I N F O R M A T I K

A Practical Minimum Spanning Tree Algorithm Using the Cycle Property

Irit Katriel, Peter Sanders and Jesper Larsson Träff

MPI–I–2002–1–003 October 2002

FORSCHUNGSBERICHT RESEARCH REPORT

M A X - P L A N C K - I N S T I T U T
F Ü R
I N F O R M A T I K

Stuhlsatzenhausweg 85 66123 Saarbrücken Germany

Authors' Addresses

Irit Katriel, Peter Sanders
Stuhlsatzenhausweg 85
Max-Planck-Institut für Informatik,
66123 Saarbrücken, Germany
email: {irit,sanders}@mpi-sb.mpg.de

Jesper Larsson Träff
C&C Research Laboratories, NEC Europe Ltd.,
Sankt Augustin, Germany
email: traff@ccrl-nece.de

Abstract

We present a simple new algorithm for computing minimum spanning trees that is more than two times faster than the best previously known algorithms (for dense, "difficult" inputs). It is of conceptual interest that the algorithm uses the property that the heaviest edge in a cycle can be discarded. Previously this has only been exploited in asymptotically optimal algorithms that are considered to be impractical. An additional advantage is that the algorithm can greatly profit from pipelined memory access. Hence, an implementation on a vector machine is up to 13 times faster than previous algorithms. We outline additional refinements for MSTs of implicitly defined graphs and the use of the central data structure for querying the heaviest edge between two nodes in the MST. The latter result is also interesting for sparse graphs.

This work is partially supported by DFG grant SA 933/1-1.

Keywords

Minimum Spanning Tree, Minimum Spanning Forest, Algorithm Engineering

1 Introduction

Given an undirected connected graph G with n nodes, m edges and nonnegative edge weights, the minimum spanning tree (MST) problem asks for a minimum total weight subset of the edges that forms a spanning tree of G.

The current state of the art in MST algorithms shows a gap between theory and practice. The algorithms used in practice are among the oldest network algorithms [4, 16, 8, 13] and are all based on the *partition property*: a *lightest* edge leaving a set of nodes can be used for an MST. More specifically, Kruskal's algorithm [13] is best for sparse graphs. Its running time is asymptotically dominated by the time for sorting the edges by weight. For dense graphs ($m \gg n$), the Jarník-Prim (JP) algorithm is better [8, 18]. Using Fibonacci heap priority queues, its execution time is $\mathcal{O}(n \log n + m)$. Using pairing heaps [5] Moret and Shapiro [15] get quite favorable results in practice at the price of slightly worse performance guarantees ($\Omega(n \log n + m \log\log n)$).

On the theoretical side there is a randomized linear time algorithm [9] and an almost linear time deterministic algorithm [17]. But these algorithms are usually considered impractical because they are complicated and because the constant factors in the execution time look unfavorable. These algorithms complement the partition property with the *cycle property*: a *heaviest edge* in any cycle is not needed for an MST.

In this paper we partially close this gap. We develop a simple $\mathcal{O}(n \log n + m)$ expected time algorithm using the cycle property that is very fast on dense graphs. Our experiments show that it is more than two times faster than the JP algorithm for large dense graphs that require a large number of priority queue updates for JP. For future architectures it promises even larger speedups because it profits from pipelining for hiding memory access latency. An implementation on a vector machine shows a speedup by a factor of 13 for large dense graphs.

Our algorithm is a simplification of the linear time randomized algorithms. Its asymptotic complexity is $\mathcal{O}(m + n \log n)$. When $m \gg n \log n$ we get a linear time algorithm with small constant factors. The key component of these algorithms works as follows. Generate a smaller graph G' by selecting a random sample of the edges of G. Find a minimum spanning forest T' of G'. Then, *filter* each edge $e \in E$ using the cycle property: Discard e if it is the heaviest edge on a cycle in $T' \cup \{e\}$. Finally, find the MST of the graph that contains the edges T' and the edges that were not filtered out. Since MST edges were not discarded, this is also the MST of G.

Klein and Tarjan [11] prove that if the sample graph G' is obtained by including each edge of G independently with probability p, then the expected number of edges that are not filtered out is bounded from above by n/p. By setting $p = \sqrt{n/m}$ both recursively solved MST instances can be made small. It remains to find an efficient way to implement filtering.

King [10] suggests a filtering scheme which requires an $\mathcal{O}\big(n \log \frac{m+n}{n}\big)$ preprocessing stage, after which the filtering can be done with $\mathcal{O}(1)$ time per edge (for a total of $\mathcal{O}(m)$). The preprocessing stage runs Boruvka's [4, 16] algorithm on the spanning tree T and uses the intermediate results to construct a tree B that has the vertices of G as leaves such that: (1) the heaviest edge on the path between two leaves in B is the same as the heaviest edge between them in T'. (2) B is a *full branching tree*; that is, all the leaves of B are at the same level and each internal node has at least two sons. (3) B has at most $2n$ nodes. It is then possible to apply to B Komlós's algorithm [12] for maximum edge weight queries on a full branching tree. This algorithm builds a data structure of size $\mathcal{O}\big(n \log(\frac{m+n}{n})\big)$ which can be used to find the maximum edge weight on the path between leaves u and v, denoted $F(u, v)$, in constant time. A path between two leaves is divided at their least common ancestor (LCA) into two half paths and the maximum weight on each half path is precomputed. In addition, during the preprocessing stage the algorithm generates information with which the LCA of two leaves can be found in constant time.

In Section 2 we develop a simpler filtering scheme which is based on the order in which the JP algorithm adds nodes to the MSF of the sample graph G'. We show that using this ordering, computing $F(u, v)$ reduces to a single interval maximum query. This is significantly simpler to implement than Komlós's algorithm because (1) we do not need to convert the MSF of the sample into a different tree. (2) interval maximum computation is more structured than path maximum in a full branching tree, where nodes may have different degrees. As a consequence, the preprocessing stage involves computation of simpler functions and needs simpler data structures.

Interval maximum can be found in constant time by applying a standard technique that uses precomputed tables of total size $\mathcal{O}(n \log n)$. The tables store prefix minima and suffix maxima [7]. We explain how to arrange these tables in such a way that $F(u, v)$ can be found using two table lookups for finding the JP-order, one **xor** operation, one operation finding the most significant nonzero bit, two table lookups in fused prefix and suffix tables and some shifts and adds for index calculations. These operations can be executed independently for all edges in contrast to the priority queue accesses of the JP algorithm that have to be executed sequentially to preserve correctness.

In Section 3 and Appendix B.1 we report measurements on current high-end microprocessors that show speedup up to a factor 3.35 compared to a highly tuned implementation of the JP algorithm. An implementation on a vector computer results in even higher speedup of up to 13.

Our algorithm is also interesting for sparse graphs when we are interested in the all-pairs minimax shortest-paths problem [2, 6]. Details are explained in Appendix A.3.

2 The I-Max-Filter Algorithm

In Section 2.1 we explain how finding the heaviest edge between two nodes in an MST can be reduced to finding an interval maximum. The array used is the edge weights of the MST stored in the order in which the edges are added by the JP algorithm. Then in Section 2.2 we explain how this interval maximum can be computed using one further table lookup per node, an **xor** operation and a computation of the position of the most significant one-bit in an integer. In Section 2.3 we use these components to assemble the I-Max-Filter algorithm for computing MSTs. Appendix A presents refinements that reduce the number of cache faults, give improved performance for implicitly defined graphs and explain how our algorithm can be applied to the all-pairs minimax shortest paths problem.

2.1 Reduction to Interval Maxima

The following lemma shows that by renumbering nodes according to the order in which they are added to the MST by the JP algorithm, heaviest edge queries can be reduced to simple interval maximum queries.

Lemma 1 *Consider an MST* $T = (\{0, \ldots, n-1\}, E_T)$ *where the JP algorithm (JP) adds the nodes to the tree in the order* $0, \ldots, n-1$*. Let* e_i*,* $0 < i < n$ *denote the edge used to add node* i *to the tree by the JP algorithm. Let* w_i*, denote the weight of* e_i*. Then, for all nodes* $u < v$*, the heaviest edge on the path from* u *to* v *in* T *has weight* $\max_{u<j\leq v} w_j$*.*

Proof: By induction over v. The claim is trivially true for $v = 1$. For the induction step we assume that the claim is true for all pairs of nodes (u, v') with $u < v' < v$ and show that it is also true for the pair (u, v). First note that e_v is on the path from u to v because in the JP algorithm u is inserted before v and v is an isolated node until e_v is added to the tree. Let $v' < v$ denote the node at the other end of edge e_v. Edge e_v is heavier than all the edges $e_{v'+1}, \ldots e_{v-1}$ because otherwise the JP algorithm would have added v, using e_v, earlier. There are two cases to consider (see Figure 1).

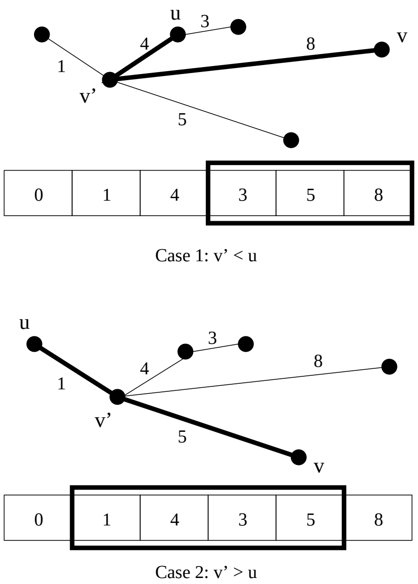

Figure 1: Illustration of the two cases of Lemma 1. The JP algorithm adds the nodes from left to right.

Case $v' \leq u$: By the induction hypothesis, the heaviest edge on the path from v' to u is $\max_{v'<j\leq u} w_j$. Since all these edges are lighter than e_v, the maximum over w_u, ... ,w_v finds the correct answer w_v.
Case $v' > u$: By the induction hypothesis, the heaviest edge on the path between u and v' has weight $\max_{u<j\leq v'} w_j$. Hence, the heaviest edge we are looking for has weight $\max\{w_v, \max_{u<j\leq v'} w_j\}$. Maximizing over the larger set $\max_{u<j\leq v} w_j$ will return the right answer since e_v is heavier than the edges $e_{v'+1}, \ldots e_{v-1}$. ■

Lemma 1 also holds when we have the MSF of an unconnected graph rather than the MST of a connected graph. When JP spans a connected component, it selects an arbitrary node i and adds it to the MSF with $w_i = \infty$. Then the interval maxima for two nodes which are in two different components is ∞, as we would expect.

2.2 Computation of Interval Maxima

Given an array $a[0]\ldots a[n-1]$, we explain how $\max a[i..j]$ can be computed in constant time using preprocessing time and space $\mathcal{O}(n \log n)$. The emphasis is on very simple and fast queries since we are looking at applications where many more than $n \log n$ queries are made. To this end we develop an efficient implementation of a basic method described in [7, Section 3.4.3] which is a special case of the general method in [3]. This algorithm might be of

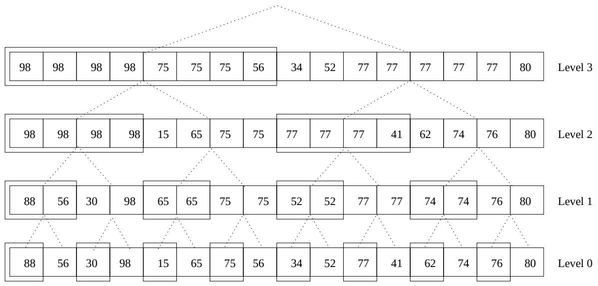

Figure 2: Example of a layers array for interval maxima. The suffix sections are marked by an extra surrounding box.

independent interest for other applications. Slight modifications of this basic algorithm are necessary in order to use it in the I-Max-Filter algorithm. They will be described later. In the following, we assume that n is a power of two. Adaption to the general case is simple by either rounding up to the next power of two and filling the array with $-\infty$ or by introducing a few case distinctions while initializing the data structure.

Consider a complete binary tree built on top of a so that the entries of a are the leaves (see level 0 in Figure 2). The idea is to store an array of prefix or suffix maxima with every internal node of the tree. Left successors store suffix maxima. Right successors store prefix maxima. The size of an array is proportional to the size of the subtree rooted at the corresponding node. To compute the interval maximum $\max a[i..j]$, let v denote the least common ancestor of $a[i]$ and $a[j]$. Let u denote the left successor of v and let w denote the right successor of v. Let $u[i]$ denote the suffix maximum corresponding to leaf i in the suffix maxima array stored in u. Correspondingly, let $w[j]$ denote the prefix maximum corresponding to leaf j in the prefix maxima array stored in w. Then $\max a[i..j] = \max(u[i], w[j])$.

We observed that this approach can be implemented in a very simple way using a $\log(n) \times n$ array preSuf. As can be seen in Figure 2, all suffix and prefix arrays in one layer can be assembled in one array as follows

$$\mathrm{preSuf}[\ell][i] = \begin{cases} \max(a[2^\ell b..i]) \text{ for odd } b \\ \max(a[i..(2^\ell + 1)b - 1]) \text{ else} \end{cases}$$

where $b = \lfloor i/2^\ell \rfloor$.

Furthermore, the interval boundaries can be used to index the arrays. We simply have $\max a[i..j] = \max(\mathrm{preSuf}[\ell][i], \mathrm{preSuf}[\ell][j])$ where $\ell = \mathrm{msbPos}(i \oplus j)$; $\oplus$ is the bit-wise exclusive-or operation and $\mathrm{msbPos}(x) = \lfloor \log_2 x \rfloor$, which is equal to the position of the most significant nonzero bit of x (starting at

```
(* Compute MST of G = ({0,...,n-1}, E) *)
Function I-Max-Filter-MST(E) : set of Edge
    E' := random sample from E of size √(mn)
    E'' := JP-MST(E')
    Let jpNum[0..n-1] denote the order
        in which JP-MST added the nodes
    Initialize the table preSuf[0..log n][0..n-1]
        as described in Section 2.2
    (* Filtering loop *)
    forall edges e = (u,v) ∈ E do
        ℓ := msbPos(jpNum[u]⊕jpNum[v])
        if w_e < preSuf[ℓ][jpNum[u]] and
           w_e < preSuf[ℓ][jpNum[v]] then
           add e to E''
    return JP-MST(E'')
```

Figure 3: The I-Max-Filter algorithm

0). Layer 0 is identical to a. $\mathrm{msbPos}(x)$ can be computed by a table lookup[1]. A further optimization stores a pointer to the array $\mathrm{preSuf}[\ell]$ in this layer table. As the computation is symmetric, we can conduct a table lookup with indices i, j without knowing whether $i < j$ or $j < i$.

To use this data structure for the I-Max-Filter algorithm we need a small modification since we are interested in maxima of the form $\max a[\min(i,j)+1..\max(i,j)]$ without knowing which of two endpoints is the smaller. Here we simply note that the approach still works if we redefine the suffix maxima to exclude the first entry, i.e., $\mathrm{preSuf}[\ell][i] = \max(a[i+1..(2^\ell+1)\lfloor i/2^\ell \rfloor - 1])$ if $\lfloor i/2^\ell \rfloor$ is even.

2.3 Putting the Pieces Together

Figure 3 summarizes the I-Max-Filter algorithm and the following Theorem establishes its complexity.

Theorem 1 *The I-Max-Filter algorithm computes MSTs in expected time $mT_{\mathrm{filter}} + \mathcal{O}(n \log n + \sqrt{nm})$ where T_{filter} is the time required to query the filter about one edge.*

In particular, if $m = \omega(n \log n)$, the execution time is $(1 + o(1))mT_{\mathrm{filter}}$.

[1] Alternatively, one could inspect the exponent in a floating point representation of x.

PROOF: Taking a sample can be implemented to run in constant time per sampled element. Running JP on the sample takes time $\mathcal{O}(n \log n + \sqrt{nm})$ if a Fibonacci heap (or another data structure with similar time bounds) is used for the priority queue. The lookup tables can be computed in time $\mathcal{O}(n \log n)$. The filtering loop takes time mT_{filter}.[2] By the sampling lemma explained in the introduction [11, Lemma 1], the expected number of edges in E'' is $n/\sqrt{n/m} = \sqrt{nm}$. Hence, running JP on E'' takes expected time $\mathcal{O}(n \log n + \sqrt{nm})$. Summing all the component execution times yields the claimed time bound. ∎

3 Experimental Evaluation

The objective of this section is to demonstrate that the I-Max-Filter algorithm is a serious contestant for the fastest MST algorithm for dense graphs ($m \gg n \log n$). We compare our implementation with a fast implementation of the JP algorithm. In [15] the execution time of the JP algorithm using different priority queues is compared and pairing heaps are found to be the fastest on dense graphs. We took the pairing heap from their code and combined it with a faster, array based graph representation.[3] This implementation of JP consistently outperforms [15] and LEDA [14].

3.1 Graph Representations

One issue in comparing MST-algorithms for dense graphs is the underlying graph representation. The JP algorithm requires a representation that allows fast iteration over all edges that are adjacent to a given node. In a linked list implementation each edge resides in two linked lists; one for each incident node. In our *adjacency array* representation each edge is represented twice in an array with $2m$ entries such that the edges adjacent to each source node are stored contiguously. For each edge, the target node and weight is stored. In terms of space requirements, each source and each target is stored once, and only the weight is duplicated. A second array of size n holds for each node a pointer to the beginning of its adjacency array.

The I-Max-Filter algorithm, on the other hand, can be implemented to work well with any representation that allows sampling edges in time linear

[2]Note that it would be counterproductive to exempt the nodes in E' from filtering because this would require an extra test for each edge or we would have to compute $E - E'$ explicitly during sampling.

[3]The original implementation [15] uses linked lists which were quite appropriate at the time, when cache effects were less important.

in the sample size and that allows fast iteration over all edges. In particular, it is sufficient to store each edge once. Our implementation for I-Max-Filter uses an array in which each edge appears once as (u, v) with $u < v$ and the edges are sorted by source node (u).[4] Only for the two small graphs for which the JP-algorithm is called it generates an adjacency array representation (see Figure 3).

To get a fair comparison we decided that each algorithm gets the original input in its "favorite" representation. This decision favors JP because the conversion from an edge array to an adjacency array is much more expensive than vice versa. Furthermore, I-Max-Filter could run on the adjacency array representation with only a small overhead: during the sampling and filtering stages it would use the adjacency array while ignoring edges (u, v) with $u > v$.

3.2 Filtering Access Pattern

In the implementation, we access the interval maxima data structure by JP order of source node rather than by the order in which the edges happen to be stored. In Appendix A.1 we explain why this increases the cache efficiency of these accesses. With the graph representation we use, this access pattern adds one irregular cache access per node, when accessing the first edge of a node's list. In order for the optimization to be beneficial, these n additional irregular accesses need to be compensated by the more regular accesses to the table. For very small densities, then, we might lose. In the results reported here (for graphs with up to 10,000 nodes), this access sequence resulted in a speedup of about 5 percent. For graphs with more nodes, the table is larger and so is the impact of this heuristic. For instance, on graphs with 25,000 nodes and just over 31,000,000 edges we observed a speedup of 11 percent on the SUN. All reported execution times are with this optimization enabled.

3.3 Implementation on Vector-Machines

A vector-machine has the capability to perform operations on vectors (instead of scalars) of some fixed size (in current vector-machines 256 or 512 elements) in one instruction. Vector-instructions typically include arithmetic and boolean operations, memory access instructions (consecutive, strided, and indirect), and special instructions like prefix-summation and minimum search. Vectorized memory accesses circumvent the cache. The filtering loop

[4]These requirements could be dropped at very small cost. In particular, I-Max-Filter can work efficiently with a completely unsorted edge array or with an adjacency array representation that stores each edge only in one direction. The latter only needs space for $m + n$ node indices and m edge weights.

of Figure 3 can readily be implemented on a vector-machine. The edges are stored consecutively in an array and can immediately be accessed in a vectorized loop; indirect memory access makes vectorized lookup of source and target vertices possible. For the filtering itself, bitwise exclusive or and two additional table lookups in the preSuf array are necessary. Using the prefix-summation capabilities, the edges that are not filtered out are stored consecutively in a new edge array. Also the construction of the preSuf data-structure can be vectorized. The only possibility for vectorization in the JP-MST algorithm is the loop that scans and updates adjacent vertices of the vertex just added to the MST. We divide this loop into a scanning loop which collects the adjacent vertices for which a priority queue update is needed, and an update loop performing the actual priority queue updates. Using prefix-summation the scanning loop can immediately be vectorized. For the update there is little hope, unless a favorable data structure allowing simultaneous decrease-key operations can be devised.

3.4 Graph Types

Both algorithms, JP and I-Max-Filter were implemented in C++ and compiled using GNU `g++ version 3.0.4` with optimization level `-O6`. We use a SUN-Fire-15000 server with 900 MHz UltraSPARC-III+ processors. In Appendix B.1 we also give measurements on a Dell Precision 530 workstation with 1.7 GHz Intel P4 Xeon processors that show similar results. Source codes are available at `http://www.mpi-sb.mpg.de/~sanders/dfg/iMax.tgz`.

We performed measurements with four different families of graphs, each with adjustable *edge density* $\rho = 2m/n(n-1)$. This includes all the families in [15] that admit dense inputs. A test instance is defined by three parameters: the graph type, the number of nodes and the density of edges (the number of edges is computed from these parameters). Each reported result is the average of ten executions of the relevant algorithm; each on a different randomly generated graph with the given parameters. Furthermore, the I-Max-Filter algorithm is randomized because the sample graph is selected at random. Despite the randomization, the variance of the execution times within one test was consistently very small (less than 1 percent), hence we only plot the averages.

Worst-Case: $\rho \cdot n(n-1)/2$ edges are selected at random and the edges are assigned weights that cause JP to perform as many Decrease Key operations as possible [15].

Linear-Random: $\rho \cdot n(n-1)/2$ edges are selected at random. Each edge (u,v) is assigned the weight $w(u,v) = |u-v|$ where u and v are the integer

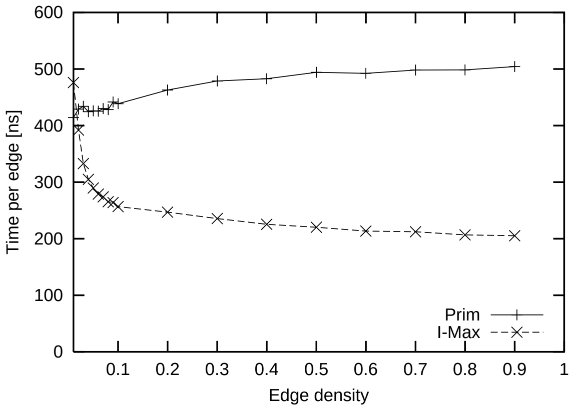

Figure 4: Worst-Case graph, 10000 nodes, SUN.

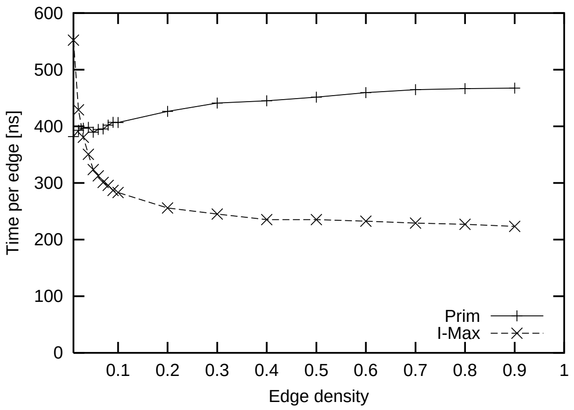

Figure 5: Linear-Random graph, 10000 nodes, SUN.

IDs of the nodes.
Uniform-Random: $\rho \cdot n(n-1)/2$ edges are selected at random and each is assigned an edge weight which is selected uniformly at random.
Random-Geometric:[15] Nodes are random 2D points in a $1 \times y$ rectangle for some stretch factor $y > 0$. Edges are between nodes with Euclidean distance at most α and the weight of an edge is equal to the distance between its endpoints. The parameter α indirectly controls density whereas the stretch factor y allows us to interpolate between behavior similar to class Uniform-Random and behavior similar to class Linear-Random.

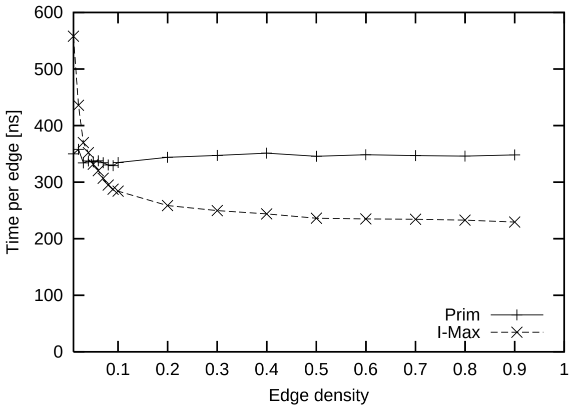

Figure 6: Uniform-Random graph, 10000 nodes, SUN.

3.5 Results on Microprocessors

Figures 4–6 show execution times per edge on the SUN for the three graph families Worst-Case, Linear-Random and Uniform-Random for $n = 10000$ nodes and varying density. We can see that I-Max-Filter is up to 2.46 times faster than JP. This is not only for the "engineered" Worst-Case instances but also for Linear-Random graphs. The speedup is smaller for Uniform-Random graphs. On the Pentium 4 (see Appendix B.1) JP is even faster than I-Max-Filter on the Uniform-Random graphs. The reason is that for "average" inputs JP needs to perform only a sublinear number of decrease-key operations so that the part of code dominating the execution time of JP is scanning adjacency lists and comparing the weight of each edge with the distance of the target node from the current MST. There is no hope to be significantly faster than that. On the other hand, we observed a speedup of up to a factor of 3.35 on dense Worst-Case graphs. Hence, when we say that I-Max-Filter outperforms JP this is with respect to space consumption, simplicity of input conventions and worst-case performance guarantees rather than average case execution time.

On very sparse graphs, I-Max-Filter is up to two times slower than JP, because $\sqrt{mn} = \Theta(m)$ and as a result both the sample graph and the graph that remains after the filtering stage are not much smaller than the original graph. Hence, the runtime is equivalent to two runs of JP on the input.

Appendix B.2 includes similar plots for Random-Geometric graphs with different stretch factors y. When the area from which node locations are selected is close to a square, the behavior of the MST algorithms is similar to that on the Uniform-Random graphs. As the stretch factor increases, the graph becomes closer to a Linear-Random graph. This is reflected in the

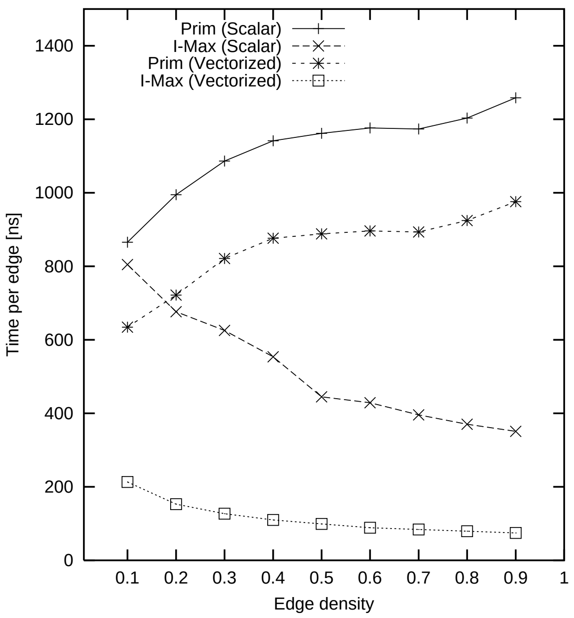

Figure 7: Worst-Case graph, 10000 nodes, NEC SX-5

results, which show that the benefit from filtering increases with the stretch.

3.6 Results On A Vector Machine

Figures 7–9 show similar measurements on a NEC SX-5 vector computer. For each of the two algorithms (JP and I-Max-Filter), runtimes per edge are plotted for scalar as well as vectorized version. The results of the scalar code show, once again, that JP is very fast on Uniform-Random graphs while I-Max-Filter is faster on the difficult graphs. In addition, we can see that on the "difficult" inputs I-Max-Filter benefits more than JP from vectorization. This is to be expected; JP becomes less vectorizable when many decrease key operations are performed, while the execution time of I-Max-Filter is dominated by the filtering stage, which in turn is not sensitive to the graph type. As a consequence, we see a speedup of up to 13 on the "difficult" graphs [5].

[5]comparing the vectorized versions of JP and I-Max-Filter.

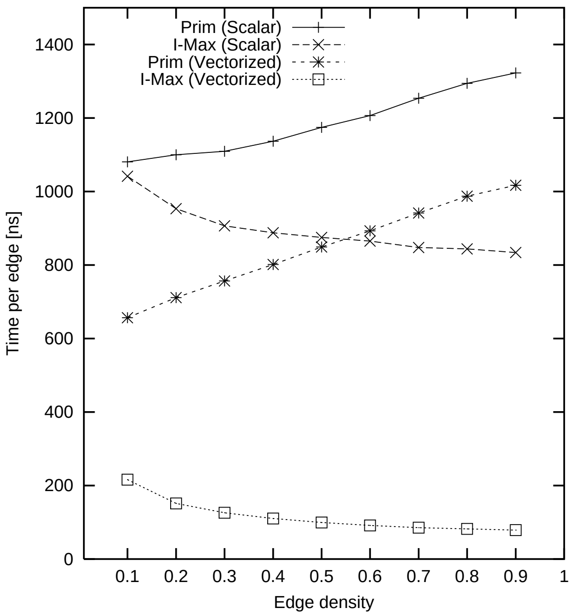

Figure 8: Linear-Random graph, 10000 nodes, NEC SX-5

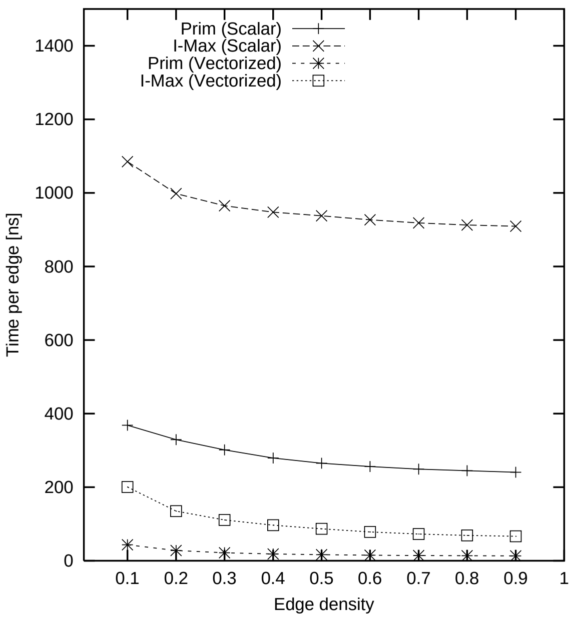

Figure 9: Uniform-Random graph, 10000 nodes, NEC SX-5

4 Conclusions

We have seen that the cycle property can be practically useful to design improved MST algorithms for rather dense graphs. An open question is whether we can find improved practical algorithms for sparse graphs that use further ideas from the asymptotically best theoretical algorithms. One issue is whether reducing the number of nodes based on Boruvka's [4, 16] algorithm has competitive speed. On current machines this seems a bit unlikely for sequential internal memory algorithms. But node reduction has great potential for parallel and external-memory implementations.

References

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

[2] R. K. Ahuja, R. L. Magnanti, and J. B. Orlin. *Network Flows*. Prentice Hall, 1993.

[3] N. Alon and B. Schieber. Optimal preprocessing for answering on-line product queries. Technical Report TR 71/87, Tel Aviv University, 1987.

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

[5] M. L. Fredman. On the efficiency of pairing heaps and related data structures. *Journal of the ACM*, 46(4):473–501, July 1999.

[6] T. C. Hu. The maximum capacity route problem. *Operations Research*, 9:898–900, 1961.

[7] J. Jájá. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.

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

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

[10] V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18:263–270, 1997.

[11] P. N. Klein and R. E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. In *Proceedings of the Twenty-Sixth Annual ACM Symposium on the Theory of Computing*, pages 9–15, Montréal, Québec, Canada, 23–25 May 1994.

[12] J. Komlós. Linear verification for spanning trees. In IEEE, editor, *25th annual Symposium on Foundations of Computer Science, October 24–26, 1984, Singer Island, Florida*, pages 201–206, 1109 Spring Street, Suite 300, Silver, 1984. IEEE Computer Society Press. IEEE catalog no. 84CH2085-9.

[13] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.

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

[15] B. M. E. Moret and H. D. Shapiro. An empirical analysis of algorithms for constructing a minimum spanning tree. In *Workshop Algorithms and Data Structures (WADS)*, number 519 in LNCS, pages 400–411. Springer, August 1991.

[16] Nesetril, Milkova, and Nesetrilova. Otakar boruvka on minimum spanning tree problem: Translation of both the 1926 papers, comments, history. *DMATH: Discrete Mathematics*, 233, 2001.

[17] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. In *27th ICALP*, volume 1853 of *LNCS*, pages 49–60. Springer, 2000.

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

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

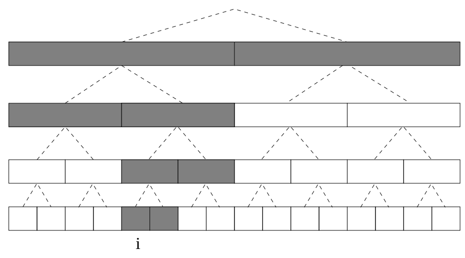

Figure 10: The active set for source node with jpNum $= i$.

A Algorithmic Refinements

A.1 Cache Efficiency

By carefully selecting the order in which the edges are filtered, we can reduce the space requirements of the interval maxima data structure from $\mathcal{O}(n \log n)$ to $\mathcal{O}(n)$. Assume that the edges are stored as pairs (i, j) and that they are sorted by source node (i). We propose to filter the edges in the order that their source nodes were inserted by the JP algorithm.

Let the *active set* A_i be the set of prefix and suffix arrays that can be accessed while filtering the edges (u, v) such that jpNum$[u] = i$ (see Figure 10). Note that each active set contains at most $\log n$ arrays corresponding to the source node and $\log n$ arrays corresponding to the target node: from each layer, one prefix array to the right of the source node and one suffix array to its left are active. When filtering iterates over the edges by nondecreasing jpNum of the source node i, each prefix or suffix array becomes active in A_i once, stays active for a while, and then becomes inactive forever. This means that the arrays can be generated on-the-fly instead of in a preprocessing stage such that each prefix or suffix array is generated at most once and not more than $2n$ space is required at a time.

Even if the whole $\mathcal{O}(n \log n)$ table is calculated in a preprocessing stage, this observation gives us a way to improve cache efficiency: filtering the edges in the order described above reduces the irregularity of cache accessed such that at any point in time, $\mathcal{O}(n)$ active entries are in cache.

A.2 Implicitly Defined Graphs

Many applications of MSTs work with complete graphs that are defined implicitly by an oracle function that returns the edge weight for any pair of nodes [2]. In this case our algorithm can be implemented to work with linear space: Run JP on an implicitly defined sample of the graph by picking sample edges with source v only when v is inserted into the tree. For the filtering stage, we are free to iterate over the edges (u, v) such that $(\text{jpNum}[u], \text{jpNum}[v])$ are visited in increasing lexicographic order. This not only allows us to compute lookup tables just in time as described in Appendix A.1 but also means that these arrays are just scanned leading to only $\mathcal{O}(n + n^2/B)$ cache faults overall for cache blocks of size B. Furthermore, the inner loop from Figure 3 can be rewritten in such a way that most values are kept in registers. Only computing the prefix maximum for the target node will require a single table lookup. Edges that are not filtered out are not stored but immediately inserted into the MST of edges seen so far. Using dynamic trees this can be implemented to run in $\mathcal{O}(n)$ space and $\mathcal{O}(\log n)$ time per opertation [1, 19]. All in all, we get an $\mathcal{O}(n^2)$ time $\mathcal{O}(n)$ space algorithm for implicitly defined graphs with very favorable constant factors.

A.3 All-Pairs Minimax Shortest Paths

A *minimax shortest path* from u to v is a path P from u to v that minimizes the weight of the heaviest edge on P. An important application of minimum spanning trees is the observation that a minimax shortest path can be obtained by taking the unique path from u to v in the minimum spanning tree [2, 6]. In particular, the heaviest edge weight on this path can be computed in constant time using $\mathcal{O}(n \log n + m)$ preprocessing time by running the JP algorithm on the input and constructing the lookup tables described in Sections 2.1 and 2.2. Our contribution here is a very simple method with better constant factors for the queries.

B More Experimental Results

B.1 Results on an Intel processor

Figures 11–13 show execution time on a PC per edge for the three graph families Worst-Case, Linear-Random, and Uniform-Random, for $n = 7000$ nodes and varying density (Currently this machine lacks sufficient memory for reliable measurements with $n = 10000$).

B.2 Random-Geometric graphs

Figures 14–16 show execution time per edge on a PC for three families of Random-Geometric graphs; with stretch factors $y = 2, 160$ and 500. In the first family, the nodes are spread in something close to a square, and the MST algorithms behave as on the Uniform-Random graphs; JP is faster because

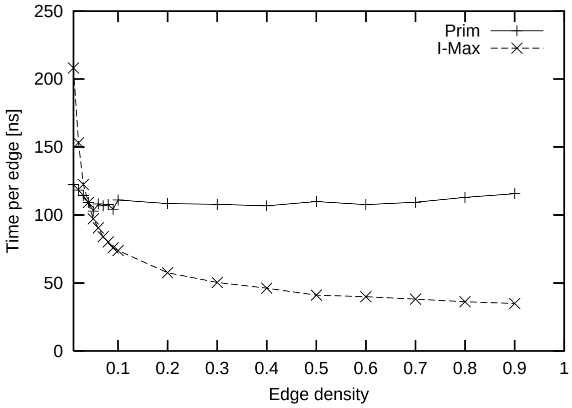

Figure 11: Worst-Case graph, 7000 nodes, PC.

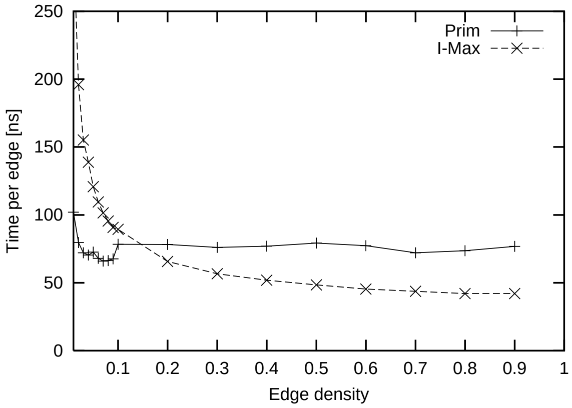

Figure 12: Linear-Random graph, 7000 nodes, PC.

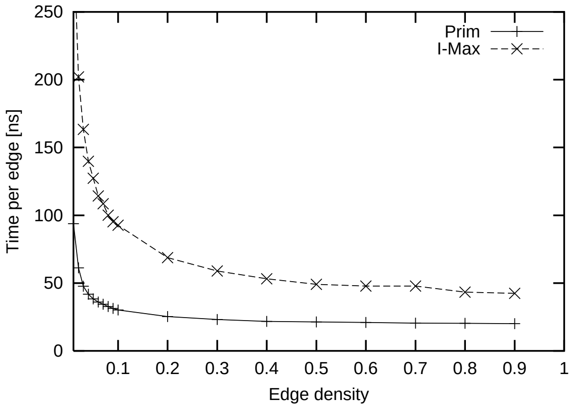

Figure 13: Uniform-Random graph, 7000 nodes, PC.

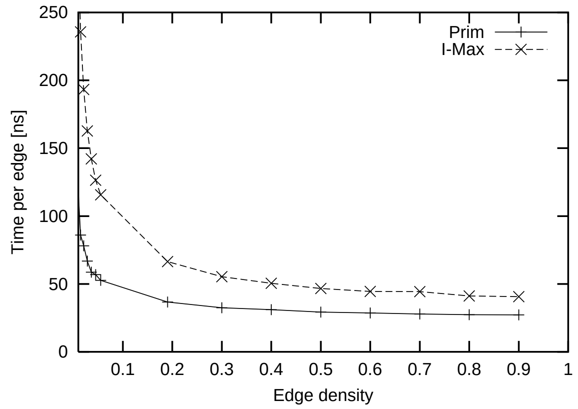

Figure 14: Random-Geometric graph, stretch factor 2, 7000 nodes, PC.

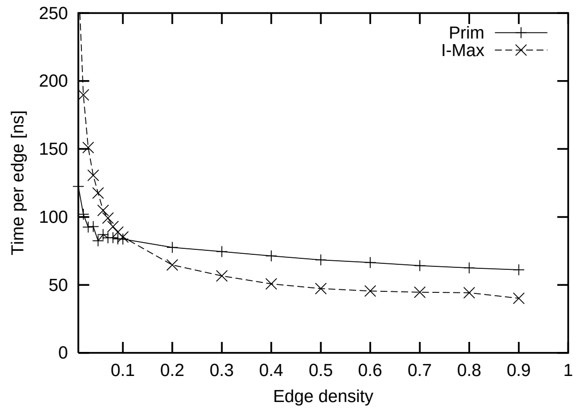

Figure 15: Random-Geometric graph, stretch factor 160, 7000 nodes, PC.

there are few decrease keys operations. As y increases, the graphs become closer to the Linear-Random family and the plots, accordingly, reflect an increasing gain from filtering.

B.3 Larger graphs with fixed density

Figures 17 and 18 show the effect of increasing the size of a Linear-Random graph while keeping the density fixed at 0.1. The results show again that I-Max-Filter is faster than JP on large graphs and that I-Max-Filter benefits more from the vector machine. Furthermore, these effects become more significant as the graph size increases.

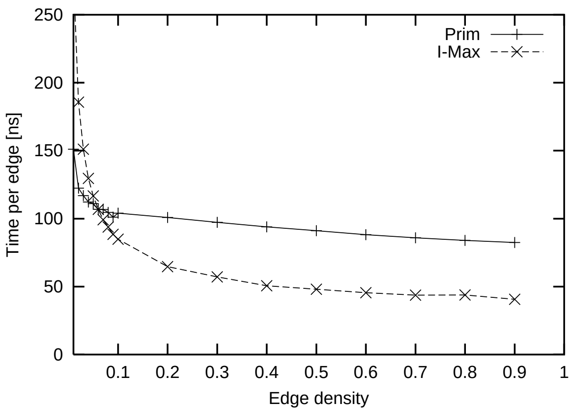

Figure 16: Random-Geometric graph, stretch factor 500, 7000 nodes, PC.

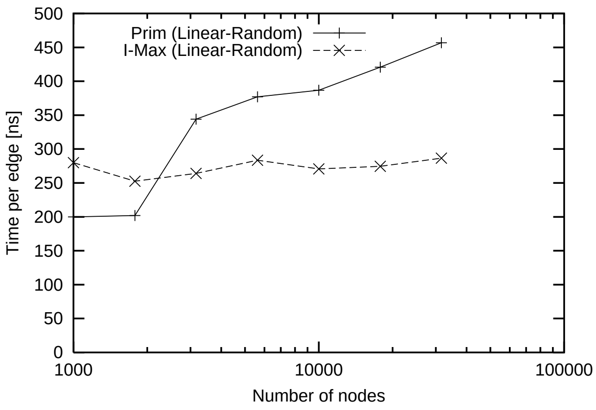

Figure 17: Linear-Random graph, density 0.1, SUN.

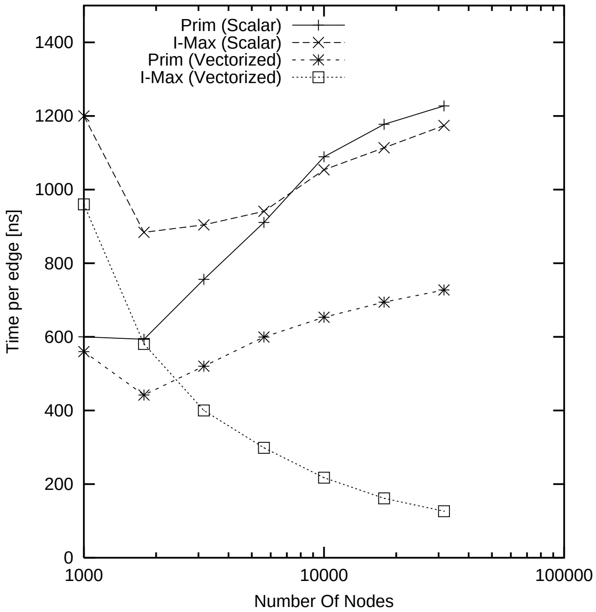

Figure 18: Linear-Random graph, density 0.1, NEC SX-5.

Graph Type	Edge Density	Filter Time (sec)	Total Time (sec)
Uniform-Random	0.5	4.75	6.26
Uniform-Random	0.9	8.80	10.70
Linear-Random	0.5	4.56	5.90
Linear-Random	0.9	8.72	10.36
Worst-Case	0.5	4.15	5.60
Worst-Case	0.9	7.73	9.34

Table 1: Filtering time compared to other stages. All graphs are with 10000 nodes.

B.4 Lower Order Terms Of The I-Max-Filter algorithm

Table 1 shows the runtime on a SUN of the filtering stage as well as the total running time of the I-Max-Filter algorithm, for several instances. The difference between the two figures is the time required for generating a sample of the edges, converting it to adjacency list form, running JP on it and after the filter stage, converting the remaining edges into adjacency list form and running JP on them. The results indicate that the filtering stage strongly dominates the execution time.

I N F O R M A T I K

Below you find a list of the most recent technical reports of the Max-Planck-Institut für Informatik. They are available by anonymous ftp from `ftp.mpi-sb.mpg.de` under the directory `pub/papers/reports`. Most of the reports are also accessible via WWW using the URL `http://www.mpi-sb.mpg.de`. If you have any questions concerning ftp or WWW access, please contact `reports@mpi-sb.mpg.de`. Paper copies (which are not necessarily free of charge) can be ordered either by regular mail or by e-mail at the address below.

Max-Planck-Institut für Informatik
Library
attn. Anja Becker
Stuhlsatzenhausweg 85
66123 Saarbrücken
GERMANY
e-mail: `library@mpi-sb.mpg.de`

MPI-I-2002-4-002	F. Drago, W. Martens, K. Myszkowski, H. Seidel	?
MPI-I-2002-4-001	M. Goesele	Tutorial Notes ACM SM 02 A Framework for the Acquisition, Processing and Interactive Display of High Quality 3D Models
MPI-I-2002-2-008	W. Charatonik, J. Talbot	Atomic Set Constraints with Projection
MPI-I-2002-2-007	W. Charatonik, H. Ganzinger	Symposium on the Effectiveness of Logic in Computer Science in Honour of Moshe Vardi
MPI-I-2002-1-008	P. Sanders, J.L. Träff	The Factor Algorithm for All-to-all Communication on Clusters of SMP Nodes
MPI-I-2002-1-002	F. Grandoni	Incrementally maintaining the number of l-cliques
MPI-I-2002-1-001	T. Polzin, S. Vahdati	Using (sub)graphs of small width for solving the Steiner problem
MPI-I-2001-4-005	H.P.A. Lensch, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing and Interactive Display of High Quality 3D Models
MPI-I-2001-4-004	S.W. Choi, H. Seidel	Linear One-sided Stability of MAT for Weakly Injective Domain
MPI-I-2001-4-003	K. Daubert, W. Heidrich, J. Kautz, J. Dischler, H. Seidel	Efficient Light Transport Using Precomputed Visibility
MPI-I-2001-4-002	H.P.A. Lensch, J. Kautz, M. Goesele, H. Seidel	A Framework for the Acquisition, Processing, Transmission, and Interactive Display of High Quality 3D Models on the Web
MPI-I-2001-4-001	H.P.A. Lensch, J. Kautz, M. Goesele, W. Heidrich, H. Seidel	Image-Based Reconstruction of Spatially Varying Materials
MPI-I-2001-2-006	H. Nivelle, S. Schulz	Proceeding of the Second International Workshop of the Implementation of Logics
MPI-I-2001-2-005	V. Sofronie-Stokkermans	Resolution-based decision procedures for the universal theory of some classes of distributive lattices with operators
MPI-I-2001-2-004	H. de Nivelle	Translation of Resolution Proofs into Higher Order Natural Deduction using Type Theory
MPI-I-2001-2-003	S. Vorobyov	Experiments with Iterative Improvement Algorithms on Completely Unimodel Hypercubes
MPI-I-2001-2-002	P. Maier	A Set-Theoretic Framework for Assume-Guarantee Reasoning
MPI-I-2001-2-001	U. Waldmann	Superposition and Chaining for Totally Ordered Divisible Abelian Groups

MPI-I-2001-1-007	T. Polzin, S. Vahdati	Extending Reduction Techniques for the Steiner Tree Problem: A Combination of Alternative-and Bound-Based Approaches
MPI-I-2001-1-006	T. Polzin, S. Vahdati	Partitioning Techniques for the Steiner Problem
MPI-I-2001-1-005	T. Polzin, S. Vahdati	On Steiner Trees and Minimum Spanning Trees in Hypergraphs
MPI-I-2001-1-004	S. Hert, M. Hoffmann, L. Kettner, S. Pion, M. Seel	An Adaptable and Extensible Geometry Kernel
MPI-I-2001-1-003	M. Seel	Implementation of Planar Nef Polyhedra
MPI-I-2001-1-002	U. Meyer	Directed Single-Source Shortest-Paths in Linear Average-Case Time
MPI-I-2001-1-001	P. Krysta	Approximating Minimum Size 1,2-Connected Networks
MPI-I-2000-4-003	S.W. Choi, H. Seidel	Hyperbolic Hausdorff Distance for Medial Axis Transform
MPI-I-2000-4-002	L.P. Kobbelt, S. Bischoff, K. Kähler, R. Schneider, M. Botsch, C. Rössl, J. Vorsatz	Geometric Modeling Based on Polygonal Meshes
MPI-I-2000-4-001	J. Kautz, W. Heidrich, K. Daubert	Bump Map Shadows for OpenGL Rendering
MPI-I-2000-2-001	F. Eisenbrand	Short Vectors of Planar Lattices Via Continued Fractions
MPI-I-2000-1-005	M. Seel, K. Mehlhorn	Infimaximal Frames: A Technique for Making Lines Look Like Segments
MPI-I-2000-1-004	K. Mehlhorn, S. Schirra	Generalized and improved constructive separation bound for real algebraic expressions
MPI-I-2000-1-003	P. Fatourou	Low-Contention Depth-First Scheduling of Parallel Computations with Synchronization Variables
MPI-I-2000-1-002	R. Beier, J. Sibeyn	A Powerful Heuristic for Telephone Gossiping
MPI-I-2000-1-001	E. Althaus, O. Kohlbacher, H. Lenhof, P. Müller	A branch and cut algorithm for the optimal solution of the side-chain placement problem
MPI-I-1999-4-001	J. Haber, H. Seidel	A Framework for Evaluating the Quality of Lossy Image Compression
MPI-I-1999-3-005	T.A. Henzinger, J. Raskin, P. Schobbens	Axioms for Real-Time Logics
MPI-I-1999-3-004	J. Raskin, P. Schobbens	Proving a conjecture of Andreka on temporal logic
MPI-I-1999-3-003	T.A. Henzinger, J. Raskin, P. Schobbens	Fully Decidable Logics, Automata and Classical Theories for Defining Regular Real-Time Languages
MPI-I-1999-3-002	J. Raskin, P. Schobbens	The Logic of Event Clocks
MPI-I-1999-3-001	S. Vorobyov	New Lower Bounds for the Expressiveness and the Higher-Order Matching Problem in the Simply Typed Lambda Calculus
MPI-I-1999-2-008	A. Bockmayr, F. Eisenbrand	Cutting Planes and the Elementary Closure in Fixed Dimension
MPI-I-1999-2-007	G. Delzanno, J. Raskin	Symbolic Representation of Upward-closed Sets
MPI-I-1999-2-006	A. Nonnengart	A Deductive Model Checking Approach for Hybrid Systems
MPI-I-1999-2-005	J. Wu	Symmetries in Logic Programs
MPI-I-1999-2-004	V. Cortier, H. Ganzinger, F. Jacquemard, M. Veanes	Decidable fragments of simultaneous rigid reachability
MPI-I-1999-2-003	U. Waldmann	Cancellative Superposition Decides the Theory of Divisible Torsion-Free Abelian Groups
MPI-I-1999-2-001	W. Charatonik	Automata on DAG Representations of Finite Trees
MPI-I-1999-1-007	C. Burnikel, K. Mehlhorn, M. Seel	A simple way to recognize a correct Voronoi diagram of line segments
MPI-I-1999-1-006	M. Nissen	Integration of Graph Iterators into LEDA
MPI-I-1999-1-005	J.F. Sibeyn	Ultimate Parallel List Ranking ?
MPI-I-1999-1-004	M. Nissen, K. Weihe	How generic language extensions enable “open-world” desing in Java
MPI-I-1999-1-003	P. Sanders, S. Egner, J. Korst	Fast Concurrent Access to Parallel Disks

MPI-I-1999-1-002	N.P. Boghossian, O. Kohlbacher, H.-. Lenhof	BALL: Biochemical Algorithms Library
MPI-I-1999-1-001	A. Crauser, P. Ferragina	A Theoretical and Experimental Study on the Construction of Suffix Arrays in External Memory
MPI-I-98-2-018	F. Eisenbrand	A Note on the Membership Problem for the First Elementary Closure of a Polyhedron