$. We shall also see that each phase requires no more than $O(scan(E))$ I/Os.

3.1.1 An Algorithm for Large-Degree Graphs

The undirected graph $G = (V, E)$ is represented as a list of adjacency lists. At any intermediate stage in the algorithm, we have a graph G', each vertex of which corresponds to a component of G shrunk into a single vertex. Our representation of the graph is a sequence of the adjacency lists of the vertices of G'. At the head of each adjacency list is the smallest-weight edge incident on the vertex. Then follows the list of vertices of G which have merged to form this vertex. This is followed by the rest of the edges, ordered by the vertex number of the vertex at the other end of the edge.

The algorithm consists of a number of *phases*. In one phase, each adjacency list undergoes no more than one merge, and a constant fraction of the lists undergo a merge.

Description of a phase: The process involves a scan of the list of lists to produce a new list of lists. When the scan encounters a new adjacency list (for some vertex s not involved in a merge so far in this phase), we try to merge it with another list. Let the smallest-weight edge point to vertex u. u may have become part of some vertex v through merging and renaming. One lookup is needed to know v. If the adjacency list of v has not been involved in a merge in this phase, merge the two lists and put the result in the new list of lists being produced. The new vertex gets the smaller of the two vertex numbers. The smallest-weight edge, which led us to v, is removed and included in a set of tree edges.

In case the vertex v has already been merged with some vertex in this phase, we will try to merge s with another vertex that desires to merge with v, so that over a number of phases, v and all the vertices whose smallest-weight edges point to v are merged together. If another such vertex t has already been encountered, it will be 'found waiting on v', and can be discovered by a single lookup of v. If such a vertex t is indeed found, we merge t and s, remove t from the 'waiting', and delete the heavier of the smallest-weight edges for s and t and include it among the tree edges. This is justified because the smallest-weight edge on any vertex has to be in a minimum spanning tree. The lighter of the two smallest-weight

edges is the smallest-weight edge on the vertex resulting from the merge and will ensure that the resulting vertex merges with v in some future phase. This means that for any smallest weight edge (x, y), the vertices x and y are ultimately going to be merged, and the edge included in the spanning tree.

If no vertex is waiting on v, we simply make s wait on v.

The graph thus produced by merging vertices may contain multiple edges between vertices. Merging two vertices with more than one edge between them gives rise to internal edges. While merging the adjacency lists for some u and v, we do the following simple bookkeeping to make sure all internal edges are thrown away. Each of the two adjacency lists being merged contains a list of vertices and a list of edges. We take the list of edges from u and the list of vertices that constitute v, and scan them simultaneously. Those edges of u whose destinations are among the vertices which have been merged to form v are going to become internal edges after u and v are merged. These edges are removed from the graph. As the edge-list of u and the vertex-list of v are both sorted by vertex number, one simultaneous scan over the two lists is sufficient to remove all such edges. Similarly, we remove edges of v that will become internal edges as a result of this merge.

To analyze the algorithm's performance, we simply bound the number of I/Os used per phase and the number of phases required by the algorithm, and the desired bound on the algorithm follows.

Lemma 6 *The large-degree MST algorithm involves no more than* $\log V$ *phases.*

Proof: This is because at the end of a phase, each vertex has either been involved in a merge or is waiting on another vertex, and no more than one vertex can be waiting at any particular vertex. It follows that each phase reduces the number of vertices by at least a third. ■

Lemma 7 *The number of I/Os involved in a phase is no more than* $O(scan(E))$.

Proof: It is clear from the description of a phase that we require no more than two scans of any adjacency list, plus a constant number of block reads/writes per vertex. As every

adjacency list is of length at least B, $V = O(\frac{E}{B})$. So the total number of I/Os involved in these reads/writes is bounded from above by $O(scan(E))$. ■

It follows that

Theorem 3 *The large-degree MST algorithm requires at most* $O(\log V \cdot scan(E))$ *I/Os.* ■

3.1.2 An Algorithm for the General Case

Now we present an algorithm that does not require that all adjacency lists be large. Small adjacency lists are a problem, because it is more difficult to process them in an I/O-efficient way. The previous algorithm pairs vertices together and merges their adjacency lists. If at least one of the adjacency lists is large, the process of merging them needs no more than $O(l/B)$ I/Os, where l is the size of the resulting list. However, if both the lists are small, we require at least $O(1)$ I/Os for the merging, even though l/B may be much smaller than 1. So we no longer have the $O(scan(E))$ upper bound for a phase. The following algorithm remedies the situation by converting the process of merging small lists into that of carrying out $O(1)$ sorts on them.

The algorithm, like the previous one, consists of a sequence of phases. In each phase, we group vertices together and shrink each group into a single vertex, in the process reducing the number of vertices in the graph by at least a factor of two. For grouping the vertices, we take the smallest-weight edge incident on each vertex, and consider the subgraph induced by this set of edges. Every vertex is then grouped with one or two of its neighbours in this induced graph, using a process that employs $O(V)$ heap operations and brings neighbouring vertices together in the list of adjacency lists.

Once we have grouped the vertices thus, we have to merge each group into a single vertex. Each group that includes at least one vertex with a large adjacency list is merged as before. The remaining adjacency lists are then marked with the vertices they must merge with, and by sorting the collection of lists, we bring 'partners' together and merge them.

We assume that small-degree vertices have smaller vertex numbers than large-degree vertices. Their adjacency lists will therefore occur before large adjacency lists in our list of

lists. This can be initially achieved by renumbering the vertices and reordering the list of adjacency lists. This requires $O(1)$ sorts on the set of edges, costing $O(sort(E))$ I/Os. Our algorithm then maintains this property throughout.

3.1.3 Description of a phase

First of all, we do a scan of the list of adjacency lists to get the smallest-weight edge for every vertex. Using these edges, we use the following scheme to pair up the vertices for merging:

First consider all the edges going from higher-numbered vertices to lower-numbered vertices. For every edge (i, j), create a pair (i, j) and insert the pairs in a heap, treating j as the primary and i as the secondary key (that is, $(i_1, j_1) < (i_2, j_2)$ if and only if either $i_1 < i_2$ or $i_1 = i_2$ and $j_1 < j_2$). We also insert a pair $(0, v)$ for every vertex v. Now, using *deletemin* successively, we obtain the vertices pointing to the smallest numbered vertex u (and also the vertex u itself if the entry $(0, u)$ has not already been deleted). We construct pairs out of these vertices. If an odd number of vertices are involved, the pairing process leaves one unpaired vertex, which is left *waiting* on u. Whenever a vertex v is paired up, we delete the entry $(0, v)$ from the heap. This ensures that v is not paired up with some w for which there is an entry (w, v) in the heap.

The pairing step is then repeated using edges directed from lower-numbered vertices to higher-numbered vertices. But we make sure that the entry $(0, v)$ is not inserted in the heap for any vertex v that has already been paired up in the previous step.

Each of these two pairing steps may leave one vertex waiting on a vertex u. If in the end there are two vertices waiting on u, they are paired together. If there is one vertex waiting on u, it is added to the pair that contains u, to form a triple.

At this point, all our vertices are grouped in groups of size two and three, and all the vertices in a group belong to the same connected component in the subgraph induced by the collection of the smallest-weight edges of all the nodes. Instead of shrinking the whole connected component all at once and extracting all the smallest-weight edges, we shrink each group into one vertex. So over a number of phases, the whole connected component will shrink into one vertex. If more than one of the vertices within a group have their smallest-weight edges pointing outside the group, we retain only the one with the smallest

weight, and remove the rest from the resulting graph and add them to the MST. The removal is justified since the smallest-weight edge on any vertex is always included in a minimum spanning tree, while the edge that is left behind will ensure that the resulting vertex is later merged into the rest of the connected component.

Now that we have determined which vertices are to be merged together, all that remains is the actual task of merging. The pairs (or triples) that contain at least one large-degree vertex are merged as in the restricted-case algorithm above, and the resulting vertex is named after the smallest-numbered large-degree constituent vertex. To merge pairs (or triples) containing small-degree vertices only, we give every adjacency list a *destination*, that is the vertex number of the vertex to be formed as a result of the merge. Then all the small adjacency lists are sorted together by destination. This ensures that the adjacency lists that merge together are placed one after the other. Then the merging can be accomplished in one scan. Finally, the small adjacency lists are sorted by size, and the small-degree vertices renumbered to ensure that any vertices whose adjacency lists become large as a result of the last merge are placed after the small-degree vertices. This completes one phase of our algorithm. Once again, $O(\log V)$ phases are required by the algorithm, since a phase reduces the number of vertices by a fraction.

3.1.4 I/O complexity of the algorithm

As noted before, the initial reordering takes $O(sort(E))$ I/Os.

Below, we break up the I/O cost of the algorithm into a sum of many terms and derive an upper bound for each.

Lemma 8 *Excluding merges that involve small-degree vertices only, the merging of adjacency lists requires $O(\log V \cdot scan(E))$ I/Os over all the phases of the algorithm.*

Proof: As in the last algorithm, we require $O(scan(E))$ I/Os per phase, and there are no more than $O(\log V)$ phases. ■

Lemma 9 *Extraction of smallest-weight edges from vertices and the process of pairing vertices requires a total of $O(sort(V) + \log V \cdot scan(E))$ I/Os over all the phases of the algorithm.*

Proof: Let us consider a phase in which the *residual graph* (the graph resulting from the action of the algorithm so far) G' has V' vertices and E' edges. Consider the process of extracting the smallest-weight edges from all the vertices and using them to pair vertices up. Extracting the edges requires just a scan of the list of adjacency lists, which costs $scan(E') = O(scan(E))$ I/Os. Over all phases, this part of the cost adds up to $(\log V \cdot scan(E))$ I/Os. Then, we need $O(V')$ heap operations, and (let us say) a constant number of scans and sorts on $O(V')$ items to order and and combine the results of the two pairing steps and so forth. The I/O cost of all this is $O(\frac{V'}{B} \log_{\frac{M}{B}} \frac{V'}{B})$. As V' goes down by at least a constant fraction in every phase, this term adds up to just $O(\frac{V}{B} \log_{\frac{M}{B}} \frac{V}{B}) = O(sort(V))$ over the whole algorithm. ■

As noted above, the merging process requires $O(scan(E))$ I/Os as before, except for the extra work done on the smaller adjacency lists; that is, in merging them and in renaming and reordering their vertices. Let us now bound the cost of this work.

Lemma 10 *Merging, renaming and reordering small-degree vertices requires a total of* $O(\log B \cdot sort(E))$ *I/Os over the entire algorithm.*

Proof: Let E'_s be the total number of edges in all small adjacency lists in this phase. The extra I/O done in dealing with the small adjacency lists is $O(sort(E'_s))$, as it involves a constant number of sorts on the small lists. Let each edge thus sorted be charged $O(\frac{1}{B} \log_{\frac{M}{B}} \frac{V}{B})$.

What does this cost add up to over the entire algorithm? Let us for the moment ignore adjacency lists with one or two edges at this point: they can separately be taken care of easily. An edge is charged no more than once on this count in any particular phase. After i phases, the size of every list is at least 2^i. Thus, an edge can only be charged $O(\log B)$ times in this fashion before it becomes a part of an adjacency list of size B or more, after which it is never involved in any sorting. The total cost charged to all the edges in this fashion is therefore no more than $O(\log B \cdot \frac{E}{B} \log_{\frac{M}{B}} \frac{E}{B}) = O(\log B \cdot sort(E))$ over the entire algorithm. ■

As the three lemmas above account for all the I/O cost of the algorithm, we have

Theorem 4 *The general-case MST algorithm requires at most* $O(\log B \cdot sort(E) + \log V \cdot scan(E))$ *I/Os.* ■

This I/O performance is better than that of the best previously-known algorithm, due to Chiang et al. [6], except when the graph is extremely dense ($E = \Theta(V^2)$).

Finally, we note that our algorithm will also suffice to find minimum spanning forests in graphs that are not connected, and thus immediately leads to an algorithm to compute connected components with the same bound on I/Os.

3.2 A Generalised Deterministic List-Ranking Algorithm

In the following, an edge is a *forward* edge if it points from a higher-numbered vertex to a lower-numbered one, and a *backward* edge otherwise.

Chiang et al. [6] suggested a method to deterministically rank an N-vertex linked list using $O(sort(N))$ I/Os, by splicing out entire chains of forward or backward edges, but it requires that M/B be sufficiently large for the algorithm to work. We present an alternate algorithm using an I/O-efficient heap that removes this restriction on B. Another way to remove this restriction is suggested by Arge [2].

First, we show that given a set of lists constructed out of N vertices in which all the edges are forward edges, we can in $O(sort(N))$ I/Os rank all the lists. This can be done as follows: insert each edge (i, j) into a heap, with i as the key. Extract the first edge (u, v) from the heap; u must be the head of a list, so give it rank 1, and insert a special element $(v, -2)$ into the heap. Thereafter, each time we remove an edge (v, w) from the heap, if it is not immediately preceded by an edge of the form $(v, -r)$, we give it rank 1 and insert $(w, -2)$ into the heap; if it is preceded by an edge $(v, -r)$, we give it rank r and insert $(w, -(r + 1))$ into the heap. (The head of the list can also be encoded into the element and passed along, for splicing purposes.) Clearly, this ranking procedure will take $O(sort(N))$ I/O steps, since there are $O(N)$ heap operations involved, and the heap size is no more than $O(N)$.

Now let us describe a technique to splice out vertices from a list. First, remove all edges going from higher-numbered to lower-numbered vertices (*back* edges) from the list. Now, using the method described above, rank the collection of lists thus obtained. This ranks all the vertices relative to the first vertices of their respective lists. Now, splice out all vertices except the first and last of every list by constructing a bridge edge from the first vertex to the last one. Similarly, remove all the forward edges, rank the set of lists that consists of all

the back edges, and splice out all the internal vertices. Clearly, all this can be done in $O(1)$ sorts and scans.

Will the above remove at least a constant fraction of all vertices? It may not. Consider a list in which there are no consecutive forward or back edges. The above procedure will remove no vertices at all. The next step takes care of this. Let $p(v)$ and $s(v)$ denote the predecessor and successor respectively of a vertex v. Construct a tuple $(v, p(v), s(v))$ for every v and insert in a heap with v as the key. Now we will extract these tuples in order. If $v < p(v)$ and $v < s(v)$, then $(p(v), v)$ is a back edge while $(v, s(v))$ is a forward edge. We will splice out all such v, and construct a bridge edge from $p(v)$ to $s(v)$.

The last step described above splices out the head of every maximal list of forward edges. It is easy to see that this, together with the rank-and-splice process before it, splices out at least half the vertices. Now we can recursively rank the smaller list, and from that obtain the ranks of the spliced-out vertices. The non-recursive part of the algorithm just requires $O(1)$ sorts and scans, which cost $O(sort(N'))$ I/Os where N' is the number of vertices left. The overall I/O complexity of the algorithm is the sum of a series of such $O(sort(N'))$ terms, one for each recursive call. As the N' is N to begin with and is reduced by a constant fraction every time, the sum telescopes to just $O(sort(N)) = O(\frac{N}{B} \cdot \log_{\frac{M}{B}} \frac{N}{B})$.

Theorem 5 *List ranking can be done deterministically using $O(sort(N))$ I/Os.* ■

(Arge [2] reports the same bound, using a different data structure.)

3.3 An Algorithm for Single-Source Shortest Paths

Using our tournament tree, we can obtain an I/O-efficient version of Dijkstra's algorithm for the single-source shortest paths problem. The tournament tree replaces the priority queue in the original algorithm. Initially we let our V vertices be initialised to infinite distance except for the source which is at distance zero. Using the distance as the key, we construct a tournament tree of these V elements. For the next V steps, we do a *deletemin* on the tree, read the adjacency list of the vertex obtained, and for every edge $e = (u, v)$ in the adjacency list (except when v is the vertex that precedes u on the shortest path from the source to u:

the identity of that vertex v can be stored in the tournament tree entry for u) we issue a tournament tree *update* which tries to update the key of vertex v to $x + w(e)$, where x is the distance of vertex u from the source and $w(e)$ the weight of edge e. Recall that an *update* does not affect the present tournament tree entry if the present value is smaller than the new value. It is easy to see that this takes $O(V + \frac{E}{B} \log_2 \frac{E}{B})$ I/Os: $O(V + \frac{E}{B})$ to read all the adjacency lists, and $O(\frac{E}{B} \log_2 \frac{E}{B})$ for all the tournament tree operations.

This algorithm differs from Dijkstra's in one respect. We update the tournament tree entry for each neighbour v of u except for its shortest-path predecessor. If we were to follow Dijkstra exactly, we would do this only for those neighbours which aren't yet included in the shortest-path tree being constructed. But we have no I/O-efficient way of knowing whether a vertex has already been included in the shortest path tree. This departure causes the following problem. Let $d(v)$ denote the length of the shortest path from the source to a vertex v. Let u and z be two neighbors and let (v, z) be the last edge on the shortest path from the source to z. Further, assume that $d(u) < d(v) < d(z)$. When *deletemin* returns $d(z)$ from the tournament tree, the algorithm will update the key of u to $d(z) + w(z, u)$, which is not correct, as $d(u)$ has already been determined, and the tournament tree entry for u has been deleted. We do not want to visit u again, so we must do a $delete(u)$ to undo the update just performed. The problem is to identify the vertices u for which this is to be done.

An auxiliary heap is employed to keep track of such vertices. When a vertex x is returned by a deletemin on the tournament tree, we not only update the tournament tree key of every neighbor y of x to $k = d(x) + w(x, y)$, but also insert a pair (x, k) into the heap, using k as the heap key. As the algorithm proceeds, we keep track of the smallest element in both the heap and the tournament tree. If the element deleted from the tournament tree is smaller, we execute the basic step described above, and then remove the next smallest element from the tournament tree. Otherwise we use the smallest element from the heap, which is a pair (x, k) in the following fashion: the tournament tree entry for the vertex x is deleted. The next smallest item from the heap is now extracted.

Because $d(u) < d(z) \leq d(u) + w(u, z) < d(z) + w(z, u)$, this is the sequence of events: first $d(u)$ is computed, the tournament tree entry for u deleted, and $(u, d(u) + w(u, z))$ inserted in the heap. At some time after this the spurious tournament tree update for vertex u (with key $d(z) + w(z, u)$) is introduced. But as $d(u) + w(u, z) < d(z) + w(z, u)$, the heap entry

$(u, d(u) + w(u, z))$ is taken out (and consequently a $delete(u)$ is executed on the tournament tree) before the spurious entry can be returned by deletemin.

The $O(E)$ added heap operations require an additional $O(\frac{E}{B} \log_{\frac{M}{B}} \frac{E}{B})$ I/Os. Our result follows immediately.

Theorem 6 *Single-source shortest paths can be found using* $O(V + \frac{E}{B} \log_2 \frac{E}{B})$ *I/Os.* ■

We believe that this is the best known upper bound on I/Os for this problem.

3.4 An Algorithm for Breadth-First Search

Our breadth-first search algorithm is similar to the algorithm for single-source shortest paths just discussed.

We start with a tournament tree containing the V vertices and their key values, which are as follows: the root is initialised to zero, the rest to infinity. We keep a counter initialised to 0. The basic step of the algorithm is as follows: the vertex v with the smallest key is extracted from the tournament tree. Let $v_1, v_2, \cdots, v_l$ be vertices in the adjacency list of v, excluding the parent of v in the BFS tree. Their keys are updated to $k, k+1, k+2, \cdots, k+l$ respectively, where k is the value of the counter. The counter is changed to $k + l$.

The scheme follows the familiar BFS algorithm except in one way. Although we should only visit the unvisited neighbours of v, we visit all of its neighbours, as we are making no distinction between neighbours that have already been visited and the ones that haven't. This is because we can not efficiently find out if a particular vertex has already been visited. The problem is similar to the one we encountered in our algorithm to find single-source shortest paths. Again, an auxiliary heap corrects for spurious updates of vertices that have already been visited and deleted.

Let $key(v)$ denote the key of the vertex v when it is extracted by deletemin. Let $p(v)$ be the parent of v in the BFS tree. If two vertices in the graph have an edge between them, and neither is the BFS parent of the other, then they are either at the same level or at adjacent levels in the BFS tree. Let u and v be two such vertices, such that $key(u)$ is smaller than

$key(v)$. Evidently, $key(p(v)) < key(u)$. The key of vertex v is set to $key(v)$ by $p(v)$, and on encountering u, we attempt an update (which fails) of the key of v. Later, when v is extracted, we try to update the key of u. But by now, u has already been extracted and its place in the BFS order determined. The current key value of u is, therefore, infinity. Updating it will cause u to be visited again, which is incorrect.

To take care of this problem, we use an auxiliary heap to keep information that will tell us that u precedes v in the BFS order, so v should not update the key of u. This is done as follows. When the entry $(u, key(u))$ is extracted from the tournament tree, we attempt to update the keys of the neighbours of u, including v. Let us say that we sought to update the key of v to some value x. We keep track of these attempted updates by inserting an entry (u, x) into the heap, which tells us that u attempted an update with key value x. Heap entries are pairs of values, and the second value is used as the heap key.

The modified basic step of the algorithm is: we keep extracting the smallest entries in both the heap and tournament tree. So when the smallest entry (of the form $(v, key(v))$) is extracted from the tournament tree, we proceed, as already described, to include v in the BFS tree and to attempt to update the keys of all its neighbours. But if the smallest entry (of the type (u, x)) comes from the heap, we conclude that the vertex u attempted to update the key of some v to the value x, and the update failed. Two things then become obvious: one, the entry for v has already been extracted from the tournament tree and has caused an 'undesirable' update of u; and two, that this undesirable update is of some key value larger than x. So we promptly proceed to issue the signal *delete(u)* to the tournament tree. The effect of the undesirable update of u is thus undone before it can cause us to visit u again.

With this correction, our algorithm visits vertices in the same order as the usual BFS algorithm. The resulting algorithm performs a total of $O(E)$ tree and heap operations, in addition to reading each adjacency list once. This yields the following bound:

Theorem 7 *Breadth-first search can be performed using* $O(V + \frac{E}{B}\log_2 \frac{E}{B})$ *I/Os.* ■

As in the case of the single-source shortest paths problem, we believe that this is the best known upper bound on I/Os for this problem.

3.5 An Algorithm for Depth-First Search

The basic difference between depth-first search and breadth-first search is the criterion for the selection of the next vertex to be visited. As in the case of BFS, we will use a tournament tree to store all the vertices that are candidates for future visits, but we will use a different criterion for comparison among candidates. There is an element in the tournament tree for each vertex v of the input graph G. In the tournament tree entry for a vertex v, the *key* field will contain a timestamp that records when the parent of v in the DFS tree was visited, and the index of v in its parent's adjacency list. In comparing two candidate vertices, the one with a later timestamp will be chosen. In case of equality, the one having the smaller index in the adjacency list of the (common) parent will be selected. Upon visiting a new vertex, the tournament tree entries for all its neighbours will be updated.

As in the case of BFS and single-source shortest path problems, the problem is avoiding updating the tournament tree entries for vertices that have already been visited. In an undirected DFS tree, any neighbour of v that has been visited before v must be an ancestor of v in the tree. Our objective, then, is to avoid updating tournament tree entries for the ancestors of the node being visited, so as not to visit them again. To do this, we keep track of the DFS ancestors of every node. The I/O cost of storing the ancestors of each vertex can be high. So we store the lists of DFS ancestors in a simple data structure described below.

The data structure used is essentially a balanced binary tree. The V leaves correspond to the vertices of G. Each leaf is to contain the list of DFS ancestors of the corresponding vertex. The internal nodes are buffers, each of size B. To add vertex u to the ancestor list of vertex v, we add the pair (v, u) to the root buffer. When a buffer becomes full its contents are divided appropriately between its two children. The list of ancestors of a vertex v can be obtained by reading all the buffers from the corresponding leaf to the root, extracting all entries of type (v, u) for some u that may be found in the buffers and combining them with the contents of the list contained in the leaf.

Now that we have a way of obtaining the list of all DFS ancestors of a vertex, we are able to implement the basic step of our DFS algorithm correctly: from the tournament tree, select the next vertex to be visited, read its adjacency list, and for all its neighbours that are not its ancestors, update their tournament tree entries. This entails sorting the list of ancestors and the list of neighbours and removing all common entries from the latter.

3.5.1 I/O Complexity of the Algorithm

There are no more than $E + V$ tournament tree operations. The additions to the lists of ancestors require $O(\frac{\log_2 V}{B})$ I/Os for each edge in G, and the extraction of information from that data structure requires $O(\log_2 V + l(v)/B)$ I/Os for a vertex v of G that has $l(v)$ DFS ancestors. Sorting of adjacency lists and ancestor lists requires $O(sort(E))$ I/Os overall. So the overall I/O complexity is $O(V \log_2 V + \frac{E}{B} \log_2 \frac{E}{B})$.

Chiang et al. [6] present a DFS algorithm that requires $O((1 + V/M)scan(E) + V)$ I/Os. Our algorithm outperforms that algorithm on all except very sparse graphs.

4 Conclusions and Open Questions

We have presented two I/O-efficient data structures — the I/O-efficient heap and tournament tree — and demonstrated how their use can simplify the design of I/O-efficient graph algorithms. Both of these data structures support all their basic operations: the heap with an amortised cost of $O(\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os per operation, and the tournament tree with an amortised cost of $O(\frac{1}{B} \log_2 \frac{N}{B})$ I/Os per operation on a data structure of at most N items.

The I/O-efficient heap led immediately to simple descriptions of optimal algorithms for sorting and list-ranking, while the update operation available in the I/O-efficient tournament tree proved useful in the design of algorithms for breadth-first and depth-first search and single-source shortest paths. We also obtained an algorithm for finding minimum spanning trees with improved I/O efficiency on all but the densest graphs. As is the goal of data structures, the I/O-efficient heap and tournament tree both simplified the descriptions of external-memory graph algorithms and led to improved efficiency.

Our results can easily be extended to the D-disk model when $D = O((\frac{M}{B})^\alpha)$ for some $\alpha < 1$. In this case, the asymptotic constant increases by a factor of $\frac{1}{1-\alpha}$, which is a fairly small constant as long as α is not too close to 1. We believe they are extendible to the more general case, but we do not at the moment have such an extension.

There are many other open questions, including:

- Can minimum spanning trees be found in as few as $O(sort(E))$ I/Os? How about in $O(sort(V))$, which was shown to be a lower bound by Chiang et al. [6]?
- Can depth-first search be performed as efficiently as breadth-first search — that is, with only an additive V rather than a $V \log_2 V$?
- Can the additive V (or $V \log_2 V$) terms be removed from the running times of the algorithms for breadth-first and depth-first search and single-source shortest paths?
- Can a tournament tree be designed with an (optimal) amortized I/O complexity of $O(\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B})$ I/Os per operation, instead of $O(\frac{1}{B} \log_2 \frac{N}{B})$ I/Os per operation?
- In what other areas can these data structures lead to improved algorithms? What other data structures would it be helpful to transfer to an external-memory setting?

References

[1] Ruemuler, C. and Wilkes, J.: An Introduction to Disk Drive Modelling. *IEEE Computer* 27(3), pp. 17-28, 1994.

[2] Arge, L.: The Buffer Tree: A new technique for optimal I/O-algorithms. In *Proc. Fourth Workshop on Algorithms and Data Struc.*, pp. 334-345, 1995.

[3] Callahan, P., Goodrich, M.T. and Ramaiyer, K.: Topology B-Trees and Their Applications. In *Proc. Fourth Workshop on Algorithms and Data Struc.*, pp. 381-392, 1995.

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

[5] Vitter, J.S. and Shriver, E.A.M.: Algorithms for parallel memory I: two level memories. *Algorithmica*, 12(2), 1994.

[6] Chiang, Y.-J., Goodrich, M.T., Grove, E.F., Tamassia, R., Vengroff, D.E. and Vitter, J.S.: External memory graph algorithms. In *Proc. 6th ACM-SIAM Symp. on Discrete Algorithms*, pp. 139-149, 1995.

[7] Nodine, M.H. and and Vitter, J.S.: Large-scale sorting in parallel memories. *Proc. 3rd Annual ACM Symp. on Parallel Algorithms and Architectures*, pp. 29–39, 1991.

[8] Nodine, M.H. and and Vitter, J.S.: Deterministic distribution sort in shared and distributed memory multiprocessors. In *Proc. 5th ACM Symp. on Parallel Algorithms and Architectures*, pp. 120-129, 1993.

[9] Goodrich, M.T., Tsay, J.-J., Vengroff, D.E. and Vitter, J.S.: External-memory computational geometry. In *Proc. 34th Annual IEEE Symp. on Foundations of Comp. Sci.*: 714-723, 1993.

[10] Arge, L., Vengroff, D.E. and Vitter, J.S.: External Memory Algorithms for Processing Line Segments in Geographic Information Systems. In *Proc. Third Annual European Symp. Alg.*, pp. 295-310, 1995.

[11] Johnson, D.B. and Metaxas, P.: A Parallel Algorithm for Computing Minimum Spanning Trees. In *Proc. 4th ACM Symp. on Parallel Algorithms and Architectures*, pp. 363-372, 1992.