I/O-Complexity of Graph Algorithms

Kameshwar Munagala[*] *Abhiram Ranade*[†]
Department of Computer Science and Engineering
Indian Institute of Technology Bombay
Mumbai 400076 India.

Abstract

We show lower bounds of $\Omega(\frac{E}{V}\,\mathrm{Sort}(V))$ for the I/O-complexity of graph theoretic problems like connected components, biconnected components, and minimum spanning trees, where E and V are the number of edges and vertices in the input graph, respectively. We also present a deterministic $O(\frac{E}{V}\,\mathrm{Sort}(V)\cdot\max(1,\log\log\frac{VBD}{E}))$ algorithm for the problem of graph connectivity, where B and D denote respectively the block size and number of disks. Our algorithm includes a breadth first search; this maybe of independent interest.

1 Introduction

Data sets of many modern applications are too large to fit into main memory, and must reside on disk. To run such applications efficiently, it is often necessary to explicitly manage disk accesses as a part of the algorithm. In other words, the algorithm must be designed for a model that includes disk, rather than the customary RAM model. Recently, this area has received a lot of attention, and algorithms have been developed for many problems using the *Parallel Disk Model*[1]. In this paper, we study several graph problems using this model and present improved upper and lower bounds.

The Parallel Disk Model consists of a processor, its memory, and D disks which can be accessed in parallel. Data is organized in *records*, with the memory able to hold M records. A record maybe thought of as a constant number of words. Records are stored sequentially on each disk, and the disk is divided into fixed contiguous blocks of B records, called *tracks*. The global disk address space is *striped* across the disks in the following manner: record i in track j of disk k is the $i+B(k-1)+BD(j-1)$ th record in the global disk address space. In one I/O step B records from any single track on each disk maybe exchanged with B records in the RAM, so that in a single step a total of at most BD records can be read or written from disk. The time required for computation is ignored; but computations can only be performed on records in memory. This is reasonable since typically disk accesses are orders of magnitude slower than CPU cycle times. It is customary to assume that $M > BD$. The following expressions are useful[1]:

$\mathrm{Scan}(N) = \frac{N}{BD}$: The number of disk accesses needed just to read N records stored contiguously in the global disk address space. Because the address space is striped across disks, the records must be in N/BD tracks on each disk, and thus that many parallel steps suffice. $\mathrm{Scan}(N)$ is usually a lower bound if input size is N.

$\mathrm{Sort}(N) = \frac{N\log N/B}{BD\log M/B}$: The optimal number of I/Os needed to sort N records stored contiguously in the disk address space using just comparisons[1]. This number is important because sorting forms an important primitive in many external memory algorithms.

The main problem considered in this paper is the connected component labelling problem. We assume that the input is provided as an unordered *Edge List.* We will use E to denote the number of edges and V the number of vertices. The edge list is given as a sequence of E records stored contiguously in the global disk address space, with each record consisting of a pair of integers in the range [1,V] indicating the edge endpoints. The output is an array $L[1\ldots V]$ stored contiguously as V records; we require $L[i] = L[j]$ iff i and j are in the same component.

We also study the problem of duplicate elimination, for itself and because it arises naturally while developing upper and lower bounds for many graph problems. The input in this case is a sequence of N records where each record is an integer in the range $[1,P]$, for any given P. The output is an array $C[1\ldots P]$ stored contiguously in P records on disk, with $C[i]=1$ if i occurs in the input,

[*]Author's current address: Department of Computer Science, Stanford University, Stanford CA94305. E-Mail: `kamesh@cs.stanford.edu`

[†]E-Mail: `ranade@cse.iitb.ernet.in`

and 0 otherwise.

Our main results are as follows:

1. **Lower bounds for graph problems:** We show that the connected component labelling problem requires time $\Omega(\frac{E}{V}\,\text{Sort}(V))$. We extend this bound to several problems including biconnectivity, minimum spanning tree, and decision versions of these problems.

2. **Upper bounds for connected component labelling:** We present a deterministic algorithm which can do labelling in time $O(\frac{E}{V}\,\text{Sort}(V) \cdot max(1, \log\log\frac{VBD}{E}))$. This is optimal for graphs with average degree $\Omega(BD)$.

3. **Duplicate elimination:** We show a lower bound of $\Omega\left(\frac{N}{P}\,\text{Sort}\,P\right)$ and give a matching upper bound. We also extend the upper and lower bound to problems such as histogram computation, and more generally, multiprefix computation.

Previously the only lower bounds known for graph problems were the obvious bound Scan(E) and a bound Sort(V) for sparse graphs ($E = O(V)$) proved by Chiang et al[2]. Our bound is stronger if the graph is not sparse, and further our bound applies to other problems besides connected component labelling. We note however that the bound of [2] applies also when the input is presented as an adjacency list (defined in Section 5.2), while ours applies only if the input is an edge list.

The best previous deterministic algorithm for connected component labelling is by Kumar and Schwabe [3], and this works in time $O(\log B \cdot sort(E) + \frac{E}{B}\log V)$, which is roughly $O(\log M \cdot \frac{E}{V} sort(V))$. Our algorithm is much faster than this, since $M > BD > VBD/E$, and in fact optimal when the average degree of the graph is $\Omega(BD)$. Earlier a $O(\log\frac{V}{M} \cdot \text{Sort}(E))$ algorithm was given by Chiang et al [2], and this bound was matched recently by Abello et al [4] using a simple recursive algorithm. For sparse graphs($E = O(V)$) closed under edge contraction, the algorithm in [2] is shown to work in time $O(\text{Sort}(V))$, and is hence optimal. Our algorithm also has this property, and is of course far better for general sparse graphs.

Chiang et al [2] also give a $O(\frac{E}{V}\,\text{Sort}(V))$ randomized algorithm, which is optimal.

Duplicate elimination was investigated by Arge et al [5], who present a general lowerbound technique for comparison based external memory algorithms. We extend their work in two ways. In Arge et al's model, the records being compared are assumed to come from an unbounded set. In our model, the records come from a finite set of integers $[1, P]$, which makes finding a lower bound harder. Our second extension is that we allow indirect addressing while performing I/O; we believe that this is a very natural operation to use when the records come from a finite set, and thus lower bounds must account for it. Also, our model is powerful enough to express I/O cacheing.

Throughout this paper we assume that V and P are larger than M, (if not there are simple algorithms which work in a single scan of data) but that $\log V$ and $\log P$ are smaller than $B \log\frac{M}{B}$.[1]

Paper Outline: After a discussion of the lower bound framework, we derive lower bounds for the problem of eliminating duplicates from a multiset whose elements can take values only from a finite domain. We then reduce this problem to the problem of graph connectivity, and other related graph problems, thereby finding lower bounds for them. We then describe improved upper bounds for the connected components problem.

2 Lower Bound Framework

The computational model we use for proving lower bounds is called the *P-way Indexed I/O tree*, which is an extension of the I/O tree of Arge et al [5]. In addition to the operations allowed in the I/O Tree, our model allows P-way indexed accesses to disk, with P a parameter to be fixed later. We first discuss a restricted model, which has $D = 1$ disks. The case of general D and an additional extension is considered in section 2.3.

2.1 Restricted model

A restricted indexed I/O tree T is similar to a decision tree, with execution starting at the root and travelling to a leaf. Initially, the disk holds the input in contiguous records, and the memory is empty. The tree may have three kinds of non-leaf nodes:

Comparison Node: Each such node is labelled by pair of numbers $< i, j >$, and has outgoing edges labelled $\{<, \geq\}$. The node causes a comparison to be performed between the ith input record and the jth input record, both of which must be present somewhere in main memory. Depending upon the outcome, the appropriate outgoing edge is taken.

Indexed Input node: Each node has a label $< i, m_1, m_2, \ldots, m_B, t_1, t_2, \ldots, t_P >$, with P outgoing edges labelled 1 through P. This node causes a track to be read into main memory record positions $m_1, \ldots, m_B$. The track is determined by the value

[1] The pagesize for many disks is several kilobytes; assuming one kilobyte for our problems, say 50 records fit in a kilobyte, i.e. $B = 50$. Even for a trivial memory size of 1 Megabyte we get $\log\frac{M}{B} = 10$. Thus we require $V, P < 2^{500}$ which is reasonable.

of the ith input record which must be present somewhere in main memory: if the value is j then track t_j is read, and the corresponding outgoing edge traversed. The value j is required to lie in the range $[1, P]$. If all $t_1 = t_2 = \cdots = t_P$, then the parameter i is ignored and the track is read into the specified positions, and the first outgoing edge is traversed (we dont require that all tree edges be traversible for some input). This corresponds to non-indexed input– this would be necessary sometimes, say at the beginning of execution.

Indexed Output node: Similar to indexed input nodes, except the track is written from the specified records in memory.

We will use the term I/O node to mean either in indexed input node or an indexed output node.

Each leaf v has a label Solution(v). Let π denote the problem being solved, x an input instance, $\pi(x)$ the output required, and $T(x)$ the leaf reached during execution. The tree T is said to solve the problem π if for every x, Solution$(T(x)) = \pi(x)$. Let $I/O_T(x)$ be the number of I/O-nodes, i.e. indexed input nodes and indexed output nodes on the path from the root to $T(x)$. Let I/O_T denote the maximum value of $I/O_T(x)$, this is the quantity we will lower bound. Note finally that we dont insist that the correct answer be written out on disk; only that the correct answer (leaf with the correct label) be identified by the algorithm. This is adequate since we are only interested in using the model to prove lower bounds.

I/O trees vs. Decision trees: It turns out that an Indexed I/O tree for solving a problem can be transformed to a (short) *binary* decision tree for solving the same problem. Then by bounding the number of comparisons in the decision tree we can establish a bound on the number of I/O operations. Our notion of a decision tree is standard. We allow comparison nodes in it with two outgoing edges labelled $\{<, \geq\}$. Further at each node, comparisons are allowed between arbitrary records, whether or not they are in memory. In a decision tree we also allow comparison between records and constants, this is not allowed in the restricted model (but also see section 2.3). For a tree T (indexed I/O tree or decision tree) and input instance x let Path$_T(x)$ denote the number of comparison nodes on the path from the root to $v_T(x)$. The result stated below is a generalization of the one stated for I/O-trees in [5].

LEMMA 2.1. *Let π be problem solved by a restricted indexed I/O tree T, with n the number of records constituting the input. There exists a decision tree T_c solving π, such that:*

$$\mathrm{Path}_{T_c}(x) \leq n \log B + I/O_T(x) \cdot O\left(B \log \frac{M-B}{B} + \log P\right)$$

Proof. We will first apply a sequence of transformations to T to get another indexed I/O tree T' solving π in which the number of comparisons between successive I/O nodes is small. This construction is adapted from [5], and is reproduced here for completeness. We will next modify T' to the required decision tree T_c.

Constructing T': Initially all the I/O nodes in T except the root are coloured white, the root is coloured black. Each basic transformation alters the tree and colours some nodes black. After all nodes have been coloured black we get T'. After each transformation, the resulting tree will continue to be an indexed I/O tree solving π. Further, it will satisfy the following *Ordering Assertion: If execution reaches any black node, the comparisons performed till then will be sufficient to construct the total order of the records in main memory at that time.*

The basic transformation is applied at any black node v having some white descendants and no black descendants. Let $v_1, \ldots, v_k$ be descendant I/O nodes that can be reached from v without passing through another I/O node. Only comparison nodes occur between v and all v_i and these form a subtree S.

If v is an output node, then notice that by the ordering assertion the outcome of every comparison inside S is already known! Thus we can directly connect v to a single v_i and colour that v_i black, completing the transformation.

If v is an input node there are two cases depending upon whether or not the track t read at v was read for the first time. In case t had been read earlier, we know the ordering of the records in t. By the ordering assertion we know the total order on the $M - B$ records in main memory as well. Thus to maintain the ordering assertion at the next I/O node, we simply need to do a merge; this can be done using a decision tree S' of height $O(B \log \frac{M-B}{B})$. This tree S' is attached below v instead of S. Of course, S' will in general have more leaves than S. Say the leaves of S' are $v'_1, v'_2, \ldots, v'_l$. Let $I(u)$ denote the set of input instances for which a given node u is visited. It is easily seen that $I(v'_1), \ldots, I(v'_l)$ and $I(v_1), \ldots, I(v_k)$ are both partitions of $I(v)$, and that the former is a refinement of the latter. Thus beneath each v'_j we attach the subtree that is under v_i, where $I(v'_j) \subseteq I(v_i)$. Clearly, the new tree that results also solves π.

In case t is being read for the first time, we may not know the ordering of the records in t. So we use a decision tree to sort these records first, and then do the merge with the $M - B$ records in main memory. Subtrees are attached below the leaves of this as in the previous case. Thus the total height needed is at most $B \log B + O(B \log \frac{M-B}{B})$.

When all I/O nodes become black, we are guaranteed to have a tree T' which solves π, and in which between consecutive I/O nodes there are at most $B \log B + O(B \log \frac{M-B}{B})$ comparison nodes. Further, the term $B \log B$ can arise only while reading inputs for the first time– thus only n/B times. Thus the total number of comparison nodes encountered on input x is $n \log B + I/O_{T'}(x) \cdot O(B \log \frac{M-B}{B}) = n \log B + I/O_T(x) \cdot O(B \log \frac{M-B}{B})$.

Constructing T_c: T' is almost a decision tree, except that the I/O nodes in T' effectively allow a P way branch based on the index variable. But we can replace this with a decision tree of depth $O(\log P)$. When this is done, we get the required decision tree T_c. Since each I/O node contributes a $\log P$ additional height, the result follows.

2.2 I/O Cacheing

In our full model the indexed input nodes are adaptive: when control reaches such a node with label $< i, m_1, m_2, \ldots, m_B, t_1, t_2, \ldots, t_P >$, a check is first made to see if track t_j is already in memory, where j is the value of input record i. If t_j is already in memory, then no I/O is actually performed, but the data is copied within the memory into memory positions $m_1, \ldots, m_B$. If the track is not in memory, then an I/O is performed as usual.

Say a value i is a key for a record r if r is the label on the ith outgoing edge from some indexed input node. In other words, some input record must take the value i in order for r to be fetched. It is legal for a record to have many keys; however, it will be observed that generally the number of keys will be small. For instance, if our algorithm only contains array references of the form $A[X]$ (where X is some input record), then clearly the record $A[j]$ will only have j as the key. If for instance if our algorithm allows references of the form $A[X + K]$ where K is a constant, then the key for $A[j]$ would have to be $j + K$.

Under the assumption that *every record has $O(1)$ keys*, Lemma 2.1 can be shown to hold with small modification. The key idea is to extend the ordering assertion to be: *If execution reaches any black node, the comparisons performed till then will be sufficient to construct the total order of the records in main memory at that time and the keys of these records.* Note that at any instant the memory can hold only M records, and by the $O(1)$ keys assumption there are only $O(M)$ keys to consider (the key for any record is known by examining the indexed input nodes that can potentially read that record). Thus the ordering assertion now requires us to maintain a total order over $O(M)$ values instead of the M earlier. The rest of the details are omitted from this abstract.

We note that this change makes our model very powerful: using it we can implement dynamic behaviour such as LRU block replacement policy. We also note that it is possible to enhance the model so that cacheing happens at the granularity of records than tracks. We will discuss this at length in the final version.

2.3 Full Model

Our full model extends the restricted model in the following manner. First we allow D **Disks**. Instead of one track being read/written in a single I/O node, we allow D tracks to be read or written. Each I/O node now has D labels of the form described for the restricted model. Further, there are P^D outgoing edges, one edge for each of the P^D tuples that can be the values for the D index variables. Second, we allow **compiler generated constants**. At the start of execution, we allow the disk to hold tracks containing constants that might be needed by the program in addition to the program input. The constants would have to be read in explicitly in order to be used, just like input records, of course.

THEOREM 2.1. *Let π be problem solved by an indexed I/O tree T, with n the number of records constituting the input. There exists a decision tree T_c solving π, such that:*

$$\begin{aligned} &\mathrm{Path}_{T_c}(x) \\ &\quad \leq n \log B + D \cdot I/O_T(x) \cdot O\left(B \log \tfrac{M-B}{B} + \log P\right) \end{aligned}$$

Proof. Follows along the lines of Lemma 2.1, after noting that compiler generated constants do not need to be sorted even when read for the first time.

Following [5] we note that Lemma 2.1 and Theorem 2.1 also applies to algorithms having some other operations besides comparisons and indexed I/O. In particular, we can allow assignment statements which copy records within memory. These copying statements do not even need to be represented in our tree, since our comparison nodes refer to input records directly and not the positions at which the records are held in memory.

3 Duplicate Elimination in a Multiset

The $DE(N, P)$ duplicate elimination problem is as follows. The input is a sequence of N records where each record takes an integral value in the range $[1, P]$. The output is an array $C[1 \ldots P]$ stored contiguously in P records on disk; we require that $C[i] = 1$ if at least one input record takes the value i, and 0 otherwise.

LEMMA 3.1. *The depth of any decision tree solving $DE(N, P)$ is at least $N \log \frac{P}{2}$.*

Proof. Say an input instance is odd if the values of all its records are odd integers. Else it is non odd. We will show that in any decision tree solving $DE(N, P)$, distinct leaf nodes will be visited for distinct odd input instances. From this the Lemma follows since there are $\left(\frac{P}{2}\right)^N$ odd instances, and thus at least that many number of leaves.

Let $R = [r_1, r_2, \ldots, r_N]$ be the input records. Suppose that leaf l is reached in execution for an odd input instance. At l the set R is partitioned into subsets $R_1, \ldots, R_k$, such that all elements within each subset are known to be equal, and the subsets are related by $<, \leq, >, \geq$. This is the unique partial ordering of the sets R_i corresponding to the leaf l, and all input instances visiting l satisfy this ordering. Further since we allow comparisons between constants and records as well, our partial ordering will comprise of these constants in addition to the sets R_i. Suppose u_i and d_i are respectively the maximum and minimum values that are assigned to R_i over all odd input instances that visit l. If $u_i = d_i$ for all i, then we are done, since this means there is a unique odd instance visiting l.

Else there exists R_j for which $u_j > d_j$. Let Σ_u denote the set of subsets R_i, the records in which are required to have values $\geq$ the value assigned to the records in R_j in order to confirm with the partial ordering at l. Let I_u and I_d respectively denote input instances in which records in each R_i are respectively assigned values u_i and d_i. Note that the innput instance formed by taking the record-wise $max(min)$ of any two input instances visiting l also visits l. From this observation, is clear that the instances I_u and I_d visit l. Let e denote an even integer in the range $[d_j, u_j]$.

Now it will be observed that there exists a non-odd input instance which visits l in which the records in R_i are assigned value e. To complete this instance, we simply assign records in sets in Σ_u their values in I_u, and the rest their values in I_d.

THEOREM 3.1. (DUPLICATE ELIMINATION) *For any indexed I/O-tree T solving $DE(N, P)$, we have $I/O_T = \Omega(\frac{N}{P} \operatorname{Sort}(P))$ assuming $\log P < B \log \frac{M}{B}$.*

Proof. Using Theorem 2.1 and Lemma 3.1 we have

$$\begin{aligned} &N \log \frac{P}{2} \\ &\quad = N \log B + D \cdot I/O_T \cdot O\left(B \log \frac{M-B}{B} + \log P\right) \\ &\quad = N \log B + D \cdot I/O_T \cdot O\left(B \log \frac{M}{B}\right) \end{aligned}$$

using the assumption $\log P < B \log \frac{M}{B}$. The bound follows after rearranging.

3.1 Upper Bounds

The lower bound stated in Theorem 3.1 can be met using a simple algorithm as follows.

Divide the N input records into N/P groups of P records each. Sort the records within each group. For each group construct the solution to the duplicate elimination problem, i.e. construct a vector of size P where the ith entry indicates whether or not i occurs in the group. At this point there are N/P bit vectors, each of size P. The result of "OR"ing the bit vectors is clearly the solution to the original problem.

Sorting each group takes time $\operatorname{Sort}(P)$, so for N/P groups the time is $\frac{N}{P} \operatorname{Sort}(P)$. Each individual vector can clearly be constructed in time $\operatorname{Scan}(P)$ and hence the total time for constructing the vectors is $\operatorname{Scan}(N)$. Computing the OR of the vectors can also be done in time $\operatorname{Scan}(N)$. The total time is thus $O(\frac{N}{P} \operatorname{Sort}(P) + \operatorname{Scan}(N)) = O(\frac{N}{P} \operatorname{Sort}(P))$.

3.2 Generalization

It is easily seen that the upper bound as well as the lower bound applies to operations such as histogramming, where the ith output record must be set to the number of times i occurs in the input.

In fact we can define a generalized Multiprefix operation as in [6]. The input in this case consists of 3 arrays $Colour[1 \ldots N]$ and $Value[1 \ldots N]$ and $Table[1 \ldots P]$. The output is a single array $R[1 \ldots N]$. Let $i_1, \ldots, i_k$ denote in ascending order the indices for which the values in $Colour$ all equal i. Then we require that $R[i_j] = Table[i] + Value[i_1] + \cdots + Value[i_j]$. Further it is required to set $Table[i] = Table[i] + Value[i_1] + \cdots + Value[i_k]$. Clearly this operation has upper and lower bounds same as $DE(N, P)$. Note that the multiprefix operation is very versatile, by using appropriate associative operators + we can implement several operations such as table lookup/update, ranking keys of the same value, concurrent read/write operations as on PRAMs.

4 Lower Bound for Connected Components

We reduce the problem of duplicate elimination of certain kinds of sequences to the problem of computing connected components on graphs. Consider a sequence b of length N of numbers from the range $[P+1, 2P]$, where $P < N < P^2$. Suppose that b can be divided into P contiguous sequences, $b_1, b_2, \ldots, b_P$ each of length N/P such that all the elements in each sequence b_i are distinct. Let $DE'(N,P)$ denote the problem of removing duplicates from such sequences.

LEMMA 4.1. *The depth of any decision tree solving problem $DE'(N,P)$ is $N \log \Omega(P)$.*

Proof. Similar to the proof of Lemma 3.1, and thus omitted.

THEOREM 4.1. (CONNECTED COMPONENTS) *Any indexed I/O-tree T finding the connected components of a V vertex, E edge graph has $I/O_T = \frac{E}{V}\,\mathrm{Sort}(V)$ assuming $E \geq V$ and $\log V < B \log \frac{M}{B}$.*

Proof. We transform the problem DE' to connected component labelling. For any instance b of a $DE'(E - V/2, V/2)$ problem let b_j denote the jth subsequence of b of length $(E - V/2)/(V/2)$. Consider a graph G with V vertices and E edges as follows:

1. If $V/2 + i \in b_j$, then edge $(j, V/2+i)$ present.
2. If $i \in \{1, 2, \ldots, V/2 - 1\}$, then the edge $(i, i+1)$ is present.

Clearly the edge list of G can be constructed in $\mathrm{Scan}(E)$ time given the input instance b.

Given a connected component labelling $L[1 \ldots V]$ of G, we can solve duplicate elimination by noting that $i \in b$ iff $L[V/2 + i] = L[1]$. Thus, in $O(\mathrm{Scan}(E))$ extra time, we can find the array specifying which elements are present in b, thereby solving problem DE' for b. The bound then follows, using Theorem 2.1 and Lemma 4.1 and noting that $\mathrm{Scan}(E) = o(\frac{E}{V}\,\mathrm{Sort}(V))$.

4.1 Other Graph Problems

The lower bound in Theorem 4.1 actually applies to other problems such as connected components in bipartite graphs, biconnected components labelling, minimum spanning trees, ear decomposition, and their decision versions. Since all of these problems relate to duplicate elimination in a fashion similar to Theorem 4.1, we omit the details from this abstract.

5 Upper Bounds for Connected Components

We first show how to implement *Breadth First Search* (BFS) in time $O(\frac{E}{V}\,\mathrm{Sort}(V) + V)$. This by itself is an optimal labelling algorithm for $\frac{E}{V} \geq BD$. We then present a preprocessing step that transforms a labelling problem on V vertices to one on E/BD vertices. By running BFS after this we get a nearly optimal algorithm for sparse graphs as well.

5.1 Breadth First Search

We first convert the input to adjacency list form. This consists of two arrays $Ptr[1 \ldots V]$ and $Adj[1 \ldots 2E]$, with $Adj[Ptr[i] \ldots Ptr[i] + d_i - 1]$ holding the list of vertices adjacent to vertex i (d_i is the degree of vertex i). It is easily seen however that the conversion requires time $O(\frac{E}{V}\,\mathrm{Sort}(V))$.

Our connected components algorithm based on breadth first search is shown in Algorithm 1. The difference from the natural implementation is best explained by considering the execution for a connected graph. In this case, $\mathrm{Front}(t)$ as constructed by the algorithm is simply the set of vertices at distance t from the source (say vertex 1). On a RAM, it is customary to construct $\mathrm{Front}(t)$ by taking all the vertices that are adjacent to $\mathrm{Front}(t-1)$ but which havent been marked visited. Here we proceed differently. For vertex x let $\mathrm{Nbr}(x)$ denote its neighbouring vertices. For a set of vertices X let $\mathrm{Nbr}(X)$ denote the multiset formed by concatenating $\mathrm{Nbr}(x)$ for all $x \in X$. We construct $\mathrm{Front}(t)$ by (1) constructing $\mathrm{Nbr}(\mathrm{Front}(t-1))$, (2) removing duplicates from it, (3) eliminating from it those elements that occur in $\mathrm{Front}(t-1) \cup \mathrm{Front}(t-2)$.

We maintain a doubly linked list U in external memory to store the unvisited vertices at any stage. Initially, the list contains the vertices $1, \ldots, V$ in that order, with 1 at the head of the list. We also maintain an array $L[1 \ldots V]$, where $L[i]$ is the smallest vertex to which vertex i is connected.

Because of the adjacency list representation the total time spent in concatenating the lists (step 9) over the course of the entire algorithm is at most $\sum_{v \in V} \left\lceil \frac{\mathrm{Nbr}(v)}{BD} \right\rceil = \frac{E}{BD} + V = \mathrm{Scan}(E) + V$. The time for duplicate elimination is

$$\sum_t \left\lceil \frac{|\mathrm{Nbr}(\mathrm{Front}(t))|}{V}\,\mathrm{Sort}(V) \right\rceil$$
$$= \frac{\sum_t \mathrm{Nbr}(\mathrm{Front}(t))}{V}\,\mathrm{Sort}(V) + V$$
$$= O\left(\frac{E}{V}\,\mathrm{Sort}(V) + V\right)$$

since each edge occurs at most twice in the sum. To eliminate from $\mathrm{Nbr}(t)$ those elements that occur in $\mathrm{Front}(t-1) \cup \mathrm{Front}(t-2)$, we simply need to scan through these (sorted) lists. But $\sum_t \mathrm{Nbr}(t) = O(E)$, and $\sum_t \mathrm{Front}(t) = V$. Thus the time for scanning is clearly $O(\text{Scan}(E) + V)$, with the term V again accounting for the possible round off.

Algorithm 1: Breadth First Search
(1) **for** $i = 1$ **to** V **do** $L[i] = 0$.
(2) $\text{Front}(-1) = \text{Front}(-2) =$ **null**.
(3) $t = 0$.
(4) **while** $\text{Front}(t-1)$ is nonempty or unvisited vertices exist
(5) **if** $\text{Front}(t-1)$ is empty
(6) $\text{Front}(t) = l =$ Head of list U.
(7) **else**
(8) $\text{Nbr}(t)$ = concatenation of adjacency lists of vertices in $\text{Front}(t-1)$.
(9) Remove duplicates from $\text{Nbr}(t)$.
(10) $\text{Front}(t)$ = vertices in $\text{Nbr}(t)$ not in $\text{Front}(t-1) \cup \text{Front}(t-2)$.
(11) **foreach** $v \in \text{Front}(t)$
(12) $L[v] = l$.
(13) Remove v from U.
(14) $t = t + 1$.
(15) **endwhile**
(16) Output the array L.

Finally, steps 6, 12 and 13 take time $O(1)$ each, and so, contribute at most $O(V)$ to the total running time. The total time is thus $O(\frac{E}{V}\,\text{Sort}(V) + V)$.

5.2 Preprocessing

The preprocessing phase(Algorithm 2) is based on a PRAM algorithm for computing connected components due to Cole and Vishkin [8]. This in turn uses an idea from the algorithm due to Chin et al [7] (lines 6–9). The basic step in Chin et al's algorithm (originally due to Hirschberg) is to (a) identify a group of vertices as being in the same connected component (b) identify a leader vertex from the group, (c) move edges incident on every vertex in the group to the leader. In the preprocessing phase, we simply repeat this step several times while keeping track of the leader for each vertex. Clearly if we now find a labelling for the graph induced by the remaining leaders we can construct a labelling for all the vertices.

The following terminology is useful. A leader is *passive* if it spans a complete component, else it is *active*. The *size* of a leader is the number of vertices having it as their leader. Edges connecting distinct leaders are *active*, otherwise passive.

We first estimate the number of active leaders at the end of Algorithm 2. Consider any single component. A single iteration of Chin et al's algorithm (steps 6–9) halves the number of active leaders [7]. Thus $\log d$ iterations reduce the number of active leaders by a factor $d = \sqrt{n_i}$ (if the number of leaders reduced to 1, then the leader is inactive– in fact the leader is itself the label for the component). Thus we know that $n_{i+1} \geq dn_i = n_i^{3/2}$. Given that $n_1 \geq 2$, we see that if some leader is active after k iterations of the outer loop (steps 3–11) its size must be at least $2^{1.5^k}$. For $k = \log_{1.5} \log VBD/E$ this is VBD/E. Thus the number of active leaders is no more than E/BD, as claimed.

Algorithm 2: Preprocessing
(1) Make each vertex its own leader: for each vertex v set $L(v) = v$.
(2) **for** $i = 1$ **to** $\log_{1.5} \log \frac{VBD}{E}$
(3) Let n_i be the size of the smallest size active leader. If $n_i = 1$, then **exit**.
(4) For each active leader, choose upto $d = \sqrt{n_i}$ active edges. If fewer than d active edges are available, choose all of them. From these edges form a graph G_i on the active leaders.
(5) **for** $j = 1$ **to** $\log d$
(6) For each active leader v set $L(v) =$ smallest neighbour of v in G_i. This partitions active leaders into pseudotrees.
(7) Use pointer doubling to identify the smallest numbered supervertex $\min(T)$ in each pseudotree T.
(8) For each v in each pseudotree T set $L(v) = \min(T)$.
(9) Replace each active edge (u, v) in G_i with an edge $(L(u), L(v))$.
(10) **endfor**
(11) Replace each active edge (u, v) in G with an edge $(L(u), L(v))$.
(12) **endfor**

Step 1 clearly takes time $\text{Scan}(V)$. For step 3, the size of each vertex can be computed using a histogram operation (see section 3.2). To determine if a vertex is active, we simply need to know if there are nontrivial (non self loop) edges incident on it. This can be done by a suitable duplicate elimination problem. For step 4, we first number the edges incident on each vertex (done using multiprefix of section 3.2), and collect together those with number at most d. We also collect together the relevant entries from the array L, to prepare the data structures of graph G_i for the loop in lines 5–10. Step 11 is just a table lookup, and can be done using the ideas of section 3.2. All these operations can be done in time $O(\frac{E}{V}\,\text{Sort}(V))$.

To estimate the time for steps 6–9, let V_i and E_i respectively be the number of vertices and edges in G_i. Steps 7–8 are analyzed by Chiang et al [2], who show that these can be done in time $O(\mathrm{Sort}(V_i))$.[2] Step 6 can be done by a multiprefix, step 9 is similar to step 12. Both these can be done in time $O(\frac{E_i}{V_i}\,\mathrm{Sort}(V_i))$. Noting that $E_i \le V_i\sqrt{n_i}$ and $V_i \le V/n_i$, the time for a single iteration of lines 6–9 is $O(\sqrt{n_i}\,\mathrm{Sort}(V_i)) = O(\frac{\mathrm{Sort}(V)}{\sqrt{n_i}})$. Thus the time for the entire loop 5–10 is clearly $O(\mathrm{Sort}(V))$.

Thus the time for a single iteration of lines 3–11 is $O(\frac{E}{V}\,\mathrm{Sort}\,V)$.

5.3 Final Algorithm

The total time for preprocessing is thus $O(\max(1, \log\log\frac{VBD}{E})\frac{E}{V}\,\mathrm{Sort}\,V)$. The number of vertices left after preprocessing is clearly $\min(V, \frac{E}{BD})$. When BFS is run on this the time required is $O(\frac{E}{V}\,\mathrm{Sort}\,V + \frac{E}{BD}) = O(\frac{E}{V}\,\mathrm{Sort}(V))$. Thus the time is $O(\max(1, \log\log\frac{VBD}{E})\frac{E}{V}\,\mathrm{Sort}\,V)$.

6 Open Problems

Our lower bound proof assumes input in the edge-list format, and also assumes a comparison based model. It would be nice to see if the bound holds when these restrictions are removed. One approach could be to use the *External Memory Turing Machine* model defined in [9].

There is still a small gap between the upper and lower bounds for connectivity. We consider finding an optimal connectivity algorithm a challenging open problem.

It would also be interesting to see if the lower bound techniques developed here can be used in proving lower bounds for other data models, like the *Data Stream* model [10].

References

[1] Jeffery Vitter, and Elizabeth Shriver, *Optimal Algorithms for Parallel Memory I: Two-Level Memories*, Algorithmica **12**(2–3), 1994, pp. 110–147.

[2] Yi-Jen Chiang, Michael Goodrich, Edward Grove, Roberto Tamassia, Darren Vengroff, and Jeff Vitter, *External-Memory Graph Algorithms*, in Proceedings of the Sixth SIAM-ACM Symposium on Discrete Algorithms, 1995, pp. 139–149.

[3] Vijay Kumar and Eric Schwabe, *Improved Data Structures and Algorithms for Solving Graph Problems in External Memory*, in Proceedings of the 8th IEEE Symposium on Parallel and Distributed Processing, 1996, pp. 169–176.

[4] James Abello, Adam Buchsbaum, and Jeffery Westbrook, *A Functional Approach to External Graph Algorithms*, in Proc. 6th ESA, 1998.

[5] Lars Arge, Mikael Knudsen, and Kirsten Larsen, *A General Lower Bound on the I/O-Complexity of Comparison-based Algorithms*, in Proceedings of the Third Workshop on Algorithms and Data Structures, 1993.

[6] Abhiram Ranade, Sandeep Bhatt, and Lennart Johnsson, *The Fluent Abstract Machine*, In Proceedings of the Fifth MIT Conference on Advanced Research in VLSI, March 1988, pp. 71–94.

[7] F. Y. Chin, J. Lam, and I. Chen, *Efficient Parallel Algorithms for some Graph Problems*, Communications of the ACM, **25**(9), 1982, pp. 659–665

[8] Richard Cole and Uzi Vishkin, *Approximate Parallel Scheduling Part II: Applications to Optimal Parallel Graph Algorithms in Logarithmic Time*, Information and Computation, **92**(1), 1991, pp. 1–47.

[9] Lars Arge and Peter Bro Miltersen, *On Showing Lower Bounds for External-memory Computational Geometry Problems*, in External Memory Algorithms and Visualization, James Abello and Jeffery Vitter(Eds.), American Mathematical Society, 1998.

[10] Monica Henzinger, Prabhakar Raghavan, and Sridar Rajagopalan, *Computing on Data Streams*, SRC Techical Note 1998-011, Digital Systems Research Center.

[2] Actually, Chiang et al's connected algorithm is simply $\log V/M$ iterations of this loop for the original graph G. We could use their analysis completely, but our analysis is slightly tighter.