# External Memory Minimum Spanning Trees

Dominik Schultes

August 2003

Bachelor-Arbeit

Fachrichtung 6.2 – Informatik, Universität des Saarlandes
angefertigt unter Betreuung von Priv. Doz. Dr. Peter Sanders, Max-Planck-Institut für Informatik

Hiermit versichere ich, dass ich diese Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe.

Saarbrücken, im August 2003

**Abstract**

While in the last years much theoretical work about external memory minimum spanning trees was done, the practical realization of the designed algorithms was neglected. It is the goal of my Bachelor thesis to fill this gap, i.e., we will show that the computation of minimum spanning trees of very large graphs is possible efficiently not only in theory but also in practice.

# Contents

# Chapter 1

# Introduction

## 1.1 Minimum Spanning Trees

Finding a minimum spanning tree is the graph theoretical notation of a quite natural problem: we want to connect several objects — not necessarily directly — with each other and the total costs should be as low as possible. As example we could take a computer network: several computers have to be linked so that every pair of computers can communicate with each other regardless of possible intermediate stations. The expense of the used wires should be minimal.

In graph theory the problem can be formalized easily. In a connected, undirected and weighted graph $G = (V, E)$ with $n$ vertices and $m$ edges a spanning tree $T = (V, F)$ is a connected and acyclic subgraph of $G$ that contains all nodes of $G$. The weight of a spanning tree is the sum of the weights of all edges: $w(T) = \sum_{e \in F} w(e)$. A minimum spanning tree (MST) is a spanning tree with minimum weight.

The problem of finding a MST is interesting because of several reasons. First, it appears in praxis. Second, the problem itself is interesting from a theoretical point of view, and third, the solution of it can be used in order to solve other problems in graph theory, for instance, to find an approximate solution for the travelling salesperson problem.

There are two well known algorithms that can be used to detect a MST in $O(n^2)$ (if the input is given as adjacency matrix) resp. in $O(m \log m)$ time (if the input is given as adjacency lists), Prim's algorithm and Kruskal's algorithm.

## 1.2 External Memory Model

In the main, our considerations refer to a simple model that focuses on two levels of the memory hierarchy, the internal memory (main memory) and the external memory (hard disks). Due to physical reasons (the access head has to find the right position) a disk access takes comparatively much time, the transfer itself plays a tangential role [MSS03, p. 3]. Hence, it is reasonable to deal with blocks of data instead of single items. The *internal memory size* is denoted as $M$ and the *block size* as $B$ [AV88].

The main reason why we have to use external memory algorithms is the simple fact that we do not possess enough internal memory since main memory is $C$ times more expensive than hard disks; we assume that $C \approx 200$ and that consequently a balanced system consists of internal and external memory in the ratio of 1:200.

## 1.3 External Memory Minimum Spanning Trees

The External Memory MST problem deals with very large graphs, i.e., the graphs are so large that not all information which are required for the computations fit in internal memory. Finding a minimum spanning tree of such a graph is more difficult than the internal case since you have to ensure that you take advantage of both spacial and temporal locality as much as possible. Neither Prim's nor Kurskal's algorithm has been designed with this requirement in mind. Hence, in the worst case the processing of each node resp. edge requires one external memory access. Therefore, we need a different approach.

# Chapter 2

# Dense Graphs

If we deal with large dense graphs, we are confronted with the following situation: there are so many edges that they do not fit in internal memory so that we cannot apply an internal algorithm — but there are so few nodes that we can afford some internal memory for each node, in other words, the internal memory is in $O(n)$. Generally, this is the precondition for a semi-external algorithm. In our case, we can use a semi-external version of Kruskal's algorithm [Ski98] in order to find a minimum spanning tree of a dense graph. We sort the list of edges (by weight) in external memory and apply Kruskal's algorithm so that the sorted edges have to be scanned once. Hence, we need $O(\text{sort}(m)) + O(\text{scan}(m)) = O(\text{sort}(m))$ I/Os.

A union-find data structure is essential for Kruskal's algorithm. Basically, we need for each node a reference to the parent node. Additionally, we want for each canonical node of a set (= root of a tree) a rank that is used to perform the union operations in such a way that the trees do not degenerate. Thus, the size of the union-find data structure depends only linearly on the number of nodes so that it can be kept in internal memory on the above mentioned precondition.

In order to keep the memory usage per node small, we store only the height of a tree (instead of the size) as rank. Consequently, we use the union-by-height strategy [OW96], i.e., when we unite two trees, we make the shorter one a subtree of the taller one; if both trees have the same height, an arbitrary one becomes the subtree of the other one, whose height is increased. This strategy guarantees that the height of each tree does not exceed $\log n$ so that $\log \log n$ bits are sufficient to store the rank. After each find operation path compression is applied. (It is possible that the height of one tree is reduced due to path compression. In this case the stored height is *not* adjusted. The expense of a correction would exceed the advantage of an exact value by far.)

Usually a dense graph is expected to consist of $O(n^2)$ edges. As we want to use the distinction between "dense" and "sparse" in order to specify the applicability of the semi-external algorithm, we consider graphs with less edges as dense, too. With this in mind, the boundary between "dense" and "sparse" can be computed with the help of the estimation of Section 1.2 that a balanced system consists of internal and external memory in the ratio of 1:$C$. Principally, we can process only graphs that fit in external memory. If the edges are given as a list, we have to store for each edge the source vertex, the target vertex and the weight. (To store the edges in adjacency lists or in an adjacency matrix would require more memory.) In order to keep this calculation simple, we assume that a node identifier, a weight and a rank in the union-find data structure needs one memory unit each. Hence, we need $3m$ memory units to store the graph, while the size of the external memory is $C \cdot M$. This leads to $\frac{3}{C}m \leq M$. For each node two memory units are allocated in the union-find data structure, a reference to the parent node and the rank. As we want to treat the case that the union-find data structure does not fit in internal memory, we obtain the constraint that $2n > M$. If we combine these inequalities, we get $\frac{m}{n} < \frac{2}{3}C$. Hence, we consider a graph with at least $\frac{2}{3}Cn$ edges as dense and a graph with less edges as sparse. Furthermore, we can state that dense graphs (within this scope) can be processed by our semi-external algorithm.

Actually, a more sophisticated calculation could draw the line even at a smaller average vertex degree.

# Chapter 3

# Sparse Graphs

## 3.1 General Approach

If we deal with large sparse graphs, we can keep neither the nodes nor the edges in internal memory. Hence, the semi-external version of Kruskal's algorithm introduced in chapter 2 cannot be applied directly so that we need an external algorithm. Our basic goal is *to reduce the number of nodes* of the original graph $G$ until the union-find data structure of the remaining nodes (graph $G'$) fits in internal memory so that we can apply the algorithm of chapter 2 to $G'$. During the node reduction phase we obtain a part of a minimum spanning tree of $G$, which is combined with a minimum spanning tree of $G'$ in order to get a complete MST of $G$. The following lemma [JaJ92, p. 223] provides the foundation in order to achieve this goal.

**Lemma 1.** *Let $V = \bigcup V_i$ be an arbitrary partition of $V$ with the corresponding subgraphs $G_i = (V_i, E_i)$, where $1 \le i \le t$. For each $i$, there exists one minimum-weight edge $e_i$ connecting a vertex in $V_i$ to a vertex in $V \setminus V_i$ that belongs to a minimum spanning tree of the graph $G = (V, E)$.*

*Proof.* For an arbitrary subset $V_i \subset V$, let the set $R_i := \{(r, s) \in E \mid r \in V_i \wedge s \in V \setminus V_i\}$ contain all edges that lead from one vertex in $V_i$ to a vertex out of $V_i$, and let the set $S_i \subseteq R_i$ contain the shortest edges of $R_i$, i.e., $S_i := \{e \in R_i \mid w(e) = \min_{e' \in R_i} w(e')\}$. We assume that there is a minimum spanning tree $T = (V, F)$ with a subset $V_i \subset V$ so that $F \cap S_i = \emptyset$, i.e., $T$ does *not* contain any of the shortest edges that leave $V_i$.

Let $(u', v') \in S_i$ be an arbitrary shortest edge that connects $V_i$ with $V \setminus V_i$. We add $(u', v')$ to $F$ and obtain a cycle that contains $(u', v')$ and a different edge $(u, v) \in R_i \setminus S_i$, i.e., an edge that leaves $V_i$, but does *not* belong to the shortest ones. We remove $(u, v)$ and get another spanning tree $T'$ with $w(T') < w(T)$ since $w((u', v')) < w((u, v))$. This contradicts the assumption that $T$ is a minimum spanning tree. □

## 3.2 Boruvka's Algorithm

The most known node reduction algorithm that uses Lemma 1 is *Boruvka*'s. At the beginning the partition of $G = (V, E)$ consists of single nodes. For each node a minimum-weight edge incident to it is found so that the selected edges do not form a cycle. These edges are added to the tree $T = (V, F)$, which initially contains no edges and eventually will be a MST of $G$. The next iteration deals with the partition $V = \bigcup V_i$, where each $V_i$ is a connected component of the subgraph $G' = (V, F)$. Hence, one iteration, called a Boruvka step, consists of the following substeps [Liu01]:

- for each node, find and mark an appropriate minimum-weight edge incident to it;
- determine the connected components formed by the marked edges;
- replace each connected component by a single (super-)vertex, in other words, relabel the edges in such a way that the vertex IDs are replaced with the IDs of the appropriate connected components;
- optionally, eliminate the self-loops and multiple edges created by these contractions.

The algorithm terminates when the remaining graph can be processed by the semi-external algorithm or — if Boruvka's algorithm is used to find a complete MST — when $|F| = n - 1$.

In a basic version of this algorithm, we need $O(\text{sort}(m))$ I/Os for one Boruvka step in order to reduce the number of vertices by a constant factor. $O(\log(n/M))$ steps are required so that the remaining nodes fit in internal memory $M$. This results in $O(\text{sort}(m) \cdot \log(n/M))$ [CGG+95]. A top-down variant with the same I/O complexity is presented in [ABW02].

An improved version [MSS03, p. 80–81] uses $O(\text{sort}(m) \cdot \max\{1, \log\log(nB/m)\})$ I/Os in order to find a MST. This improvement is achieved by combining several steps into supersteps, where each superstep still needs $O(\text{sort}(m))$ I/Os, but reduces more nodes than the basic step. A randomized algorithm, presented in [ABW02], uses $O(\text{sort}(m))$ I/Os in the expected case.

In spite of the good asymptotic behaviour, an implementation of Boruvka's algorithm probably would lead to high constants in the running time. Hence, we will use a different algorithm that has the same general approach.

## 3.3 Sibeyn and Meyer's Algorithm

### 3.3.1 Informal Description

In a way Sibeyn and Meyer's algorithm [SM] is a variant of Boruvka's algorithm. During one step only the minimum-weight edge incident to a node in the last subset $V_t$ is determined and added to the tree instead of finding a shortest edge for each subset $V_i$.

The input is given as a set of $m$ edges $(u, v, c)$, where $u$ is the source vertex, $v$ the target and $c$ the weight. As the graph is undirected an edge $(u, v, c)$ implies that there is also an edge $(v, u, c)$, although it is not listed explicitly. The edges are stored in adjacency lists; each edge is stored only once, namely in the list of the node with the higher identifier; self loops are thrown away since they are irrelevant. The chosen data structure has the feature that only the list of the last node definitely contains all edges incident to it; but that is quite enough as we concentrate on the last node during each step.

In order to get a new partition after each step, the graph is shrunk. The last node is merged with the target vertex of the shortest edge incident to the last node, i.e., the target vertex adopts all edges from the last node and the last node stops existing. Self loops that are created by this action are thrown away. This merging corresponds with the union of the last node and the target vertex to a new subset of $V$ that is part of the partition of the graph. Due to merging edges are relabeled because the former source vertex is replaced with the target vertex of the shortest edge. In order to be able to restore the original endpoints when an edge is added to the MST, the original labels are saved at the beginning so that each adjacency list contains edges $(v, c, e_1, e_2)$, where $v$ is the target, $c$ the weight and $e_1, e_2$ the original endpoints.

### 3.3.2 Pseudo Code

**Input:** a set $E$ of edges $(u, v, c)$ that defines a connected, undirected and weighted graph $G$ with $n$ nodes,
$n'$, the number of nodes that should remain
**Output:** a set $T \subseteq E$ that defines (a part of) a minimum spanning tree of $G$

**let** $\pi$ **be** a random permutation over $\{1, \ldots, n\}$
**foreach** $(u, v, c) \in E$ **do**
  **if** $v < u$ **then** add $(\pi(v), c, u, v)$ to the list of $\pi(u)$
  **else if** $v > u$ **then** add $(\pi(u), c, u, v)$ to the list of $\pi(v)$;

**for** $u = n$ **down to** $n' + 1$ **do**
  traverse all $(v, c, e_1, e_2)$ in the list of $u$ and
  determine the $v$ for which $c$ is minimum;
  add $(e_1, e_2, c)$ to $T$;
  **foreach** $(w, c, e_1, e_2)$ in the list of $u$ **do**
    **if** $w < v$ **then** add $(w, c, e_1, e_2)$ to the list of $v$
    **else if** $w > v$ **then** add $(v, c, e_1, e_2)$ to the list of $w$;

### 3.3.3 Correctness

The correctness follows directly from Lemma 1.

### 3.3.4 Complexity

We assume that the input does not contain any self-loop (self-loops would be eliminated anyway during the first step).

If initially the node indices are randomized (so that we get a uniform distribution), the probability that an arbitrary edge is incident to the last vertex is $\frac{1}{n} + \frac{1}{n} = \frac{2}{n}$ as it is sufficient if one of the endpoints is the last node (the case that both endpoints are the last node cannot occur due to our assumption that there is no self-loop). Hence, the expected number of edges in the list of the last node is $\frac{2}{n}m$.

Since the target vertex of the shortest edge is uniformly distributed, too, we obtain a uniform distribution over the set $\{1, \ldots, n-1\}$ after one reduction step if the edges have been uniformly distributed over $\{1, \ldots, n\}$: the probability that an endpoint of an edge is a certain node $x \in \{1, \ldots, n-1\}$ amounts to $\frac{1}{n} + \frac{1}{n} \cdot \frac{1}{n-1} = \frac{1}{n-1}$, namely the probability according to the assumed uniform distribution over $\{1, \ldots, n\}$ plus the probability that the endpoint had been the last node and was then relabeled to $x$ due to a reduction step.

Using these facts, we can show by induction that the expected number of edges in the list of the currently last node $u \in \{n'+1, \ldots, n\}$ is always less than (or equal to) $\frac{2}{u}m$. Therefore, we obtain [SM]

**Theorem 1.** *For reducing the number of nodes from $n$ to $n'$, the above algorithm processes an expected number of less than $\sum_{u=n'+1}^{n} \left(\frac{2}{u}m\right) \simeq 2 \cdot m \cdot (\ln n - \ln n')$ edges.*

### 3.3.5 Comparison with Boruvka's Algorithm

The advantages of Sibeyn and Meyer's algorithm over Boruvka's algorithm are the following:

- During one Boruvka step $2 \cdot m$ edges are processed in order to reduce the number of nodes by only a factor 2 in the worst case. For a reduction by this factor, the expected number of processed edges of Sibeyn and Meyer's algorithm is less than $2 \cdot m \cdot \ln 2 \simeq 1.39 \cdot m$.

- Relabeling the edges due to shrinking of the graph is very easy in Sibeyn and Meyer's algorithm because the new identifier need not be looked up — in contrast with Boruvka's algorithm, where the new source *and* target vertices of all edges must be looked up. Furthermore, Sibeyn and Meyer's algorithm dispenses with finding connected components.

- In Sibeyn and Meyer's algorithm the reduced graph can directly be taken as input for the semi-external version of Kruskal's algorithm since the first $n'$ nodes are preserved. In Boruvka's algorithm it is more difficult to guarantee that the node identifiers are a sequence without gaps in order to be able to index the union-find data structure.

- In Boruvka's algorithm with full adjacency lists $2 \cdot m$ edges have to be stored in total, in Sibeyn and Meyer's algorithm only $m$ edges are stored at any time.

### 3.3.6 External Realization with Buckets

In order to implement Sibeyn and Meyer's algorithm, we need one adjacency list for each node. We cannot keep all edges in internal memory at the same time, so we have to consider a reasonable disposition of the data in external memory to obtain an efficient implementation. We read only from the list of the last node, but the relabeled edges are written to arbitrary nodes (but the last). Of course we can easily read the edges of the last node blockwise, but it is difficult to write edges blockwise because we cannot afford a write buffer[1] for each node. To solve this problem, we distribute the edges to several buckets, so that we can afford a write buffer for each bucket: we have $b$ buckets and upper bounds $u_0 < u_1 < u_2 < \ldots < u_b, u_0 = 0, u_b \geq n$,

[1] In order to save I/O operations, we must not write the apartly incoming edges immediately. Rather we gather the edges in write buffers until the disk access is worthwhile.

so that bucket $i \in \{1, \ldots, b\}$ contains the edges of the nodes with the identifiers from $u_{i-1} + 1$ to $u_i$ in an arbitrary sequence. (According to the considerations of Section 3.3.1, "the edges of one node" means only the edges that lead to nodes with lower identifiers.)

The buckets can be used directly to write relabeled edges because we can add an edge to the appropriate bucket without worrying about assigning it to the exact node. Nevertheless, we have to worry about the exact node when we want to read the edges of the currently last node. Therefore, we read the complete last external bucket $i$ at a single blow and distribute the edges to internal buckets so that one internal bucket contains all edges of one node. Then the edges of the nodes $u_i$ down to $u_{i-1} + 1$ can be processed before the next external bucket $i - 1$ is loaded.

The first external bucket contains the edges of the nodes that fit in internal memory, i.e., $u_1 = n'$. So, when the second external bucket has been processed, the node reduction is completed and the first bucket contains the reduced graph and can be used as input for Kruskal's algorithm.

Figure 3.1 represents the two layers of data and the processing of the edges during the node reduction phase.

### 3.3.7 External Realization with a Priority Queue

Alternatively, one external priority queue [San00] can be used instead of several external and internal buckets; in this case only one external bucket is needed in order to store the edges of the first $n'$ nodes, which will be processed by Kruskal's algorithm. The shortest edge incident to the last node is on top of the queue, followed at first by the other edges of the last node and then by the shortest edge incident to the second last node and so on. The elements in the queue are quintuples $(u, v, c, e_1, e_2)$, where $u$ is the source, $v$ the target, $c$ the weight and $e_1, e_2$ the original endpoints. Hence, the algorithm can be restated in the following way:

**let** $\pi$ **be** a random permutation over $\{1, \ldots, n\}$
**foreach** $(u, v, c) \in E$ **do** push($(\pi(u), \pi(v), c, u, v)$);

$s := -1$;
**while not** pqueue.empty() **do**
  $(u, v, c, e_1, e_2) :=$ pqueue.pop();
  **if** $u \neq s$ **then**
    $(s, t) := (u, v)$;
    add $(e_1, e_2, c)$ to $T$;
  **else**
    push($(t, v, c, e_1, e_2)$);

**procedure** push($(u, v, c, e_1, e_2)$)
    **if** $u \neq v$ **then**
      **if** $\max(u, v) \leq n'$ **then** bucket.push($(u, v, c, e_1, e_2)$)
      **else** pqueue.push($(\max(u, v), \min(u, v), c, e_1, e_2)$);

The main advantage of using an external priority queue is the scalability: we expect good results for any kind of graphs, even for degenerated ones, for instance, graphs with a small average vertex degree containing some nodes with a very high degree. However, the realization with several buckets (3.3.6) will be faster in most cases, but can get into trouble if it has to deal with such degenerated graphs.

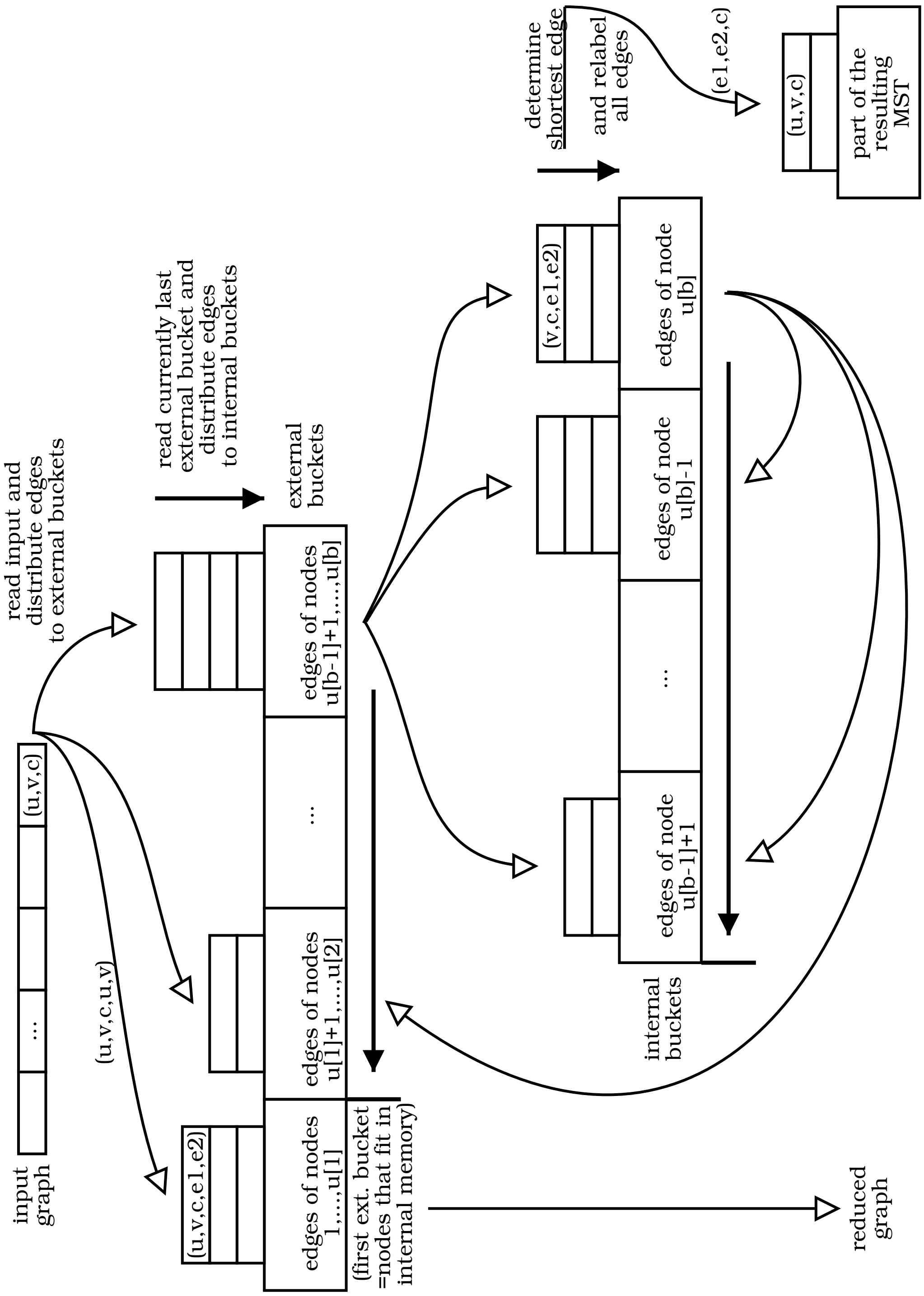



Figure 3.1: Sibeyn and Meyer’s algorithm — external realization

# Chapter 4

# Implementation

## 4.1 <stxxl> Library

The implementation uses the `<stxxl>` library, which is developed at the Max-Planck-Institute for Computer Science. “The core of `<stxxl>` is an implementation of the C++ standard template library STL for external memory (out-of-core) computations, i.e., `<stxxl>` implements containers and algorithms that can process huge volumes of data that only fit on disks. While the compatibility to the STL supports ease of use and compatibility with existing applications, another design priority is high performance [...]: transparent support of multiple disks, variable block lengths, overlapping of I/O and computation, prevention of OS file buffering overhead.” [Dem03]

## 4.2 Data Structures

The basic data structure is the class `Edge` that represents an edge consisting of two endpoints (called `source` and `target`) and the `weight`. Furthermore, we need a class `RelabeledEdge` that is a subclass of `Edge` and additionally contains the `original source` and the `original target`. Sometimes it is not necessary to store the source vertex because if we look at the adjacency list of one particular vertex, the source vertex is known implicitly. In this case we use a class `RelabeledEdgeWithoutSource` in order to reduce memory usage. A `RelabeledEdgeWithoutSource` consists of `target`, `weight`, `original source` and `original target`. (In order to avoid multiple inheritance `RelabeledEdgeWithoutSource` is *not* a superclass of `RelabeledEdge`.) A superclass `EdgeWithoutSource` encapsulates the common components of `Edge` and `RelabeledEdgeWithoutSource`.

The class `EdgeVector` extends the `stxxl::vector`-class and can be used to save a sequence of edges. As `EdgeVector` is a template class, it can be used for both edges and relabeled edges. The main feature of `stxxl::vector` is the storage of the data in external memory while some blocks of data stay in internal memory so that reading and writing is always done blockwise. As the subclass `EdgeVector` should be able to represent a graph, it additionally stores the number of nodes of the graph. Furthermore, there is a method `sortByWeight()` that uses `stxxl::ksort` [DS03] in order to sort the edges by weight.

Finally, the class `MST` represents a (part of a) minimum spanning tree and mainly consists of an `EdgeVector<Edge>`. There are several methods to `add` an `Edge`, a `RelabeledEdge` or a `RelabeledEdgeWithoutSource` to the MST. Polymorphism is avoided due to efficiency reasons.

Figure 4.1 summarizes these data structures.

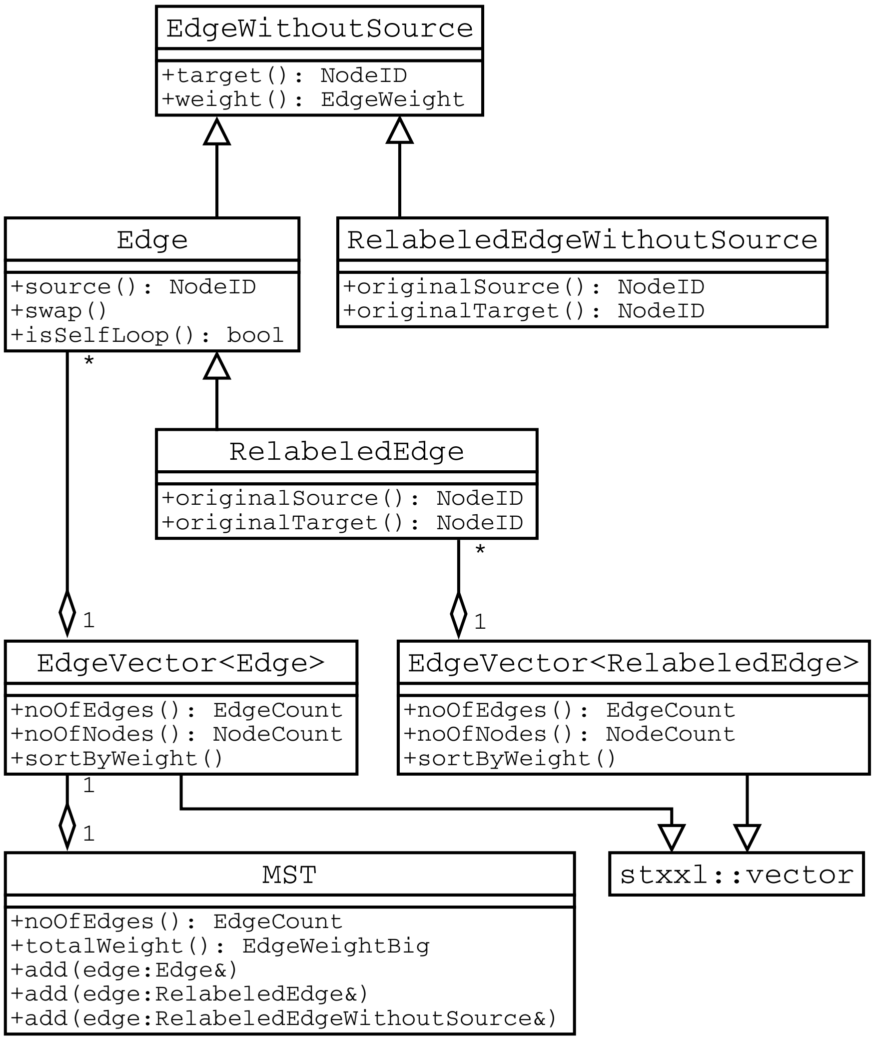



Figure 4.1: UML class diagram [BRJ99] — data structures

## 4.3 Base Case

We can apply the base case when the union-find data structure of all (remaining) nodes fit in internal memory. For this case the class `Kruskal` provides data structures and methods to apply a semi-external version of Kruskal's algorithm. The constructor is given a reference to a graph represented by an `EdgeVector` and a reference to the `MST`-object that stores the resulting MST. Since `Kruskal` is a template class, it can deal with both an `EdgeVector<Edge>` and an `EdgeVector<RelabeledEdge>`.

First the edges are sorted (using the `sortByWeight()`-method of `EdgeVector`) and then the edges are scanned and appropriate union-find operations are performed.

Figure 4.2 is an overview of the interface of the `Kruskal`-class.

## 4.4 Node Reduction with Buckets

### 4.4.1 External Buckets

We have to be able to add edges to an external bucket and read all edges of the currently last bucket, so the functionality of a stack is sufficient. Therefore, we use a `stxxl::stack` for each external bucket. The

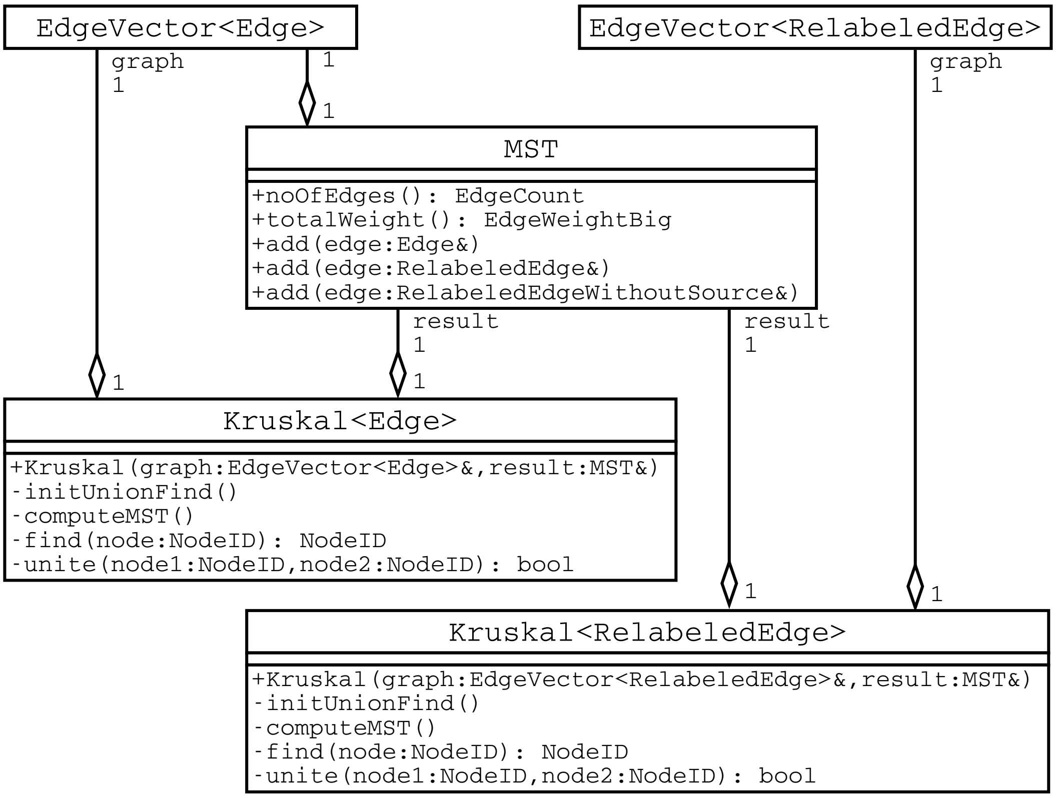



Figure 4.2: UML class diagram — Kruskal's algorithm

first bucket is an exception as we want to use it as input for Kruskal's algorithm that needs a more flexible access since it has to sort the edges. Hence, we use an `EdgeVector` as the first bucket.

The size of the first bucket is the number of nodes that fit in internal memory. The size of the other buckets should be not too small (otherwise too many buckets are needed and the buffers of the buckets exceed the memory limit) and not too large (otherwise the edges of one external bucket do not fit in internal memory). It is convenient to choose the same size for all (but the first) buckets as the computation of the appropriate bucket identifier for a given node is simplified. At first sight this seems not to be the best choice since the buckets with lower IDs probably contain much more edges than the buckets with higher IDs if all buckets have the same size (cp. 3.3.4). But, on the other hand, when the buckets with the higher IDs have been processed, their buffers are not needed any more, and so the released memory can be used to store more edges in internal memory.

### 4.4.2 Internal Buckets

The internal buckets have to be very flexible as for each node the number of edges can be very different and is not known in advance. Furthermore, the internal buckets are reused several times. For instance, the last internal bucket contains the edges of $u_b$, then the edges of $u_{b-1}$ and finally the edges of $u_1$. When it has adapted its size to $u_b$, it is possible that this size is entirely improper for $u_{b-1}$.

The usage of one std::vector for each internal bucket would lead to a waste of either memory or time: if the vectors are not reinitialized after the processing of each external bucket, the total capacity increases continuously so that it exceeds the total number of edges by a high factor. On the other hand, the reinitialization takes time.

To avoid these problems, we use a `CommonPoolOfBlocks`, which is shared by all internal buckets. The `CommonPoolOfBlocks` manages a linked list of free blocks. Each block has a small constant capacity to store edges. By invoking the `request`-method a internal bucket can get a pointer to a free block, which is removed from the free list and can be used exclusively by the requesting internal bucket to store its edges. An internal bucket can give a block back to the pool by calling the `release`-method. Due to these measures the unused capacity is at any time less than the number of internal buckets *times* the capacity of one block because for each internal bucket less than one whole block is unused.

Basically, we need to add edges to internal buckets (when the edges of an external bucket are distributed to the internal buckets) and remove them later (in order to relabel them). The functionality of a stack that uses the `CommonPoolOfBlocks` is encapsulated by the class `SparingStack`. In our case we additionally need a method `determineMinEdge` in order to iterate through all edges to find the shortest one. This method is provided by the subclass `REWS_SparingStack` that is specialized in storing `RelabeledEdgeWithoutSource`-objects. Thus, each internal bucket is represented by one `REWS_SparingStack`.

A `SparingStack` consists of at least one block that does not belong to the `CommonPoolOfBlocks` and therefore is never released. This saves time because the first block does not have to be requested, and usually an internal bucket is not empty so that at least one block is needed.

Figure 4.3 outlines the data structures that are used to implement the internal buckets.

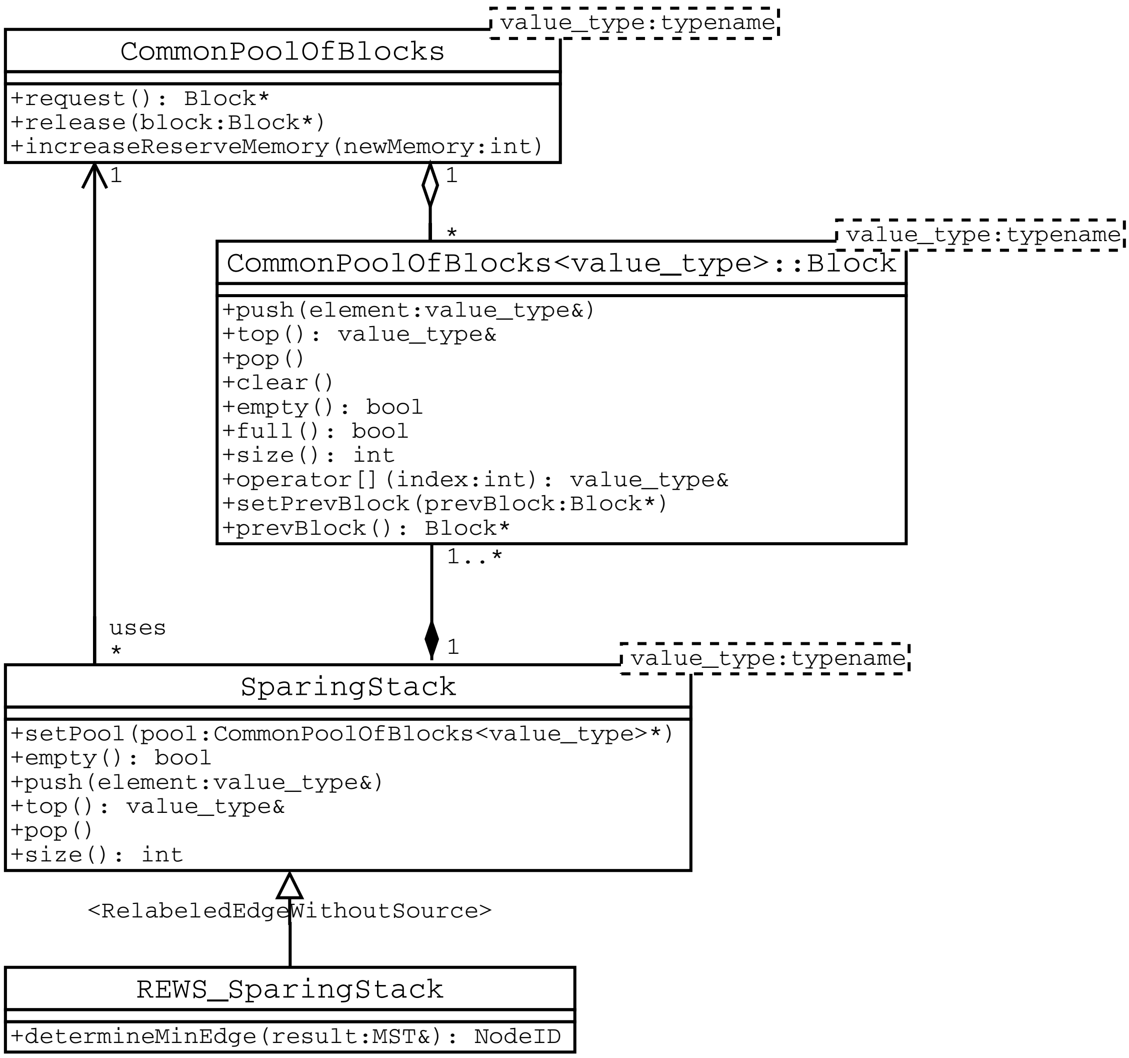



Figure 4.3: UML class diagram — internal buckets

### 4.4.3 Interface

The class `Buckets` provides an interface for the node reduction. The constructor is given (among others) a reference to a graph represented by an `EdgeVector` and a reference to the `MST`-object that stores the resulting MST. The class `Buckets` aggregates both the external and internal buckets, and it performs the node reduction. After the node reduction has been completed, the method `getIntMemBucket` returns a pointer to the first external bucket that contains the reduced graph (= the nodes that fit in internal memory). Figure 4.4 represents the class `Buckets`.

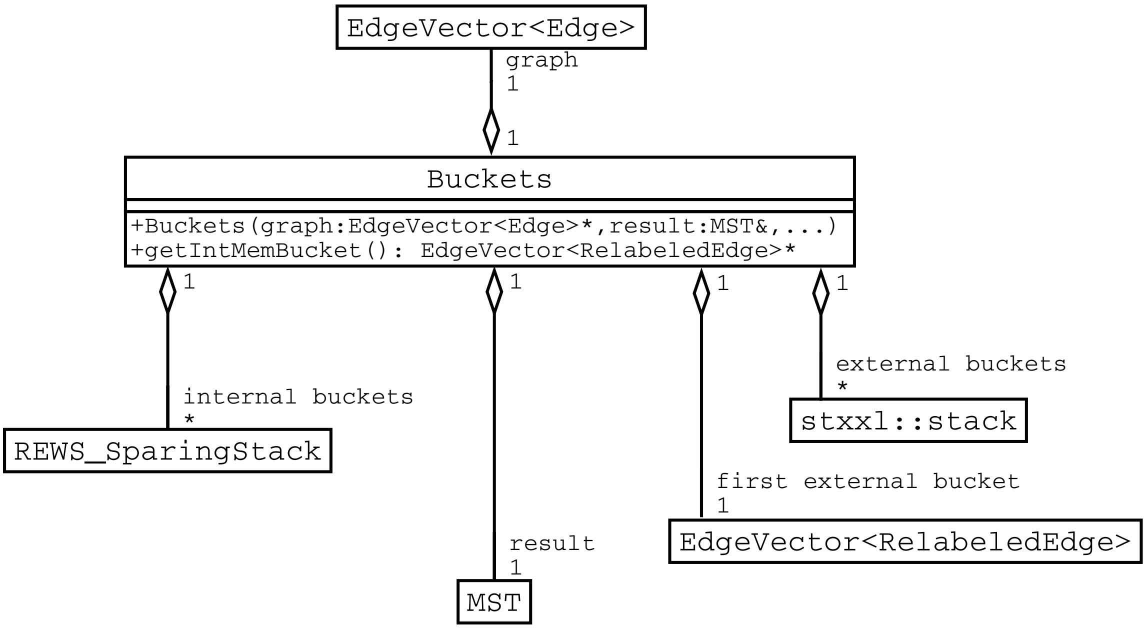



Figure 4.4: UML class diagram — node reduction with buckets

## 4.5 Node Reduction with a Priority Queue

We use an `EdgeVector` as the first external bucket (cp. 4.4.1) and a `stxxl::priority_queue`. An implementation of the node reduction algorithm presented in Section 3.3.7 is straightforward with the help of these data structures.

The interface of the `PQueue` class, which performs the node reduction and aggregates for this purpose both the first external bucket and the priority queue, is virtually identical with the interface of the `Buckets` class (4.4.3). Therefore, the following sections and figures apply to both the `Buckets` and the `PQueue` implementation, although they refer only to the first one (to simplify matters).

## 4.6 Main Program

The sequence of the main program is quite simple:

1. import or generate the graph
2. perform the node reduction
3. use the reduced graph as input for Kruskal's algorithm

Figure 4.5 represents this sequence by a diagram. The second step is skipped if the input graph is so small that it can be processed by Kruskal's algorithm immediately.

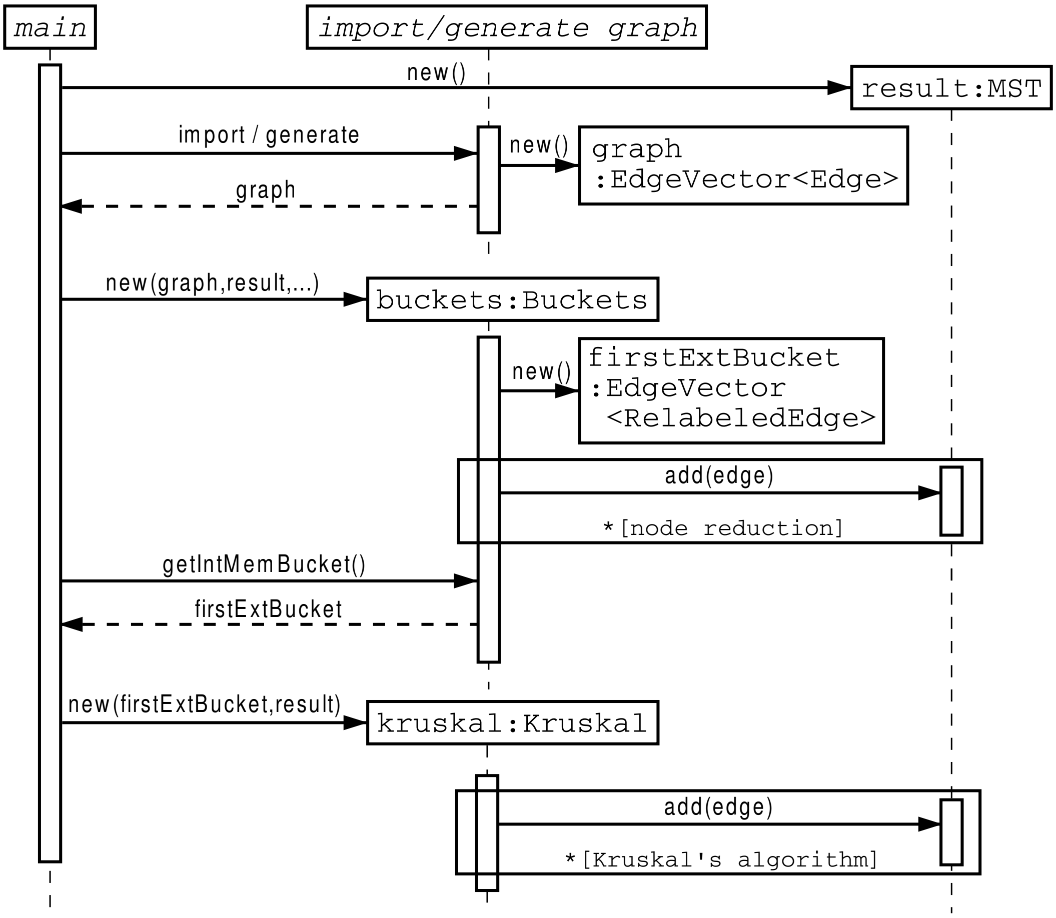



Figure 4.5: UML sequence diagram

## 4.7 Randomization

The randomization of the node indices is essential for the expected running time (cp. 3.3.4). Hence, we apply a (pseudo-)random permutation on the node indices before the nodes are distributed to the external buckets. As not all node indices fit in internal memory at the same time, we cannot use a standard procedure that swaps random elements. Instead, we apply a bijection on each node index, so each edge can be randomized independently (without looking at other edges, just by a relatively simple computation). Of course we need a special bijection that leads to a (pseudo-)random permutation.

We use a variant of a *Feistel permutation* [NR99]. Let $x$ be the node index that should be randomized. We split $x$ into two parts $a := x$ div $r$ and $b := x \bmod r$, where $r := \lceil\sqrt{n}\rceil$. During one iteration $i$ we perform the following operation: $a' := b, b' := (a + f_i(b)) \bmod r$, where $f_i(b) \in \{0, \ldots, r-1\}$ is a random number taken from a table that has been computed once. This step is executed twice (we get $a''$ and $b''$). Finally, $a''$ and $b''$ are recombined to obtain the randomized node index $x' = a'' \cdot r + b''$.

Originally, this is a bijection over $\{0, \ldots, r^2 - 1\}$, but we want a bijection over $\{0, \ldots, n-1\}$[1], so we repeat the application of the bijection, if necessary, until $x' \in \{0, \ldots, n-1\}$.

[1] In Section 3.3 we use node indices from 1 to $n$ for the abstract descriptions. However, node indices from 0 to $n-1$ are more convenient for the implementation.

## 4.8 Removal of Parallel Edges

By relabeling it is possible that parallel edges are created, i.e., edges that lead from the same source vertex to the same target vertex. As a minimum spanning tree contains at the most the shortest one of several parallel edges, the redundant duplicates can and should be ignored for further processing. Hence, it is reasonable to remove these duplicates. As the removal of parallel edges is not very expensive, the disadvantage of looking for them in graphs that do not have many of them is small; but, on the other hand, the advantage for special graphs, grid graphs for example, is clearly noticeable.

The removal of duplicates is integrated in the relabeling step. Instead of adding the relabeled edges of the last node directly to the appropriate external or internal bucket, they are first added to a hash map by calling the `insert`-method of the `DuplicatesRemover`-class, which aggregates the hash map. If an edge with the same source and the same target vertex is already stored in the hash map, it is replaced with the new edge if the new edge is shorter, otherwise the new edge is discarded. If the capacity of the hash map is exhausted, further edges are written to the appropriate external or internal bucket directly; this limits the waste of time when there are many different edges.

When all edges of the last node have been inserted in the hash map, it has to be cleared and the edges have to be written to the appropriate bucket. In order to be able to clear a hash map fast (especially if it contains only few elements), the inserted elements are additionally stored in an array without gaps. Each entry in the hash map consists of the edge and the index in the array where the edge is additionally stored, so the element in the array can be updated in constant time when the corresponding element in the hash map is replaced by a shorter edge. Due to this data structure the hash map can be cleared by iterating through the array (instead of the whole map) and deleting the elements in the map selectively.

## 4.9 Ideas for Further Improvements

There are several possible improvements that have not been implemented (yet).

- *Pipelining*. Some I/Os could be saved, if the sort and the scan part of Kruskal's algorithm were combined by a pipeline. Instead of writing the first sorted elements back and reading them later, we could process the sorted elements immediately. Furthermore, we could join the node reduction and the sorting: instead of writing all edges in an unsorted sequence to the first external bucket, we could gather a certain amount of edges in order to build sorted runs. Then we only have to merge these sorted runs. Planned enhancements of the `<stxxl>` library will make such improvements possible.

- *Exception handling in the buckets implementation.* Currently, the buckets implementation cannot deal with every imaginable graph. As mentioned in Section 3.3.7, we get into trouble if the graph has a small average vertex degree, but contains some nodes with a very a high degree. In this case reading the external bucket that contains these exceptional nodes could fail because not all edges of this external bucket fit in the internal buckets. In order to handle this exceptional cases, we could switch temporarily to the priority queue implementation when we realize that the current external bucket would not fit in internal memory.
  This improvement has not been implemented since such degenerated graphs are quite rare and, if necessary, the priority queue implementation could be used right from the start.

- *Adaptive bucket sizes.* In our buckets implementation all external buckets (but the first) have the same size. As described in Section 4.4.1, this has some advantages. However, it could be worthwhile to try using adaptive bucket sizes, i.e., the buckets with lower IDs contain the edges of less nodes as the average vertex degree increases from the last to the first bucket.

- *Intermediate buckets.* When an external bucket is read and when edges are relabeled, they are distributed to random internal buckets. This leads to many cache misses. In order to reduce the number of cache misses, it could be reasonable to install some intermediate buckets between the existing external and internal buckets, so the concept of two layers of data (introduced in Section 3.3.6) would be extended to three layers and memory accesses would be no longer distributed over the whole internal memory.

# Chapter 5

# Evaluation

## 5.1 Test Data

As there is a lack of real-world data, we use generated graphs to measure the runtime performance. Three different graph families are examined [MS94][1].

1. random graphs with a given number of vertices and a given number of edges: for each edge a random weight and two random endpoints are selected,

2. grid graphs with $nodesX \cdot nodesY$ vertices: each vertex is connected with its four neighbours (except the marginal nodes, which are connected with three resp. two neighbours), the edges have random weights,

3. geometric graphs: the given number of vertices is placed in a square, each vertex is connected with the given number of nearest neighbours, the distance between two nodes is the square of the Euclidean distance (the extraction of the root is insignificant in respect of the sequence of the algorithm and would slow down the graph generation unnecessarily), parallel edges are removed.

Apart from the grid graphs it is possible that the generated graphs are not connected. Especially (very) sparse random or geometric graphs, which have been generated that way, are almost never connected. We do not take any measures to remedy this unwanted state because, in the main, the sequence of the program is independent of the connectivity of the graph: if a connected graph is given, a minimum spanning *tree* will be determined; if an unconnected graph is given, a minimum spanning *forest* will be found. Actually, the only difference is the chance of an earlier abort if $n - 1$ edges have been added to the resulting MST: if we deal with an unconnected graph, we will never be able to fulfill this abort condition, so we will have to scan through all edges. Hence, the fact that we do not make the generated graphs connected in any case leads at the most to a slight slowdown.

## 5.2 Test Environment and Settings

The evaluation is done on a machine with two 2GHz Intel Xeon processors, 1 GB RAM and four disks (80 GB each) with a total I/O bandwidth of up to 180 MB/s [DS03]. Debian Linux with kernel version 2.4.20 is used as operating system. The chosen filesystem is XFS and the swap file has been disabled.

Unless otherwise specified, we use the buckets implementation with the following parameters:

- 4 hard disks (and appropriate parameters to take advantage of the parallelism),

[1] Moret and Shapiro additionally use graphs that represent the worst case for Prim's resp. Kruskal's algorithm. As our implementation has nothing to do with Prim's algorithm, we could not expect informative results if we evaluated the former. We do not explicitly use the latter, either, because, in contrast to Moret and Shapiro, we also process unconnected graphs so that the worst case for Kruskal's algorithm (namely that all edges have to be sorted and scanned) occurs anyway.

- 2 MB block size for `stxxl::vectors` (particularly for the first external bucket) and 512 KB block size for `stxxl::stacks` (i.e., for all other external buckets),

- the first external bucket contains the edges of the first 160,000,000 nodes (the union find data structure of these nodes fits in internal memory), the other external buckets contain the edges of 1,800,000 nodes,

- consequently, there are 1,800,000 internal buckets (each of them possesses one block that can store up to 8 edges[2]), initially the common pool, which can be used by all internal buckets, consists of 1,500,000 blocks (8 edges each),

- 650 MB of internal memory are used for sorting,

- randomization and removing of parallel edges are switched on.

## 5.3 Test Runs and Results

### 5.3.1 Main Results

Table 5.1 represents the main results, namely the results of test runs with the three graph families, several sizes and densities. If $n \leq$ 160,000,000, we are involved with a semi-external test case, otherwise with an external one. The number of processed edges $p$ and the number of removed parallel edges (duplicates) $d$ refer to the node reduction phase, so these columns are blank in semi-external cases.

Most of these test runs were done with the above mentioned settings, only the external bucket size was decreased for test cases with $m \approx 4 \cdot n$ resp. $m \approx 8 \cdot n$[3] so that all edges of one external bucket fit in internal memory in any case.[4]

The results of the semi-external test runs do not show wide differences. It is not possible to distinguish between the different graph families. The more edges are processed the more time per edge is spent, but this complies with the expected behaviour as the time complexity of Kruskal's algorithm is in $O(m \ln m)$. For example, if you compare the random graph with $10 \cdot 10^6$ nodes and $80 \cdot 10^6$ edges with the random graph with $160 \cdot 10^6$ nodes and $1,280 \cdot 10^6$ edges, the time per edge differs (1.77$\mu$s to 2.05$\mu$s), but $t/(m \ln m) \approx$ 97ns in both cases. The denser the graph the less time per edge is taken. A denser graph with the same number of edges consists of less nodes, so a MST of a denser graph consists of less edges. Hence, there are more find operations with negative results (i.e., both nodes already belong to the same set) so that less union operations are needed and the height of the trees becomes very small due to path compression. Therefore, less time is needed when we deal with denser graphs.

Obviously, the external test runs are slower than the semi-external ones, but fortunately the differences keep within reasonable limits. When we look at grid resp. geometric graphs, the differences even decrease when the graph size increases. Mainly, this effect is due to the removal of parallel edges. The more edges in a grid or geometric graph the greater the rate of removed edges (cp. column $d/m$). Hence, the number of edges that have to be processed by Kruskal's algorithm is kept small. 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 graph that is twice this size. Furthermore, the removal of duplicates is one of two reasons why the number of processed edges is distinctly less than the expected number of processed edges (cp. column $p/E(p)$) when we deal with large instances. The other reason is the fact that the analysis of the time complexity (Section 3.3.4) is rather cautious. For instance, it is not regarded that for each node at least one edge, which is added to the MST, is eliminated. Unfortunately, only the second reason applies to random graphs as the removal of parallel edges is not effective. (There are some multiple edges, but distinctly less than 1%.) Hence, the number of

[2] The more edges in one block the greater the extent of unused capacity (cp. 4.4.2) and the smaller the temporal overhead of operations on linked lists of blocks — and vice versa. Hence, 8 edges is a compromise.

[3] Originally, we wanted to evaluate test cases with $m = 2 \cdot n$, $m = 4 \cdot n$ and $m = 8 \cdot n$, but we had to restrict ourselves to approximate values as the average vertex degree of a grid graph is slightly less than four and the average vertex degree of a geometric graph depends on the given number of nearest neighbours and cannot be set to an exact value.

[4] Furthermore, the size of the first external bucket was reduced to 150,000,000 for large geometric graphs due to a slight misfeature of the memory management caused by the expensive geometric graph generator.

| *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 | 1 280 | 2 390 | 7 278 | 3.05 | 4 575 | 45 % | 15 % |
| 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 | 640 | 2 260 | 7 004 | 3.10 | 3 975 | 61 % | 34 % |
| 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 | | | |
| geometric | 320 | 2 160 | 7 549 | 3.49 | 2 650 | 81 % | 30 % |

$n$ nodes, $m$ edges, $t$ elapsed time, $p$ processed edges, $E(p)$ expected value of $p$ according to 3.3.4, $d$ duplicates (parallel edges) removed

Table 5.1: (Semi-)External test cases

processed edges is less than the expected number, but greater than the corresponding number at test runs with grid resp. geometric graphs. Therefore, the time per edge increases when we deal with larger random graphs. This is the "normal" behaviour as the time complexity of the node reduction algorithm is not in $O(m)$. If you regard the time per processed edge, you can find out that this quantity even decreases when the graph size increases.

At first sight it is surprising that test runs with denser graphs are partly as slow as test runs with sparser graphs. For instance, the time per edge for a random graph with $2,560 \cdot 10^6$ edges and $m = 2 \cdot n$ and for a random graph with the same number of edges, but $m = 4 \cdot n$, is almost identical (about 5.5$\mu$s each). As the number of processed edges depends not only on $m$, but also on $n$, we would have expected that the test run with $m = 4 \cdot n$ is faster. However, the sparser graphs benefit from another fact: due to the larger number of nodes the node reduction phase actually takes a longer time (10,571s instead of 9,177s, for the above mentioned example), but, on the other hand, more edges are eliminated because for each node at least the shortest edge is removed, so there are less edges that have to be processed by Kruskal's algorithm (about $1.5 \cdot 10^9$ instead of $2.1 \cdot 10^9$). Hence, Kruskal's algorithm is faster (3,631s instead of 5,071s) and compensates for the slower node reduction phase.

### 5.3.2 Comparison with Internal Implementations

In order to be able to judge the performance of our implementation, we need comparison values. Therefore, we fall back on internal implementations of Kruskal's and of Prim's algorithm developed at the Max-Planck-Institute for Computer Science by Irit Katriel. We used random graphs generated by Irit's program and grid and geometric graphs generated by our program. As the implementation of Prim's algorithm requires more memory, some instances were processed only by Kruskal'a algorithm. Table 5.2 contains the results of the internal test runs.

| | | | Kruskal | | Prim | |
|---|---|---|---|---|---|---|
| *type* | $n/10^6$ | $m/10^6$ | $t[s]$ | $t/m[\mu s]$ | $t[s]$ | $t/m[\mu s]$ |
| grid | 2.5 | 5.0 | 7.5 | 1.50 | 3.8 | 0.75 |
| grid | 5.0 | 10.0 | 15.2 | 1.52 | 8.2 | 0.82 |
| random | 2.5 | 5.0 | 6.5 | 1.30 | 10.0 | 1.99 |
| random | 5.0 | 10.0 | 13.5 | 1.35 | 22.1 | 2.21 |
| random | 10.0 | 20.0 | 28.2 | 1.41 | | |
| random | 1.3 | 5.0 | 5.3 | 1.07 | 6.1 | 1.22 |
| random | 2.5 | 10.0 | 10.9 | 1.09 | 12.9 | 1.29 |
| random | 5.0 | 20.0 | 22.4 | 1.12 | | |
| random | 0.6 | 5.0 | 4.7 | 0.94 | 3.7 | 0.73 |
| random | 1.3 | 10.0 | 9.6 | 0.96 | 7.5 | 0.75 |
| random | 2.5 | 20.0 | 19.9 | 1.00 | | |
| geometric | 2.5 | 4.7 | 7.3 | 1.56 | 5.6 | 1.19 |
| geometric | 5.0 | 9.3 | 14.5 | 1.56 | 13.1 | 1.41 |
| geometric | 1.3 | 4.4 | 6.3 | 1.42 | 2.9 | 0.66 |
| geometric | 2.5 | 8.8 | 12.6 | 1.43 | 6.4 | 0.73 |
| geometric | 0.6 | 4.2 | 5.3 | 1.26 | 1.7 | 0.41 |
| geometric | 1.3 | 8.4 | 10.8 | 1.27 | 3.6 | 0.43 |

Table 5.2: Internal test cases

In the main, both implementations show the expected behaviour. Kruskal's algorithm is quite independent of the graph type. When denser graphs are processed, Kruskal's algorithm gets faster due to the same reasons that applied to the semi-external test cases described in Section 5.3.1. One basic feature of Prim's algorithm is the fact that the time per edge decreases when the density increases. This feature is confirmed by our results. Furthermore, Prim's algorithm is more efficient when grid or geometric graphs are processed.

In Table 5.3, we compare external test runs with internal ones. For each graph type and for each density, the last entry in Table 5.1 is compared with the last entry in Table 5.2. The time per edge of the external test case is divided by the time per edge of the corresponding internal test case for both algorithms, Kruskal's and Prim's.

| | | $(t/m)_{\text{ext}} : (t/m)_{\text{int}}$ | |
|---|---|---|---|
| *type* | *density* | Kruskal | Prim |
| grid | $m \approx 2 \cdot n$ | 2.3 | 4.3 |
| random | $m \approx 2 \cdot n$ | 3.9 | 2.5 |
| random | $m \approx 4 \cdot n$ | 5.0 | 4.3 |
| random | $m \approx 8 \cdot n$ | 4.8 | 6.4 |
| geometric | $m \approx 2 \cdot n$ | 2.0 | 2.2 |
| geometric | $m \approx 4 \cdot n$ | 2.2 | 4.2 |
| geometric | $m \approx 8 \cdot n$ | 2.7 | 8.1 |

Table 5.3: Comparison between external and internal test cases

With regard to the internal implementation of Kruskal's algorithm, our external implementation is between two and five times slower. When we compare our implementation with Prim's algorithm, the factor ranges between 2.2 and 8.1. When we make the analogous comparison between the semi-external version of Kruskal's algorithm and the internal one, we obtain a factor between 1.5 and 2.[5] However, we have to consider that these comparisons are *dis*advantageous to our implementation as we cannot expect that the expense grows only linearly.

Figure 5.1 illustrates the results of internal, semi-external and external test runs with $m \approx 2 \cdot n$. Analogically, the Figures 5.2 and 5.3 show the results of the test runs with $m \approx 4 \cdot n$ resp. $m \approx 8 \cdot n$. To keep the figures easy to survey, we omit the internal test runs with grid and geometric graphs.

### 5.3.3 Randomization and Removal of Parallel Edges

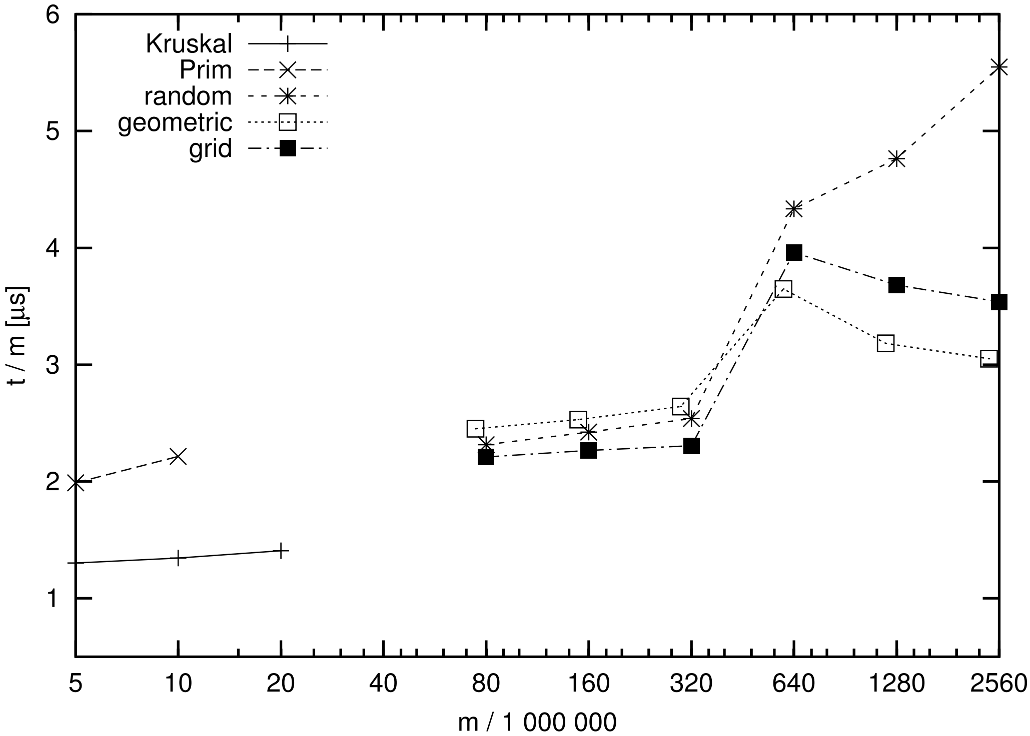
As we wanted to evaluate the benefit of the removal of parallel edges, we reran the external test cases of grid graphs with *de*activated `DuplicatesRemover` (cp. 4.8). Table 5.4 shows both the results with activated and with deactivated `DuplicatesRemover`.

| *parallel edges* | $n/10^6$ | $m/10^6$ | $t[s]$ | $t/m[\mu s]$ | $p/10^6$ | $p/E(p)$ | $d/m$ |
|---|---|---|---|---|---|---|---|
| removed | 320 | 640 | 2 535 | 3.96 | 750 | 85 % | 4 % |
| removed | 640 | 1 280 | 4 712 | 3.68 | 2 492 | 70 % | 13 % |
| removed | 1 280 | 2 560 | 9 056 | 3.54 | 6 167 | 58 % | 22 % |
| *not* removed | 320 | 640 | 2 539 | 3.97 | 760 | 86 % | |
| *not* removed | 640 | 1 280 | 5 006 | 3.91 | 2 642 | 74 % | |
| *not* removed | 1 280 | 2 560 | 10 171 | 3.97 | 6 969 | 65 % | |

Table 5.4: Grid graphs — removal of parallel edges

From these results, we can conclude that the removal of parallel edges becomes worthwhile when the graph size increases. For instance, the `DuplicatesRemover` eliminates 22% ($\approx 5.7 \cdot 10^8$) of all edges from a grid graph with $1.28 \cdot 10^9$ nodes and about $2.56 \cdot 10^9$ edges. When these edges are not removed, most of them are processed more than once, so the number of processed edges even increases from $6.2 \cdot 10^9$ to $7.0 \cdot 10^9$. Hence, the test run with activated `DuplicatesRemover` is more than 10% faster. Furthermore, less internal memory is allocated, so the external bucket size could be increased.

[5] Both the internal and the semi-external algorithm have a number of opportunities for further tuning. Currently, the external sorter benefits from the fact that only integer keys are used, while the internal sorter is comparison based. Hence, bucket sort could accelerate the internal sorter. On the other hand, the external sorter is not optimized for small elements. Furthermore, pipelining (cp. 4.9) has not been implemented, yet. But none of these measures is likely to yield more than a factor of 2.




*Kruskal* and *Prim* denote the internal test runs with random graphs, *random*, *geometric* and *grid* label the (semi-)external test runs with the corresponding graph type.

Figure 5.1: $m \approx 2 \cdot n$

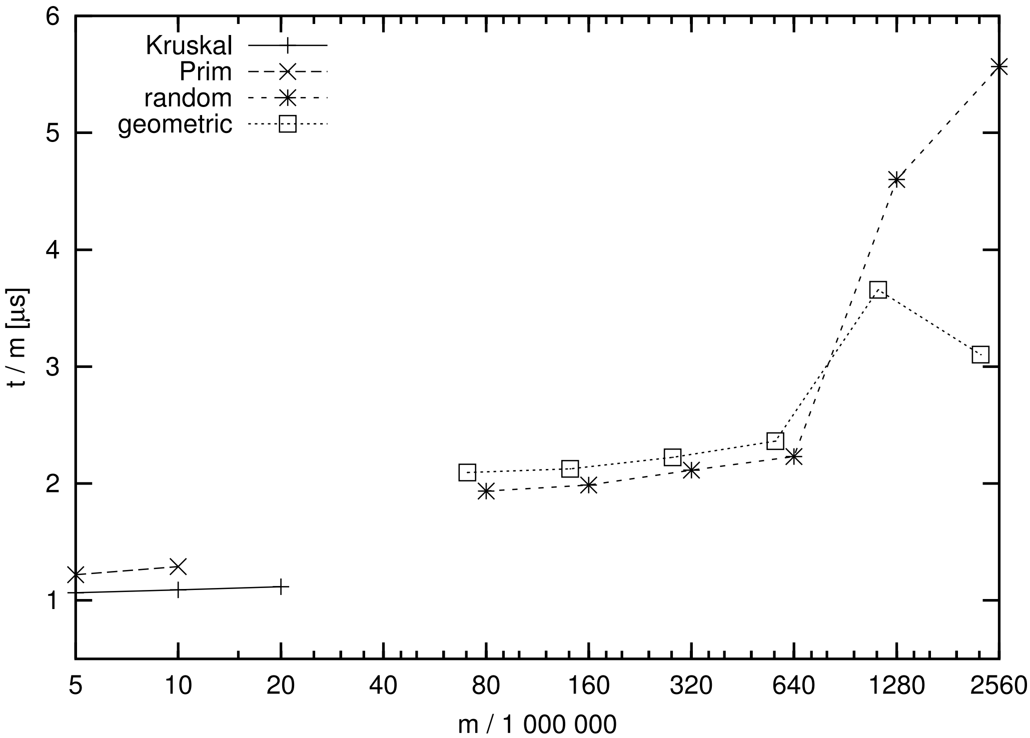



Figure 5.2: $m \approx 4 \cdot n$

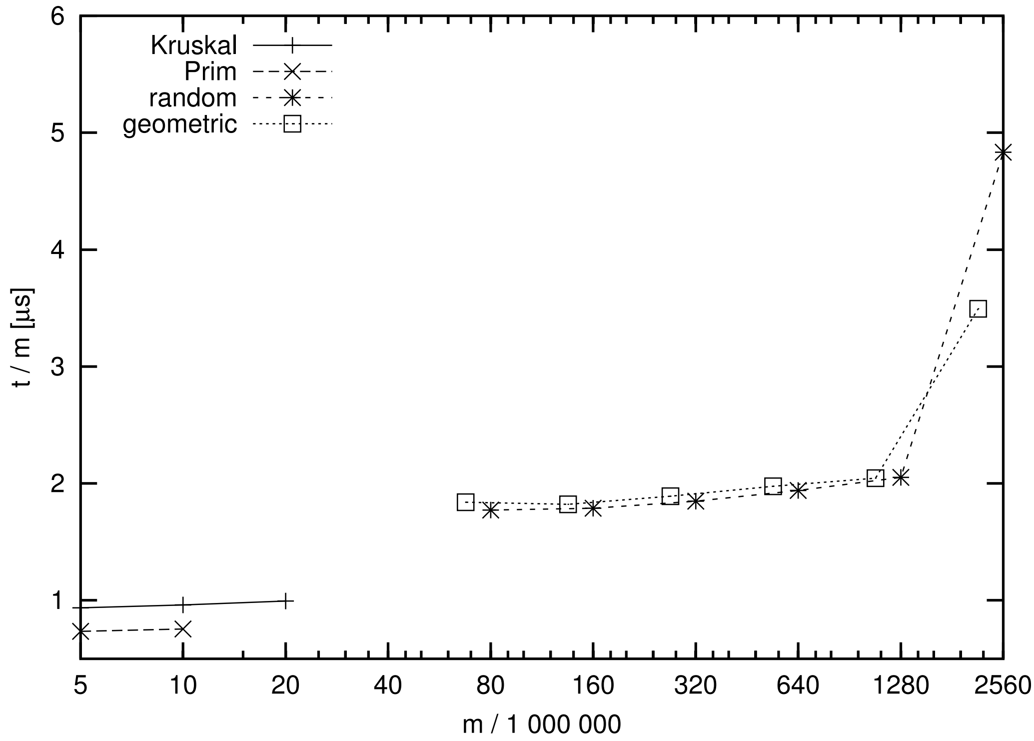



Figure 5.3: $m \approx 8 \cdot n$

The randomization of a random graph is redundant as the node indices are uniformly distributed anyway. Furthermore, the removal of duplicates is not worthwhile since a random graph contains only few parallel edges. Hence, we wanted to find out the extent of the overhead. Table 5.5 represents the results of two test runs with a random graph, one with randomization and removal of parallel edges and one without these measures.

| *parallel edges* | *randomization* | $n/10^6$ | $m/10^6$ | $t[s]$ | $t/m[\mu s]$ | $p/10^6$ | $p/E(p)$ | $d$ |
|---|---|---|---|---|---|---|---|---|
| removed | activated | 1 280 | 2 560 | 14 202 | 5.55 | 7 676 | 72 % | 555 |
| *not* removed | *de*activated | 1 280 | 2 560 | 12 465 | 4.87 | 7 675 | 72 % | |

Table 5.5: Random graph — randomization and removal of parallel edges

Firstly, these results confirm our conjecture that the randomization of a random graph is superfluous. The number of processed edges is almost identical. Secondly, it is obvious that the removal of duplicates is not worthwhile because only 555 ($\approx 0.00002\%$) parallel edges are eliminated. Finally, the test run without randomization and without `DuplicatesRemover` is about 12% faster. As both measures do not speed up the processing, this difference exactly reflects the expense of the randomization and the removal of parallel edges.

### 5.3.4 Buckets vs. Priority Queue

In order to compare both implementations, we selected three representative instances and applied both versions one after the other. Table 5.6 shows the different execution times.

The results demonstrate that currently the buckets implementation needs less than half the time of the priority queue implementation. There are two reasons for this. Firstly, the buckets implementation is optimized for the MST problem, while the priority queue of the `<stxxl>` library is a very general data structure. Secondly, the priority queue is not fully developed yet.

| *implementation* | *type* | $n/10^6$ | $m/10^6$ | $t[s]$ | $t/m[\mu s]$ |
|---|---|---|---|---|---|
| buckets | grid | 320 | 640 | 2 535 | 3.96 |
| priority queue | grid | 320 | 640 | 6 156 | 9.62 |
| buckets | random | 320 | 640 | 2 773 | 4.33 |
| priority queue | random | 320 | 640 | 6 013 | 9.40 |
| buckets | random | 1 280 | 2 560 | 14 202 | 5.55 |
| priority queue | random | 1 280 | 2 560 | 54 497 | 21.29 |

Table 5.6: Buckets vs. priority queue

### 5.3.5 Large Instances

To sound the limits of the program, we processed grid graphs with $2^{31}$ and with $2^{32}$ nodes. As the number of external buckets increased, the block size for `stxxl::stacks` had to be reduced.[6] Table 5.7 shows the external test cases of grid graphs including the above mentioned large instances.

| *block size* | $n/10^6$ | $m/10^6$ | $t[s]$ | $t/m[\mu s]$ | $p/10^6$ | $p/E(p)$ | $d/m$ |
|---|---|---|---|---|---|---|---|
| 512 KB | 320 | 640 | 2 535 | 3.96 | 750 | 85 % | 4 % |
| 512 KB | 640 | 1 280 | 4 712 | 3.68 | 2 492 | 70 % | 13 % |
| 512 KB | 1 280 | 2 560 | 9 056 | 3.54 | 6 167 | 58 % | 22 % |
| 256 KB | 2 150 | 4 290 | 15 803 | 3.68 | 11 230 | 50 % | 27 % |
| 128 KB | 4 290 | 8 590 | 31 081 | 3.62 | 26 260 | 46 % | 29 % |

Table 5.7: Grid graphs — large instances

Although the block size is halved twice, the time per edge is almost constant and does not increase when very large graphs are processed. The reduced block size is compensated by the `DuplicatesRemover`, which is very efficient for large instances. For example, the number of processed edges is less than half the expected number if we regard the grid graph with $2^{32}$ nodes. This is achieved by removing 29% of all edges.

[6]Furthermore, the sizes of the external buckets were adapted for the last test run so that they were particularly appropriate for a large grid graph: the first external bucket contained the edges of 150,000,000 nodes (instead of 160,000,000) and all other external buckets contained the edges of 2,500,000 nodes (instead of 1,800,000).

# Acknowledgements

First of all, I like to thank my supervisor, Peter Sanders, for the numerous fertile discussions. His suggestions and his optimism were very helpful.

Roman Dementiev always provided an up-to-date version of the `<stxxl>`-library and made sure that the test environment worked. Furthermore, he enhanced the library according to my requirements and supported my hunt for bugs (particularly memory leaks) with great patience.

Job Sibeyn kindly made his not yet published external memory MST algorithm available. Irit Katriel provided internal implementations of Prim's and of Kruskal's algorithm so that I was able to do comparative measurements.

# Bibliography


[ABT00] 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.

[ABW02] J. Abello, A. L. Buchsbaum, and J. R. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32(3):437–458, 2002. 4

[AV88] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988. 1

[BRJ99] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley, 1999. 9

[CGG+95] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. ACM-SIAM Symp. on Discrete Algorithms*, pages 139–149, 1995. 4

[Dem03] R. Dementiev. <stxxl> home page. http://www.mpi-sb.mpg.de/~rdementi/stxxl.html, July 2003. 8

[DS03] R. Dementiev and P. Sanders. Asynchronous parallel disk sorting. In *15th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 138–148, 2003. 8, 15

[JaJ92] J. JaJa. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992. 3

[Knu98] Donald E. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 3rd edition, 1998.

[Liu01] Yan Liu. Minimum spanning trees. http://www.csee.wvu.edu/~ksmani/courses/fa01/random/lecnotes/lecture11.pdf, Fall 2001. Lecture notes, “Randomized Algorithms” course, LDCSEE, West Virginia University. 3

[MS94] 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. 15

[MSS03] U. Meyer, P. Sanders, and J. Sibeyn, editors. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS Tutorial*. Springer, 2003. 1, 4

[NR99] Moni Naor and Omer 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. 13

[OW96] T. Ottmann and P. Widmayer. *Algorithmen und Datenstrukturen*. Spektrum Akademischer Verlag, 3rd edition, 1996. 2

[San00] P. Sanders. Fast priority queues for cached memory. *ACM Journal of Experimental Algorithmics*, 5(7), 2000. 6

[Sed92] R. Sedgewick. *Algorithmen*. Addison-Wesley, 1992.

[Ski98] Steven S. Skiena. *The Algorithm Design Manual*. Springer, 1998. 2

[SM] J. Sibeyn and U. Meyer. External connected components and beyond. *unpublished*. 4, 5