# ENGINEERING AN EXTERNAL MEMORY MINIMUM SPANNING TREE ALGORITHM*

Roman Dementiev, Peter Sanders, Dominik Schultes
*Max-Planck-Institut f. Informatik, 66123 Saarbrücken, Germany*
{dementiev,sanders}@mpi-sb.mpg.de,mail@dominik-schultes.de

Jop Sibeyn
*Universität Halle, Institut für Informatik, 06099 Halle, Germany*
jopsi@informatik.uni-halle.de

**Abstract** We develop an external memory algorithm for computing minimum spanning trees. The algorithm is considerably simpler than previously known external memory algorithms for this problem and needs a factor of at least four less I/Os for realistic inputs.

Our implementation indicates that this algorithm processes graphs only limited by the disk capacity of most current machines in time no more than a factor 2–5 of a good internal algorithm with sufficient memory space.

**Keywords:** secondary memory, random permutation, time forward processing, external priority queue, external graph algorithm

## 1 Introduction

The high capacity and low price of hard disks makes it increasingly attractive to process huge data sets using cheap PC hardware. However, the large access latency of such mechanical devices requires the design of external memory algorithms that achieve high locality of access. A simple and successful model for external memory assumes a limited fast memory of size $M$ and a large memory that can be accessed in consecutive blocks of size $B$ in one I/O step [2].

While simple algorithmic problems like sorting have very efficient external algorithms, even simple graph problems are quite difficult to solve for general graphs. For example, depth first search has no efficient external solution. Refer to [14, Chapters 3–5] for an overview. One of the most important exceptions is the minimum spanning tree (MST) problem: Consider an undirected connected graph $G$ with $n$ nodes and $m$ edges. Edges have nonnegative weights. A minimum spanning tree

*This work was partially supported by DFG grant SA 933/1-1 and the IST Programme of the EU under contract number IST-1999-14186 (ALCOM-FT).

of $G$ is a subset of edges with minimum total weight that forms a spanning tree of $G$. If the graph is not connected, most algorithms are easily adapted to find a *minimum spanning forest* (MSF), i.e., a minimum spanning tree of each connected component. The MST problem can be solved in $\mathcal{O}(\mathrm{sort}(m))$ expected I/O steps [1] where $\mathrm{sort}(N) = \mathcal{O}(N/B \log_{M/B} N/B)$ denotes the number of I/O steps required for external sorting [2]. Section 3 gives more details on previous work. We are not aware of any implementations of external MST algorithms. One reason may be that even the simplest previous I/O efficient MST algorithms turn out to be quite complicated to implement. In the full paper we take a more detailed look at some implementation details of previous algorithms and the resulting I/O overheads.

In this paper we describe the design, analysis, implementation, and experimental evaluation of a very simple randomized algorithm for external memory minimum spanning trees.

We begin in Section 4 with a discussion of *semi-external* algorithms that are applicable if $n = \mathcal{O}(M)$, i.e., there is enough internal memory to store a constant number of words for each node. We choose a simple adaptation of Kruskal's algorithm [1] that needs only a single machine word for each node.

If $n > M$, all known external algorithms reduce the number of nodes by *contracting* MST edges: If $e = (u, v) \in E$ is known to be an MST edge, we can remove $u$ from the problem by outputting $e$ and identifying $u$ and $v$, e.g., by removing node $u$ and renaming an edge of the form $(u, w)$ to a new edge $(v, w)$. By remembering where $(v, w)$ came from, we can reconstruct the MST of the original graph from the MST of the smaller graph. Our main algorithmic innovation is a very simple randomized node reduction algorithm that removes one node at a time from the graph. Section 5 develops this idea from an abstract algorithm over an external realization using priority queues to a bucket based implementation that reduces internal overhead. Besides being simpler and faster than previous node reduction algorithms, our algorithm needs to store each edge only once, whereas previous algorithms store an edge $\{u, v\}$ twice, once as $(u, v)$ and once as $(v, u)$.

The semiexternal algorithm from Section 4 and the node reduction from Section 5 can be combined to an external MST algorithm with expected I/O complexity $\mathcal{O}(\mathrm{sort}(m) \lceil \log(n/M) \rceil)$. This seems to be inferior by a factor of $\log(n/M)$ to the best previous algorithms. However, in Section 2 we argue that $n/M \leq 16$ for any problem that runs on a "well balanced" machine. Hence, $\log(n/M)$ will be a small constant. A comparison with previous algorithms in the full paper indicates that for all such inputs our algorithm uses at least a factor four less I/Os than all previous algorithms. Moreover, if $n/M$ should really get large, our node reduction algorithm could be used to speed up asymptotically better algorithms by a similar constant factor. For graphs that are *sparse under edge contraction* in the sense of [6] (e.g., planar graphs or graphs with bounded tree width), our algorithm achieves asymptotically optimal performance of $\mathcal{O}(\mathrm{sort}(m))$ I/Os.

In Section 7 we report about an implementation using `<stxxl>`,[1] an external implementation of the C++ STL library. Using a PC and 4 cheap disks, the implementation

[1]`http://www.mpi-sb.mpg.de/~rdementi/stxxl.html`

can solve instances with up to $2^{32}$ nodes using about $5\mu$s per edge. (About $2.5\mu$s per edge when the semi-external algorithm suffices.) The best internal algorithm for very sparse graphs — Kruskal's algorithm — needs about 1–1.5 $\mu$s per edge for the largest inputs our machine can handle.

## 2 "Realistic" Input Sizes

In the past few years, the cost ratio between main memory and the same amount of hard disk space has consistently been between 100 and 200. Hence, in a balanced system, the ratio between hard disk capacity and main memory size will be of the same order. Let us assume a disk capacity of $128M$. To represent an edge, algorithms based on edge contraction need at least four words to describe the incident nodes, the edge weight, and the original identity of the edge. Hence, the largest graph we may ever want to process on a balanced machine will have $m \approx 128M/4 = 32M$. If we further assume that the sparsest "interesting" graphs have about $2n$ edges we get $n \leq 16M$. A semiexternal implementation of Kruskal's algorithm needs one machine word per node so that we need node reduction by a factor of at most 16. This factor might be up to five times smaller (non-inplace sorting, edge $\{u, v\}$ stored as $(u, v)$ *and* $(v, u)$ in previous algorithms, five words per edge) or larger (somewhat unbalanced machine, even more sparse graphs). However, the complexity of simple external algorithm as ours only depends logarithmically on this factor so that the error is not very big. We have also slightly "tuned" this discussion in favor of previous algorithms. For example, Boruvka's algorithm is most efficient compared to ours if the reduction factor is a power of two.

## 3 Related Work

Boruvka's algorithm [4, 17] was the first MST algorithm. Interestingly it is the basis of most "advanced" MST algorithm. Conceptually, the algorithm is very simple: Assume that all edge weights are different. In a *Boruvka phase*, find the lightest incident edge for each node. The set $C$ of these edges can be output as part of the MST. Now contract these edges, i.e., find a representative node for each connected component of $(V, C)$ and rename an edge $\{u, v\}$ to $\{\mathrm{componentId}(u), \mathrm{componentId}(v)\}$. This routine at least halves the number of nodes.

One Boruvka phase can be implemented externally to run with $\mathcal{O}(\mathrm{sort}(m))$ I/Os [1, 3]. To achieve a node reduction by a factor two, our algorithm needs the same asymptotic I/O complexity. However, a detailed analysis in the full paper [8] shows that our algorithm is both simpler and needs a factor around four less I/Os then the most efficient external realization of a Boruvka phase that we could find [3].

Boruvka's original (internal memory) algorithm repeatedly applies Boruvka phases until only a single node remains. In this paper, when we talk about Boruvka's algorithm as an external algorithm, we assume that only $\mathcal{O}(\log(n/M))$ phases are executed before switching to a semiexternal algorithm as described in Section 4. This choice of base case should probably be considered as folklore.

Boruvka phases are also an ingredient of the asymptotically best internal algorithm [10] that runs in expected linear time. This algorithm additionally contains a component for reducing the number of edges based on random sampling. An external implementation of this approach yields an I/O complexity of $\mathcal{O}(\mathrm{sort}(m + n))$ [1]. The authors also discuss a deterministic, recursive, external implementation of Kruskal's algorithm that works in $\mathcal{O}\big(\mathrm{sort}(m) + \frac{m}{n}\mathrm{sort}(n)\log(n/M)\big)$ I/Os. The base case is a graph with $\mathcal{O}(M)$ edges. The full paper gives more details of these algorithms [8].

Several deterministic external algorithms are described by Arge, Brodal, and Toma [3]. They start with an interesting alternative base case. Rather than reducing the number of nodes until a semiexternal algorithm can be used they make the graph so dense that the average node degree is $B$. Then an external implementation of the Jarník-Prim algorithm [9, 18] takes over that stores edges in a priority queue. The algorithm needs one random I/O for each node but for very dense graphs this I/Os step can be amortized over $B$ edge accesses. We have not used this base case since for current disk technology (a block stores around $2^{16}$ edges) the semiexternal case is reached much earlier than a case with $E/V \geq B$. Although both our algorithm and the external Jarník-Prim algorithm use an edge priority queue, they are quite different. Our algorithm is a node reduction that does little else than priority queue accesses whereas the external Jarník-Prim algorithm is a base case whose limiting factor are random node accesses. The two algorithms also use different priorities. In particular, our algorithm can be modified to use only a single node index for the priority whereas the external Jarník-Prim algorithm needs to compare edge weights. This can translate into a logarithmic factor difference in internal work. The main result in [3] is an algorithm that reduces the number of nodes by a factor $r$ in $\mathcal{O}(\mathrm{sort}(m+n)\log\log r)$ I/Os rather than $\mathcal{O}(\mathrm{sort}(m+n)\log r)$.

## 4 Semi-External Algorithms

The base case of our external MST algorithm is a *semiexternal* algorithm that is applicable once the number of nodes is reduced to $\mathcal{O}(M)$. Abello, Buchsbaum, and Westbrook [1] describe two such algorithms.

The simplest one is an adaptation of Kruskal's algorithm: First sort the edges by weight using external sorting. Then the edges are processed in order of increasing weight. Kruskal's algorithm maintains a minimum spanning forest (MSF) $F$ of the edges seen so far. An edge $\{u, v\}$ is put into $F$ if it joins two components in $F$ and is discarded otherwise. The necessary operations can be implemented very efficiently using a union-find data structure [24] if nodes are numbered $0..n-1$.[2] This data structure can be implemented using a *single* array of integers $a[0..n-1]$. If node $i$ is the representative of its component then $a[i] \geq n$ and $a[i] - n$ is its merging rank. Otherwise $a[i]$ stores an index of another node in the compenent. The pointers of nodes in a component form a tree rooted at the component representative. Since the merging depths reach at most $\lceil \log n \rceil$,[3] a $w$ bit word can represent node indices in the range $0..2^w - w$. For example, using 32 bit words we can represent up to 4 294 967 264 nodes.

The second algorithm needs even less I/Os since it scans the edges in their original, unsorted order. Using dynamic trees [23] it is still possible to maintain the MSF $F$ of the edges seen so far using space $\mathcal{O}(n)$ and time $\mathcal{O}(\log n)$ per edge. However, the

[2]In this paper we use $i..j$ as a shorthand for $\{i, \ldots, j\}$.
[3]In this paper, $\log x$ stands for $\log_2 x$.

constant factors involved make this algorithm not very promising for a practical implementation. Not only are dynamic tree operations much more costly than operations on a union find data structure, but also the savings in I/O volume can be deceptive. For example, the LEDA [13] implementation of dynamic trees needs at least ten times more space for each node than an efficient implementation of the union-find data structure. This means that our algorithm would need $2 \cdot \mathrm{sort}(m) \ln 10$ additional I/Os to reduce the number of nodes sufficiently to make the dynamic tree algorithm applicable.

A scanning based algorithm is still attractive for computing MSTs of fairly dense graphs where the number of nodes is small enough for direct semiexternal treatment. We have not included such graphs into the present study since the I/O aspects of finding MSTs for them are not very interesting. However, it is worth noting that *any* internal MST algorithm with running time $T(n, m)$ can be transformed into a semiexternal MST algorithm that scans the edges once and has internal overhead $\mathcal{O}\big(\frac{m}{n} T(n, \mathcal{O}(n))\big)$: The unsorted edges are processed in batches $C$ of size $\Theta(n)$ and we remember the MSF $F$ of the edges seen so far. In each iteration, we set $F := \mathrm{MSF}(C \cup F)$. In practice, one would use Krukal's algorithm or the Jarník-Prim algorithm. A theoretically interesting observation is that together with the linear time randomized algorithm [10] we get a semiexternal MST algorithm with internal overhead $\mathcal{O}(m + n)$.

## 5 Efficient Node Reduction

Similar to Boruvka's algorithm, our *sweeping algorithm* is based on edge contraction. But the difference is that we identify only one MST edge at a time. The most abstract form of the algorithm is very simple. In each iteration, we remove a random node $u$ from the graph. We find the lightest edge $\{u, v\}$ incident to $u$. By the well known cut-property that underlies most MST algorithms, $\{u, v\}$ must be an MST edge. So, we output $\{u, v\}$, remove it from $E$, and *contract* it, i.e., all other edges $\{u, w\}$ incident to $u$ are replaced by edges $\{v, w\}$. If we store the original identity of each edge, we can reconstruct the MST from the edges that are output.

THEOREM 1 *The expected number of edges inspected by the abstract algorithm until the number of nodes is reduced to $n'$ is bounded by $2m \ln \frac{n}{n'}$.*

**Proof:** In the iteration when $i$ nodes are left (note that $i = n$ in the first iteration), the expected degree of a random node is at most $2m/i$. Hence, the expected number of edges, $X_i$, inspected in iteration $i$ is at most $2m/i$. By the linearity of expectation, the total expected number of edges processed is

$$\sum_{n' < i \leq n} \mathbb{E}[X_i] \leq \sum_{n' < i \leq n} \frac{2m}{i} = 2m \sum_{n' < i \leq n} \frac{1}{i} = 2m \left( \sum_{1 \leq i \leq n} \frac{1}{i} - \sum_{1 \leq i \leq n'} \frac{1}{i} \right)$$
$$= 2m(H_n - H_{n'}) \leq 2m(\ln n - \ln n') = 2m \ln \frac{n}{n'}$$

where $H_n = \ln n + 0.577 \cdots + \mathcal{O}(1/n)$ is the $n$-th harmonic number. ■

As a first step towards an external implementation, we replace random selection of nodes by *sweeping* the nodes in an order fixed in advance. We assume that nodes

```
ExternalPriorityQueue: $Q$
**foreach** $(e = (u, v), c) \in E$ **do** $Q.insert(((\pi(u), \pi(v)), c, e))$       $--$ rename
currentNode := $-1$                                   $--$ node currently being removed
$i$ := $n$                                            $--$ number of remaining nodes
**while** $i > n'$ **do**
    $((u, v), c, e_{\text{old}})$ := $Q.deleteMin()$
    **if** $u \neq$currentNode **then**                 $--$ lightest edge out of a new node
        currentNode := $u$                            $--$ node $u$ is removed
        $i$−−
        relinkTo := $v$
        output $e_{\text{old}}$                          $--$ MST edge
    **elsif** $v \neq relinkTo$ **then** $Q.insert((v, relinkTo), c, e_{\text{old}})$$--$ relink non-self-loops
```

*Figure 1.* An external implementation of the sweeping algorithm using a priority queue.

are numbered $0..n-1$. We first rename the node indices using a random permutation $\pi : 0..n-1 \to 0..n-1$ and then remove renamed nodes in the order $n-1$, $n-2$, $\ldots$, $n'$.

THEOREM 2 *The sweeping algorithm is equivalent to the abstract node reduction algorithm.*

**Proof:** In each iteration, the abstract algorithm can be viewed as fixing one value of a random permutation of node indices. It does that by choosing one of the remaining nodes uniformly at random. This exactly emulates the most commonly used algorithm for generating uniformly distributed random permutations [12]. ■

Note that the sweeping algorithm produces a graph with node indices $0..n'-1$, i.e., it can be directly used as input to our semiexternal Kruskal algorithm from Section 4.

## 5.1 A Priority Queue Implementation

There is a very simple external realization of the sweeping algorithm based on priority queues of edges. Edges are stored in the form $((u, v), c, e_{\text{old}})$ where $(u, v)$ is the edge in the current graph, $c$ is the edge weight, and $e_{\text{old}}$ identifies the edge in the original graph. The queue normalizes edges $(u, v)$ in such a way that $u \geq v$. We define a priority order $((u, v), c, e_{\text{old}}) < ((u', v'), c', e'_{\text{old}})$ iff $u > u'$ or $u = u'$ and $c < c'$. With these conventions in place, the algorithm can be described using the simple pseudocode in Figure 1. If $e_{\text{old}}$ is just an edge identifier, e.g. a position in the input, an additional sorting step at the end can extract the actual MST edges. If $e_{\text{old}}$ stores both incident vertices, the MST edge and its weight can be output directly.

THEOREM 3 *The sweeping algorithm can be implemented to work with* $\mathcal{O}(\lceil m'/m \rceil \operatorname{sort}(m))$ *I/Os if it processes* $m'$ *edges during its execution. It processes the same number of edges as the abstract algorithm from Theorem 1.*

**Proof:** Renaming using a random permutation can be done using $\mathcal{O}(\mathrm{sort}(n+m))$ I/Os (e.g. [19]).[4] The algorithm performs only $m+m'$ insertions and the queue size never exceeds $m$. External priority queues can be implemented to do this using $\frac{m+m'}{m}\mathrm{sort}(m) = \mathcal{O}(\lceil m'/m \rceil \,\mathrm{sort}(m))$ I/Os [5]. Outputting the MST edges takes $\mathcal{O}(n/B)$ I/Os. ■

## 5.2 A Bucket Implementation

The priority queue implementation unnecessarily sorts the edges adjacent to a node where we really only care about the smallest edge coming first. We now describe an implementation of the sweeping algorithm that has internal work linear in the total I/O volume. We first make a few simplifying assumptions to get closer to our implementation.

The representation of edges and the renaming of nodes works as in the priority queue implementation. As before, in iteration $i$, node $i$ is removed by outputting the lightest edge incident to it and relinking all the other edges. We split the node range $n'..n-1$ into $k = \mathcal{O}(M/B)$ equal sized *external buckets*, i.e., subranges of size $(n-n')/k$ and we define a special external bucket for the range $0..n'-1$. An edge $(u,v)$ with $u > v$ is always stored in the bucket for $u$. We assume that the current bucket (that contains $i$) completely fits into main memory. The other buckets are stored externally with only a write buffer block to accommodate recently relinked edges.

When $i$ reaches a new external bucket, it is distributed to *internal buckets* — one for each node in the external bucket. The internal bucket for $i$ is scanned twice. Once for finding the lightest edge and once for relinking. Relinked edges destined for the current external bucket are immediately put into the appropriate internal bucket. The remaining edges are put into the write buffer of their external bucket. Write buffers are flushed to disk when they become full.

When only $n'$ nodes are left, the bucket for range $0..n'-1$ is used as input for the semi-external Kruskal algorithm from Section 4.

A more general implementation needs a special case for internal buckets that correspond to very high degree nodes. However, although this somewhat complicates the implementation, it will not have a negative effect on running time. On the contrary, nodes with very high degree can be moved to the bucket for the semiexternal case directly. These nodes can be assigned the numbers $n'+1, n'+2, \ldots$ without danger of confusing them with nodes with the same index in other buckets. To accomodate these additional nodes in the semiexternal case, $n'$ has to be reduced by at most $\mathcal{O}(M/B)$ since for $m = \mathcal{O}\big(M^2/B\big)$ there can be at most $\mathcal{O}(M/B)$ nodes with degree $\Omega\,(M)$.

If the overall number of edges gets so large that even an average size external bucket does not fit into internal memory, one has to switch to multi-level distribution schemes. However, the added complexity for this is needed even for sorting so that we remain I/O optimal and work optimal.

[4]In Appendix 1 we give an algorithm that produces pseudorandom permutations directly without additional I/Os.

## 5.3 Parallel Edges and Sparse Graphs

The basic sweeping algorithm described above can produce parallel edges by relinking. These edges remain parallel during subsequent relinking operations. Parallel edges can be removed relatively easily. When scanning the internal bucket for node $i$, the edges $(i, v)$ are put into a hash table using $v$ as a key. The corresponding table entry only keeps the lightest edge connecting $i$ and $v$ seen so far.

This leads to an asymptotic improvement for planar graphs, graphs with bounded tree width and other classes of graphs that remain sparse under edge contraction:

THEOREM 4 *Consider a graph that has $\mathcal{O}(n - i)$ edges after any sequence of $i$ edge contractions. Then the sweeping algorithm with removal of parallel edges runs using $\mathcal{O}(\mathrm{sort}(n))$ I/Os.*

**Proof:** We charge the cost for inspecting (and immediately discarding) a parallel edge to the relinking operation that created the parallel edge. This demonstrates that the algorithm performs only a constant factor more work than an algorithm where parallel edges are not even generated. Since the graph is sparse under edge contraction, $\mathcal{O}(\mathrm{sort}(n))$ I/Os suffice to reduce the number of nodes *and edges* by a factor at least two. Hence, the I/O steps needed for the algorithm obey the recurrence $W(n) \leq \mathcal{O}(\mathrm{sort}(n)) + W(n/2)$. This recurrence has the solution $W(n) = \mathcal{O}(\mathrm{sort}(n))$. ■

## 6 Implementation

Our external implementation makes extensive use of `<stxxl>`, an external implementation of the C++ standard template library STL. The semiexternal Kruskal and the priority queue based sweeping algorithm become almost trivial using external sorting [7] and external priority queues [20]. The bucket based implementation uses external stacks to represent external buckets. The stacks have a single private output buffer and they share a common pool of additional output buffers that facilitates overlapping of output and internal computation. When a stack is switched to reading, it is assigned additional private buffers to facilitate prefetching.

The internal aspects of the bucket implementation are also crucial. In particular, we need a representation of internal buckets that is space efficient, cache efficient, and can grow adaptively. Therefore, internal buckets are represented as linked lists of small blocks that can hold several edges each. Edges in internal buckets do not store their source node because this information is redundant.

Our implementation deviates in three aspects from the previous description. Edges are stored as 5-tuples of 32 bit integers and store both endpoints of the original edge directly. This saves an additional sorting phase at the end for collecting missing information on the MST edges and it allows us to process more then $2^{32}$ edges without resorting to cumbersome packed representations with 40 bit edge-ids. Our implementation of the Union-Find data structure uses a separate byte for the merging rank. We have not implemented the special case treatment for nodes of very high degree outlined in Section 5.2 because this case does not occur for the graph families studied in [15]. We saw also no reason to invent or find such graph families since with the special case treatment we could expect them to be easier to solve than other graphs.

In any case, our priority queue based implementation covers this case and performs reasonably well for a single disk.

A more detailed account of the implementation is given in [21] and on the web `http://www.dominik-schultes.de/emmst/`.

## 7 Experiments

Our starting point for designing experiments was the study by Moret and Shapiro [15]. We have adopted the instance families for *random* graphs with random edge weights and random *geometric* graphs where random points in the unit square are connected to their $d$ closest neighbors. In order to obtain a simple family of planar graphs, we have added *grid* graphs with random edge weights where the nodes are arranged in a grid and are connected to their (up to) four direct neighbors. We have not considered the remaining instance families in [15] because they define rather dense graphs that would be easy to handle semiexternally or they are specifically designed to fool particular algorithms or heuristics. We have chosen the parameters of the graphs so that $m$ is between $2n$ and $8n$. Considerably denser graphs would be either solvable semiexternally or too big for our machine.

The experiments have been performed on a low cost PC-server (around 3000 Euro in July 2002) with two 2 GHz Intel Xeon processors, 1 GByte RAM and $4 \times 80$ GByte disks (IBM 120GXP) that are connected to the machine in a bottleneck-free way (see [7] for more details on the hardware). This machine runs Linux 2.4.20 using the XFS file system. Swapping was disabled. All programs were compiled with g++ version 3.2 and optimization level `-O6`. The total computer time spend for the experiments was about 25 days producing a total I/O volume of several dozen Terabytes.

Figure 2 summarizes the results for the bucket implementation. Tables with detailed numerical data can be found in Appendix 2. The internal implementations were provided by Irit Katriel [11]. The curves only show the internal results for random graphs — at least Kruskal's algorithm shows very similar behavior for the other graph classes. Our implementation can handle up to 20 million edges. Kruskal's algorithm is best for very sparse graphs ($m \leq 4n$) whereas the Jarník-Prim algorithm (with a fast implementation of pairing heaps) is fastest for denser graphs but requires more memory. For $n \leq$ 160 000 000, we can run the semiexternal algorithm and get execution times within a factor of two of the internal algorithm.[5] The curves are almost flat and very similar for all three graph families. This is not astonishing since Kruskal's algorithm is not very dependent on the structure of the graph. Beyond 160 000 000 nodes, the full external algorithm is needed. This immediately costs us another factor of two in execution time: We have additional costs for random renaming, node reduction, and a blowup of the size of an edge from 12 bytes to 20 bytes (for renamed nodes). For random graphs, the execution time keeps growing with $n/M$ as predicted by the upper bound from Theorem 1.

[5]Both the internal and the semiexternal algorithm have a number of possibilities for further tuning (e.g., using integer sorting or a better external sorter for small elements). But none of these measures is likely to yield more than a factor of 2.

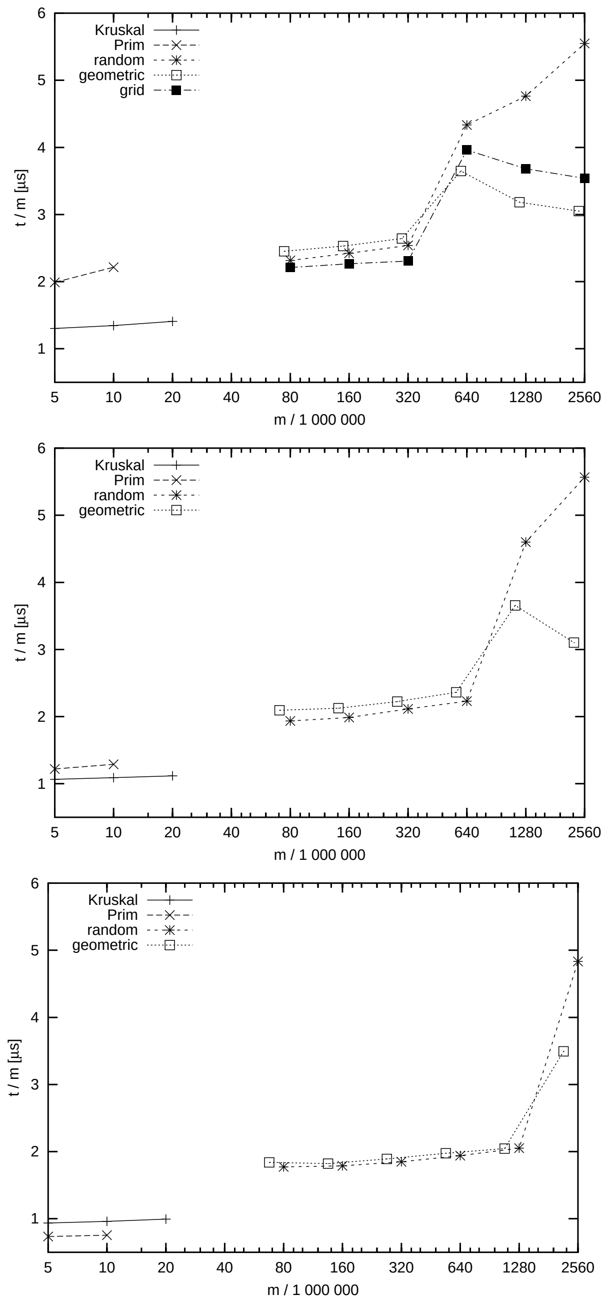

*Figure 2.* Execution time per edge for $m \approx 2 \cdot n$ (top),$m \approx 4 \cdot n$ (center), $m \approx 8 \cdot n$ (bottom).

The behavior for grid graphs is much better than predicted by Theorem 4. It is interesting that similar effects can be observed for geometric graphs. This is an indication that it is worth removing parallel edges for many nonplanar graphs.[6] Interestingly, the time per edge *decreases* with $m$ for grid graphs and geometric graphs. The reason is that the time for the semiexternal base case does not increase proportionally to the number of input edges. For example, $5.6 \cdot 10^8$ edges of a grid graph with $640 \cdot 10^6$ nodes survive the node reduction, and $6.3 \cdot 10^8$ edges of a grid graph with twice the number of edges.

Another observation is that for $m = 2560 \cdot 10^6$ and random or geometric graphs we get the worst time per edge for $m \approx 4n$. For $m \approx 8n$, we do not need to run the node reduction very long. For $m \approx 2n$ we process less edges than predicted by Theorem 1 even for random graphs simply because one MST edge is removed for each node.

We have made a few runs with even larger graphs. The largest one was a grid graph with $n = 2^{32}$ which takes 96GByte just to represent the input. Even this graph that required an I/O volume of about 830 GByte was processed in about 8h 40min.

The following small table shows running time in $\mu$s per edge for random graphs with $n = 320 \cdot 10^6$ and $m = 640 \cdot 10^6$ where we varied the number of disks and where we compare the priority queue implementation with the bucket implementation:

| | 1 disk | 4 disks |
|---|---|---|
| bucket implementation | 6.7 | 4.3 |
| priority queue implementation | 11.0 | 8.9 |

Since the speedup for the bucket algorithm after quadrupling the number of disks is only 1.56, one can conclude that even with a single disk and the internally efficient bucket algorithm, the computation is not I/O-bound. This explains why the bucket implementation brings a considerable improvement over the priority queue implementation. Considering its simplicity, the priority queue implementation is still interesting since it also achieves reasonable performance for a single disk.

## 8 Conclusions

We have demonstrated that massive minimum spanning tree problems filling several hard disks can be solved "overnight" on a PC. The key algorithmic ingredient for this result is the sweeping paradigm that yields simpler and faster algorithms than previous approaches. This paradigm is also useful for other problems like connected components, list ranking, tree rooting, ... [22]. The efficient and relatively simple implementation profits from the `<stxxl>` library that implements external sorting, priority queues, and other basic data structures in an efficient way using parallel disks, overlapping of I/O and computation, DMA directly to user space, ...

An interesting challenge for the future is whether we can solve even larger MST problems using parallel processors and external memory together. Here, the sweeping paradigm seems to break down and other simplifications of existing algorithms are sought for.

[6] Very few parallel edges are generated for random graphs. Therefore, switching off duplicate removal gives about 13 % speedup for random graphs compared to the numbers given.

## Acknowledgments

We would like to thank Irit Katriel for providing the internal implementations and some of the graph generators. Ulrich Meyer was very helpful navigating previous work.

## References

[1] J. Abello, A. Buchsbaum, and J. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002.

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

[3] L. Arge, G. Brodal, and L. Toma. On external memory MST, SSSP and multi-way planar graph separation. In *7th Scandinavian Workshop on Algorithm Theory*, volume 1851 of *LNCS*, pages 433–447. Springer, 2000.

[4] O. Boruvka. O jistém problému minimálním. *Pràce, Moravské Prirodovedecké Spolecnosti*, pages 1–58, 1926.

[5] Gerth Stølting Brodal and Jyrki Katajainen. Worst-case efficient external-memory priority queues. In *6th Scandinavian Workshop on Algorithm Theory*, number 1432 in LNCS, pages 107–118. Springer Verlag, Berlin, 1998.

[6] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proceedings of the Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 139–149, 1995.

[7] R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 138–148, San Diego, 2003.

[8] R. Dementiev, P. Sanders, D. Schultes, and J. Sibeyn. Engineering an external memory minimum spanning tree algorithm — full paper. `http://www.mpi-sb.mpg.de/~sanders/papers/emstfull.ps.gz`, 2004.

[9] V. Jarník. O jistém problému minimálním. *Práca Moravské Přírodovědecké Společnosti*, 6:57–63, 1930. In Czech.

[10] D. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. *J. Assoc. Comput. Mach.*, 42:321–329, 1995.

[11] I. Katriel, P. Sanders, and J. L. Träff. A practical minimum spanning tree algorithm using the cycle property. In *11th European Symposium on Algorithms (ESA)*, number 2832 in LNCS, pages 679–690. Springer, 2003.

[12] D. E. Knuth. *The Art of Computer Programming — Seminumerical Algorithms*, volume 2. Addison Wesley, 2nd edition, 1981.

[13] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[14] U. Meyer, P. Sanders, and J. Sibeyn, editors. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS Tutorial*. Springer, 2003.

[15] B.M.E. Moret and H.D. Shapiro. An empirical assessment of algorithms for constructing a minimum spanning tree. *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 15:99–117, 1994.

[16] M. Naor and O. Reingold. On the construction of pseudorandom permutations: Luby-Rackoff revisited. *Journal of Cryptology: the journal of the International Association for Cryptologic Research*, 12(1):29–66, 1999.

[17] J. Nešetřil, H. Milková, and H. Nešetřilová. Otakar boruvka on minimum spanning tree problem: Translation of both the 1926 papers, comments, history. *DMATH: Discrete Mathematics*, 233, 2001.

[18] R. C. Prim. Shortest connection networks and some generalizations. *Bell Systems Technical Journal*, pages 1389–1401, November 1957.

[19] P. Sanders. Random permutations on distributed, external and hierarchical memory. *Information Processing Letters*, 67(6):305–310, 1998.

[20] P. Sanders. Fast priority queues for cached memory. In *ALENEX '99, Workshop on Algorithm Engineering and Experimentation*, number 1619 in LNCS, pages 312–327. Springer, 1999.

[21] D. Schultes. External memory minimum spanning trees. Bachelor thesis, Max-Planck-Institut f. Informatik and Saarland University, `http://www.dominik-schultes.de/emmst/`, August 2003.

[22] J. F. Sibeyn. External connected components. In *12th Scandinavian Workshop on Algorithm Theory*, Springer LNCS, 2004. to appear.

[23] D. D. Sleator and R. E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

[24] R. E. Tarjan. Efficiency of a good but not linear set merging algorithm. *Journal of the ACM*, 22:215–225, 1975.

## Appendix

## 1 Fast Pseudo Random Permutations

For renaming nodes, we need a (pseudo)random permutation $\pi : 0..n-1 \rightarrow 0..n-1$. Assume for now that $n$ is a square so that we can represent a node $i$ as a pair $(a,b)$ with $i = a + b\sqrt{n}$. Our permutations are constructed from *Feistel* permutations, i.e., permutations of the form $\pi_f((a,b)) = (b, a + f(b) \bmod \sqrt{n})$ for some random mapping $f : 0..\sqrt{n}-1 \rightarrow 0..\sqrt{n}-1$. Since $\sqrt{n}$ is small, we can afford to implement $f$ using a lookup table filled with random elements. For example, for $n = 2^{32}$ the lookup table for $f$ would require only 128 KByte. It is known that a permutation $\pi(x) = \pi_f(\pi_g(\pi_h(\pi_l(x))))$ build by chaining four Feistel permutations is "pseudorandom" in a sense useful for cryptography. The same holds if the innermost and outermost permutation is replaced by an even simpler permutation [16]. In our implementation we use just two stages of Feistel-Permutations. It is an interesting question what provable performance guarantees for the sweep algorithm or other algorithmic problems can be given for such permutations.

A permutation $\pi'$ on $0..\lceil\sqrt{n}\rceil^2 - 1$ can be transformed to a permutation $\pi$ on $0..n-1$ by iteratively applying $\pi'$ until a value below $n$ is obtained. Since $\pi'$ is a permutation, this process must eventually terminate. If $\pi'$ is random, the expected number of iterations is close to 1 and it is unlikely that more than three iterations are necessary for *any* input.

## 2 Detailed Measurement Data

*Table A.1.* (Semi-)External test cases. $n$: nodes, $m$: edges, $t$: elapsed time, $p$: processed edges, $E(p)$: expected value of $p$ according to Theorem 1, $d$: duplicates removed.

| *type* | $n/10^6$ | $m/10^6$ | $t[s]$ | $t/m[\mu s]$ | $p/10^6$ | $p/E(p)$ | $d/m$ |
|---|---|---|---|---|---|---|---|
| grid | 40 | 80 | 177 | 2.21 | | | |
| grid | 80 | 160 | 362 | 2.27 | | | |
| grid | 160 | 320 | 738 | 2.31 | | | |
| grid | 320 | 640 | 2 535 | 3.96 | 750 | 85 % | 4 % |
| grid | 640 | 1 280 | 4 712 | 3.68 | 2 492 | 70 % | 13 % |
| grid | 1 280 | 2 560 | 9 056 | 3.54 | 6 167 | 58 % | 22 % |
| random | 40 | 80 | 185 | 2.32 | | | |
| random | 80 | 160 | 388 | 2.42 | | | |
| random | 160 | 320 | 813 | 2.54 | | | |
| random | 320 | 640 | 2 773 | 4.33 | 766 | 86 % | 0 % |
| random | 640 | 1 280 | 6 098 | 4.76 | 2 752 | 78 % | 0 % |
| random | 1 280 | 2 560 | 14 202 | 5.55 | 7 676 | 72 % | 0 % |
| random | 20 | 80 | 155 | 1.94 | | | |
| random | 40 | 160 | 318 | 1.99 | | | |
| random | 80 | 320 | 676 | 2.11 | | | |
| random | 160 | 640 | 1 427 | 2.23 | | | |
| random | 320 | 1 280 | 5 889 | 4.60 | 1 651 | 93 % | 0 % |
| random | 640 | 2 560 | 14 248 | 5.57 | 6 284 | 89 % | 0 % |
| random | 10 | 80 | 142 | 1.77 | | | |
| random | 20 | 160 | 286 | 1.79 | | | |
| random | 40 | 320 | 591 | 1.85 | | | |
| random | 80 | 640 | 1 242 | 1.94 | | | |
| random | 160 | 1 280 | 2 627 | 2.05 | | | |
| random | 320 | 2 560 | 12 370 | 4.83 | 3 426 | 97 % | 0 % |
| geometric | 40 | 75 | 183 | 2.45 | | | |
| geometric | 80 | 149 | 377 | 2.53 | | | |
| geometric | 160 | 298 | 787 | 2.64 | | | |
| geometric | 320 | 596 | 2 175 | 3.65 | 644 | 78 % | 7 % |
| geometric | 640 | 1 190 | 3 797 | 3.18 | 1 949 | 59 % | 13 % |
| geometric | 20 | 71 | 148 | 2.09 | | | |
| geometric | 40 | 141 | 300 | 2.13 | | | |
| geometric | 80 | 282 | 627 | 2.22 | | | |
| geometric | 160 | 564 | 1 333 | 2.36 | | | |
| geometric | 320 | 1 130 | 4 126 | 3.66 | 1 275 | 82 % | 18 % |
| geometric | 10 | 68 | 124 | 1.84 | | | |
| geometric | 20 | 135 | 246 | 1.82 | | | |
| geometric | 40 | 270 | 511 | 1.89 | | | |
| geometric | 80 | 540 | 1 067 | 1.98 | | | |
| geometric | 160 | 1 080 | 2 209 | 2.04 | | | |