A Practical Minimum Spanning Tree Algorithm Using the Cycle Property*

Irit Katriel[1], Peter Sanders[1], Jesper Larsson Träff[2]

[1] Max-Planck-Institut für Informatik, Saarbrücken, Germany
{irit,sanders}@mpi-sb.mpg.de
[2] C&C Research Laboratories, NEC Europe Ltd., Sankt Augustin, Germany
traff@ccrl-nece.de

Abstract. We present a simple new (randomized) 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 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 10 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.

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 [2, 5, 10, 13] and are all based on the *cut property*: a *lightest* edge leaving a set of nodes can be used for an MST. More specifically, Kruskal's algorithm [10] 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 [5, 15]. Using Fibonacci heap priority queues, its execution time is $\mathcal{O}(n \log n + m)$. Using pairing heaps [3] Moret and Shapiro [12] get quite favorable results in practice at the price of worse performance guarantees.

On the theoretical side there is a randomized linear time algorithm [6] and an almost linear time deterministic algorithm [14]. 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 cut property with the *cycle property*: a *heaviest edge* in any cycle is not needed for an MST.

* Partially supported by DFG grant SA 933/1-1.

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 10 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 [8] 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 [7] 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 [2, 13] 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 [9] 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 such that the LCA of two leaves can be found in constant time.

In Section 2 we develop a simpler filtering scheme that is based on the order in which the JP algorithm adds nodes to the MST 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 T' 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 maxima 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 [4]. 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 exclusive-or 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 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 10.

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 exclusive-or 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.

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 $e_{v'+1}, \ldots e_{v-1}$

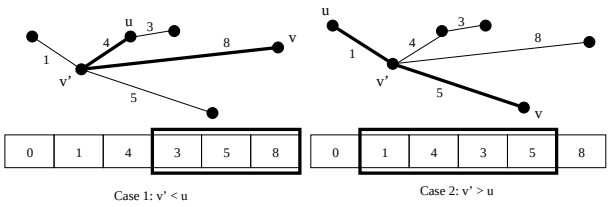

Fig. 1. Illustration of the two cases of Lemma 1. The JP algorithm adds the nodes from left to right.

because otherwise the JP algorithm would have added v, using e_v, earlier. There are two cases to consider (see Figure 1).
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 $w_u, \ldots, w_v$ 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 maximum for two nodes that are in two different components is ∞, as it should be.

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 [4, Section 3.4.3] which is a special case of the general method in [1]. This algorithm might be of 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 $\max a[i..j] = \max(u[i], w[j])$.

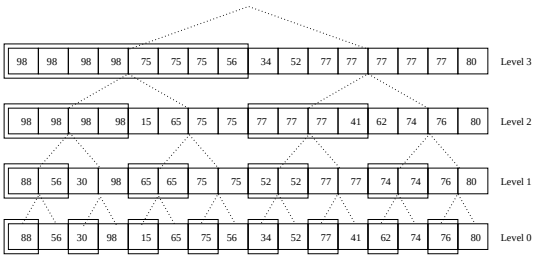

Fig. 2. Example of a layers array for interval maxima. The suffix sections are marked by an extra surrounding box.

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

$$\text{preSuf}[\ell][i] = \begin{cases} \max(a[2^\ell b..i]) & \text{if } b \text{ is odd} \\ \max(a[i..(2^\ell+1)b-1]) & \text{otherwise} \end{cases}$$

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

Furthermore, the interval boundaries can be used to index the arrays. We simply have $\max a[i..j] = \max(\text{preSuf}[\ell][i], \text{preSuf}[\ell][j])$ where $\ell = \text{msbPos}(i \oplus j)$; $\oplus$ is the bit-wise exclusive-or operation and $\text{msbPos}(x) = \lfloor \log_2 x \rfloor$ is equal to the position of the most significant nonzero bit of x (starting at 0). Some architectures have this operation in hardware[3]; if not, $\text{msbPos}(x)$ can be stored in a table (of size n) and found by table lookup. Layer 0 is identical to a. A further optimization stores a pointer to the array $\text{preSuf}[\ell]$ in the 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., $\text{preSuf}[\ell][i] = \max(a[i+1..(2^\ell+1)\lfloor i/2^\ell \rfloor - 1])$ if $\lfloor i/2^\ell \rfloor$ is even.

[3] One trick is to use the exponent in a floating point representation of x.

2.3 Putting the Pieces Together

Fig. 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* $mT_{\text{filter}} + \mathcal{O}(n \log n + \sqrt{nm})$ *where* T_{filter} *is the time required to query the filter about one edge.*

In particular, if $m = \omega(n \log^2 n)$, the execution time is $(1 + o(1))mT_{\text{filter}}$.

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}.[4] By the sampling lemma explained in the introduction [8, 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.

From a theoretical point of view it is instructive to compare the number of edge weight comparisons needed to find an MST with the obvious lower bound of m. Also in this respect we are quite good for dense graphs because the filter algorithm performs at most two comparisons with each edge that is filtered out. In addition, an edge is already filtered out if the first comparison in Fig. 3 fails. Hence, a more detailed analysis might well show that we approach the lower bound of m for dense graphs.

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

[4] 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.

```
(* 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'')
```

Fig. 3. The I-Max-Filter algorithm.

($m \gg n \log n$). We compare our implementation with a fast implementation of the JP algorithm. In [12] 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.[5] This implementation of JP consistently outperforms [12] and LEDA [11].

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 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).[6] Only for the two small graphs for which the JP-algorithm is called it generates an adjacency array representation (see Fig. 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

Our implementation filters all edges stored with a node together so that it is likely that accesses to data associated with this node resides in cache.

Furthermore, the nodes are processed in the order given by JP order. This has the effect that only $\mathcal{O}(n)$ entries of the $\mathcal{O}(n \log n)$ lookup table entries need

[5] The original implementation [12] uses linked lists which were quite appropriate at the time, when cache effects were less important.

[6] 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.

to be in cache at any time. 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 even larger graphs we have observed speedups of up to 11 % due to this optimization.

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 of Fig. 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; vectorized lookup of source and target vertices is possible by indirect memory access operations. 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 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. Measurements on a Dell Precision 530 workstation with 1.7 GHz Intel P4 Xeon processors show similar results. The vector machine used is a NEC SX-6. The SX-6 has a memory bandwidth of 32GBytes/second, and (vector) peak-performance of 8GFlops.

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 [12] 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 [12].
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 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:[12] 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.

3.5 Results on Microprocessors

Fig. 4 shows 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 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.

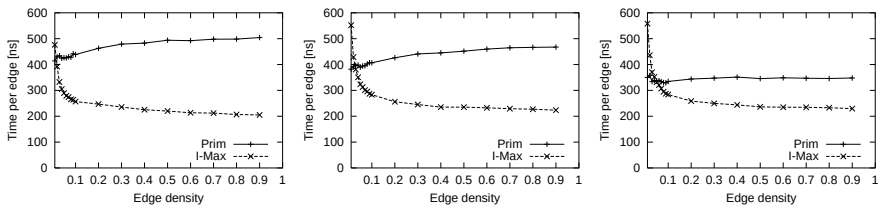

Fig. 4. Worst-Case, Linear-Random, and Uniform-Random graphs, 10000 nodes, SUN.

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. The runtime is therefore comparable to two runs of JP on the input.

3.6 Results On A Vector Machine

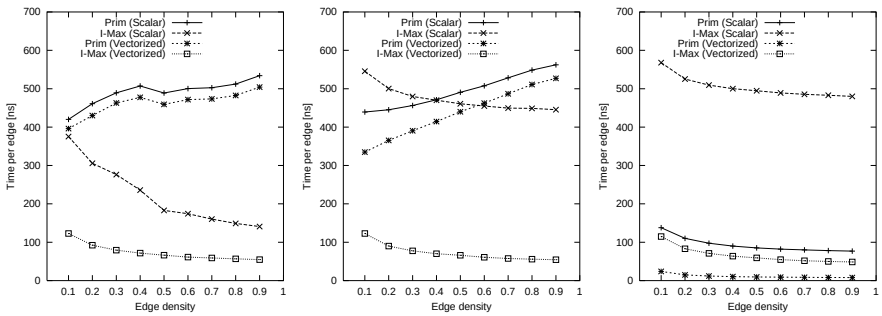

Fig. 5. Worst-Case, Linear-Random, and Uniform-Random graphs, 10000 nodes, NEC SX-6.

Fig. 5 shows measurements on a NEC SX-6 vector computer analogous to the microprocessor results reported in Fig. 4.

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 from vectorization more than JP which achieves a speedup of only factor 1.3. 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 10 on the "difficult" graphs when comparing the vectorized versions of JP and I-Max-Filter.

3.7 Can JP be made faster?

It is conceivable that the implementation of JP could be further improved using an even faster priority queue. Our implementation of JP uses the Pairing Heap variant that proved to be fastest in the comparative study of Moret and Shapiro [12]. How much can JP gain from an even faster heap? To investigate this we ran it with a best possible (theoretically impossible!) *perfect heap*, that

is, a heap in which both Decrease-Key and Delete-Minimum operations takes unit time. The *perfect heap* is implemented as an array, such that Decrease-Key takes constant time, and to simulate constant-time Delete-Minimum we simply stop the clock during this operation. Results for the worst-case graphs are shown in Fig. 6, which give both the run time break-down for I-Max-Filter, and the run time for I-Max-Filter with Pairing Heap and Perfect heap. The results show that I-Max-Filter is not very sensitive to the type of heap; its running time is dominated by the filtering stage which doesn't use the heap. JP is sensitive to the type of heap when running on graphs that incur many Decrease-Key operations, but not when it runs on a Uniform-Random graph (not shown here). All of this was to be expected, but in addition we see that I-Max-Filter is faster even when JP can access the heap almost for free and the only thing that takes time is traversing the nodes' adjacency lists.

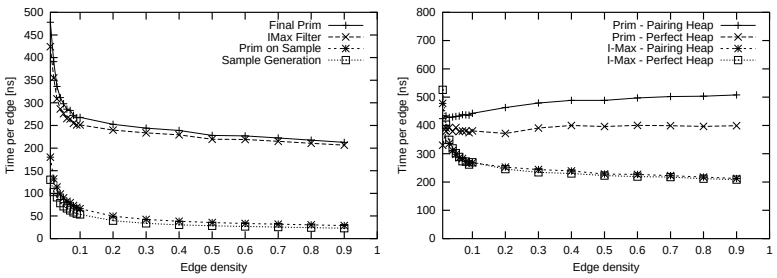

Fig. 6. Time break-down of I-Max-Filter (left). Pairing Heap vs. Perfect Heap (right). Worst-Case graph, 10,000 nodes, SUN.

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. Besides a component for filtering edges, these algorithms have a component for reducing the number of nodes based on Boruvka's [2, 13] algorithm. Although this algorithm is conceptually simple, it seems unlikely that it is useful for internal memory algorithms on current machines. However node reduction has great potential for parallel and external-memory implementations.

References

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

2. O. Boruvka. O jistém problému minimálním. *Pràce, Moravské Prirodovedecké Spolecnosti*, pages 1–58, 1926.
3. M. L. Fredman. On the efficiency of pairing heaps and related data structures. *Journal of the ACM*, 46(4):473–501, July 1999.
4. J. Jájá. *An Introduction to Parallel Algorithms*. Addison Wesley, 1992.
5. V. Jarník. O jistém problému minimálním. *Práca Moravské Přírodovědecké Společnosti*, 6:57–63, 1930.
6. D. Karger, P. N. Klein, and R. E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. *Journal of the ACM*, 42(2):321–329, 1995.
7. V. King. A simpler minimum spanning tree verification algorithm. *Algorithmica*, 18:263–270, 1997.
8. P. N. Klein and R. E. Tarjan. A randomized linear-time algorithm for finding minimum spanning trees. In *Proceedings of the 26th Annual ACM Symposium on the Theory of Computing*, pages 9–15, 1994.
9. J. Komlós. Linear verification for spanning trees. In *25th annual Symposium on Foundations of Computer Science*, pages 201–206, 1984.
10. 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.
11. K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
12. 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)*, volume 519 of *Lecture LNCS*, pages 400–411. Springer, 1991.
13. J. Nesetril, E. Milková, and H. Nesetrilová. Otakar Boruvka on minimum spanning tree problem: Translation of both the 1926 papers, comments, history. *Discrete Mathematics*, 233(1-3), 3–36, 2001.
14. S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM*, 49(1): 16–34, 2002.
15. R. C. Prim. Shortest connection networks and some generalizations. *Bell Systems Technical Journal*, pages 1389–1401, 1957.