DIMACS Series in Discrete Mathematics
and Theoretical Computer Science
Volume **00**, 0000

Implementing a Maximum Flow Algorithm: Experiments with Dynamic Trees

T. BADICS AND E. BOROS

August 15, 1992

ABSTRACT. This paper contains experimental results with a maximum flow algorithm developed by Cheriyan and Hagerup. Comparison was made to a preflow-push type maxflow implementation, and we examined also the effect of the dynamic trees data structure and several heuristic parameters.

1. Introduction

In this paper we report on an implementation of a maximum flow algorithm by Cheriyan and Hagerup [**1**]. Our aim was to test the behavior of this algorithm in practice, concerning it's good theoretical worst case bound. We were particularly interested in the effect of using theoretically well behaving data structures such as dynamic trees [**15**], and Fibonacci heaps [**17**]. We also made comparisons to two preflow-push based algorithm by Goldberg and Tarjan [**13**], and to an implementation of pushes along several edges without using dynamic trees.

In Sections 2. the basic notations and a short description of the Cheriyan–Hagerup algorithm are given. In Section 3. details of the implementation and the used data structures are presented. In Section 4. a summary of the experimental results and a few questions that could be addressed in the future are given.

2. Basic notions and the PLED algorithm

We assume that the reader is familiar with the generic maximum flow algorithm in [**13**] and refer to [**13**] for definitions of the terms *network, source s, sink t, edge capacity $c(v,w)$, flow, maximum flow, preflow f, flow excess $e(v)$ of a vertex v, residual graph, residual capacity $rescap(v,w)$ of the edge (v,w),*

1991 *Mathematics Subject Classification.* Primary 90B10; Secondary 68P05, 68-04.

The second author is supported in part by NSF (Grant DMS 89-06870) and AFOSR (Grants 89-0512 and 90-0008).

valid labeling d, active vertex, push, saturating push, and *nonsaturating push*. Let $G = (V, E)$ denote the digraph (assumed symmetric) corresponding to the network. Let $N = |V|$, $M = |E|$.

We call an edge (v, w) *eligible* if $rescap(v, w) > 0$ and $d(v) = d(w) + 1$. Each vertex $v \in V$ has an adjacency list consisting the edges $(v, w) \in E$. For each vertex v, the first eligible edge in the adjacency list of v is called its *current edge* and denoted by $ce(v)$ (such an edge may not exist, in which case $ce(v)$ refers to none).

Our implementation is based on the algorithm developed by Cheriyan and Hagerup (see [**1**]). Following [**1**] we shall refer to this algorithm as PLED (shorthand for Prudent Linking and Excess Diminishing). This algorithm is an instance (with one minor exception) of the generic preflow algorithm by Goldberg and Tarjan [**13**]. The difference (namely, the value of a push over an edge (v, w) may be less then both the excess at v and residual capacity of (v, w)) does not affect the essential properties of the generic algorithm (in particular, maximum flow is always correctly computed upon termination). PLED also uses an idea introduced by Ahuja and Orlin [**6**] of scaling the volume of the pushes. The scaling factor plays here, however, a slightly different role: the limits imposed on the volume of a push are not the same as in [**6**]. A third idea in PLED is randomization: after each relabeling of a vertex v, the edgelist of v is permuted randomly. This random permutation ensures a better theoretical running time. Noga Alon showed in [**5**] that this randomization can be replaced by a deterministic procedure. The worst case running time of PLED, using the randomized procedure is $O(NM + N^2(logN)^2)$ with high probability (see [**2, 3, 4**]). Using Alon's derandomization, the deterministic worst case bound improves to $O(NM + N^{8/3}(logN))$. The worst case bound without randomization is $O(NMlog(N))$. Randomly permuting the edgelist of the vertex after relabeling can take substantial amount of time. From the implementation point of view just one random permutation of the adjacency list of each vertex at the start of execution suffices for the best theoretical running time, though the probability of failure increases slightly, $exp(-\Theta(\sqrt{NM} + NlogN))$ versus $exp(-\Theta(N\sqrt{NM} + N^2logN))$, see [**1, 2**].

Three main data structures are essential for PLED.

- **Ordinary heap** that contains vertices which have big excesses and which are ordered by their distance labels. This structure supports the easy selection of a vertex for a push. (Select a vertex with the minimal distance label among the vertices having large enough excesses).
- **Fibonacci heap**(see [**17**]) contains the rest of vertices, ordered by their (small) excesses. This supports constant (amortized) time decrease key operation and fast update of the scaling factor.
- **Dynamic trees structure** (see [**15**]) to maintain a spanning forest F of G containing a subset of the current edges, where the value associated with an edge in F is its residual capacity. This structure is able to send

flow value along a path of length L in (amortized) time $O(logL)$

Sketch of the PLED algorithm:

PLED Algorithm
  initialize;
  loop(until all the excesses are eliminated)
    v := **select**;
    macropush(v);
  endloop;
end PLED

initialize;
  Perform the standard initialization of the variables f, d and e (see [**13**], pp.925) and of the dynamic trees, and Fibonacci heap data structures.
  $\Delta := max(\{e(v)|v \in V - \{s,t\}\} \cup \{0\});$

select: vertex;
  Return an active vertex v with $e(v) \geq \Delta$ and minimum $d(v)$ or decrease Δ and then select v as before.

macropush(v: vertex);
  Send flow from v over a path in the tree to which v belongs, or if v was a root then send flow from v using edges (v,w) for which $rescap(v,w) \leq$ **limflow**(v) until v is relabeled or until $limflow(v)$ decreases to $< \Delta/2$ due to saturating pushes. If $limflow(v) > \Delta/2$ then v is linked to the other end of its current edge, and a treepush is executed.

limflow(v : vertex): real;
  return(if $e(v) \geq 2\Delta$ then $e(v)/2$ else $e(v)$);

The bottleneck factor in virtually all preflow-push type maximum-flow algorithm is the number of nonsaturating pushes. One of the properties of PLED is to try a sequence of nonsaturating pushes along several eligible edges that constitute a path in a tree of the eligible edges while avoiding saturation on that path, rather then performing a nonsaturating push along one single edge. These trees of the eligible edges are stored in a dynamic trees structure, in which one can perform a push in logarithmic time of the length of the path. Beside this property, PLED uses a scaling parameter for controlling the size of the pushes and the trees in the dynamic trees forest. A Fibonacci heap structure handles this scaling parameter efficiently. Finally for achieving the theoretically best bound, we either have to perform a random permutation or we have to generate a permutation by the deterministic scheme of [**5**] at the start of execution. Theoretically by permuting the adjacency lists after each relabel one could reach the best bound with higher probability, but the amount of work to do this can easily be substantial even if the number of relabel operations is just $\Theta(N)$.

3. Implementation

Since the algorithm requires the above data structures to achieve the theoretically best performance, we decided to implement all of them. In this we were encouraged by earlier experiences (see [**19**]) with implementations of Fibonacci heaps, and Splay trees [**16**]. Splay trees play a main role in the dynamic trees structure. It was found in the experiments of [**19**] that a particular implementation of Splay trees was quite efficient comparing to some other search trees. In [**19**] Splay trees were compared to Binary, Red-Black and AVL trees and the Fibonacci heap was compared to Ordinary and Pairing heaps. Their study showed, that Fibonacci heaps are better than Ordinary heaps. This suggested that the use of Fibonacci heaps can have practical advantage, and that they can be implemented efficiently. So we implemented Fibonacci heaps and the dynamic trees data structure following [**15**].

Beside the above data structures, we implemented a routine for randomly permuting the edge lists of the vertices. Although the deterministic permutation of Alon derandomizes the algorithm, the overhead of such a permutation generating procedure is so large that we did not expect much improvement by implementing such a deterministic procedure. Moreover since our instances were mostly randomly generated, after some preliminary experiments we used the PLED algorithm without random permutations.

For comparison reasons we coded Goldberg's simple preflow-push algorithm using simple FIFO queue for selecting the active vertices, and the dynamic trees version of this algorithm described in the same paper ([**13**]). Later we refer to these codes as GOLD and GOLDYN respectively. In the code GOLDYN we did not control the size of the dynamic trees. Thus we got a code which has theoretical worst case bound $O(NMlogN)$ instead of $O(NMlog(N^2/M)$. The reason for this choice was mostly lack of time and the expected overhead of such a control mechanism. Besides, in the range of examples we tested the codes, the benefit from such size control, even without the overhead, probably is not significant.

For testing the performance of the dynamic trees data structure, we implemented its operations (*find-min, add-value, find-root, link, cut, etc.*) with storing the trees explicitly and executing these operations in the obvious (linear time) way. Hence we avoided the overhead of handling Splay trees and complicated updates. Later in this paper the codes using these "non-dynamic tree" operations are called NPLED and NGOLDYN. Note that the number of elementary operations for PLED and NPLED (or GOLDYN and NGOLDYN) are the same on the same instance, only the way of handling the tree operations are different. Therefore the difference in running time shows exactly the impact of the dynamic trees structure.

In our implementations of all the codes we employed an idea, mentioned in ([**13**]), the so called "global-" or "big-relabeling". Our early experiments showed clearly, that in PLED just like in GOLD or GOLDYN, the running times of the variant which uses global-relabeling were much smaller (orders of magnitude) than the one which does not use it. Therefore we built in some heuristic parameters controlling the calling frequency of big-relabeling and affecting thus the running time.

A "big-relabeling" step consist of two breadth-first-searches, one starting from the sink and working on the sink side of the residual graph, and another one for the source side, starting from the source. In these breadth-first-searches the shortest distance is calculated from each vertex to the sink on the sink side, or to the source on the source side, respectively. Unfortunately breadth-first-search is a relatively expensive operation (it takes O(M) steps), so the calling frequency of big-relabel is very important and can be a subject of later studies.

The effect of a big-relabeling is to lift all the vertices as high as possible, keeping the labeling feasible. This is the point where this expensive operation can help a lot. During the steps of these "preflow-push" type algorithms the label of a vertex v (say $d(v)$) is always a lower bound on the shortest distance from v to the sink if the sink is reachable from v in the residual graph, or $(d(v) - N)$ is a lower bound on the distance from v to the source. After each saturating push along the edge (v, w), v can either have another eligible edge for later pushes, or it will be relabeled, i.e. lifted right above its neighbors. In this latter case the deletion of the edge (v, w) cuts the tree (formed by some of the current edges) to which v belonged. All the vertices in the subtree, rooted at v, will keep their old labels, and so their real distance from the sink can become larger then the corresponding label values. Hence, after some iterations, there could be many such vertices whose label is far from their real distance. The algorithm will terminate only when all the remaining excesses are sent back to the source. Therefore vertices with positive excess have to be lifted above the source (having the label value N). Since one relabel lifts a vertex only above its neighbors, meaning an increase by 1 or 2 in its label in the average case, many simple relabel operations may be necessary right before termination. In the worst case for example, if the $s-t$ minimum cut $(V_s, V-V_s)$ with V_s minimal has $|V_s| = \Theta(N)$, then we may need $O(N^2)$ simple relabels just to terminate the algorithm. In practice, we found that this so called second phase of the algorithm can take almost all the running time (because the number of relabels became significant). In contrast only one execution of big-relabel step will lift ALL the vertices as high as possible by calculating the exact distances from each vertex to the sink or to the source, respectively. Of course, if our graph is dense, one big-relabel can also take $O(M)$ time, but each of these steps are much cheaper than a simple relabeling.

We built 3 parameters to control the calling frequency of the big-relabel procedure.

(i) **Relab-freq**, a nonnegative integer. A big-relabel is called after each **Relab-freq** iterations. By iteration we mean a vertex selection. (This is a simple heuristic to call big-relabel with some constant frequency.)

(ii) **Cut-freq**, a nonnegative integer. After this many cuts in the tree structure a big-relabel is called. (This parameter estimates the total sum of the differences between the actual labels of the vertices and their shortest distances to the sink or to the source, respectively. After a cut performed at the vertex v, not only the label of v, but all the labels of its descendants in the tree can loose their exact meaning. That is the label of a vertex w in the subtree of v can be less than the shortest distance from w to the sink, since the shortest distances may increase by cutting an edge. Hence, by calculating the number of cuts only, we are not taking into account all the labels in the subtree. Some further improvement could be done by making this estimation more precise.)

(iii) **Gap**, a switch which can be on or off. If on, it checks whether the labels of the vertices, not larger than $N-2$, form a consecutive sequence or not. If there is a gap (according to [**11**]), there is a cut in the residual graph, hence each vertex in the source side have to be lifted above the source, and therefore a big-relabel is called.

Our experiments showed that before the maxflow value is reached, parameters **Relab-freq** or **Cut-freq** can be helpful. Their effect seemed to be equivalent in the sense that in almost all problems we tested, the number of iterations was roughly four times more than the number of cuts. So by setting **Cut-freq** = **Relab-freq**/4 we could get about the same number of big-relabels. The use of the **Gap** parameter is extremely benefitial in this phase and its update is a very cheap operation. Therefore, in our experiments we always turned **Gap** on.

In the second phase, i.e. in the phase of sending back the excesses, **Relab-freq** or **Cut-freq** are useful, while **Gap** has no impact at all. The second phase can take quite a long time, in particular for problems with long paths from the source to the sink, and with large excess to be sent back to the source. Therefore, the use of the big-relabel is important in this final phase, too.

On the other hand, since big-relabel is a costly operation, we kept its calling frequency below a small linear function of the number of arcs, in order to keep the overall complexity at $O(NM)$.

We implemented another mechanism to achieve better running times in all three codes. Namely at initialization the algorithm calculates an upper bound U on the maximum flow value by taking the minimum of capacities of some cuts. Then it creates a new source by adding an artificial vertex S and a new arc (S, s) with capacity U to the network, where s was the old source. The new problem is obviously equivalent with the old one, and the extra cost of its implementation is negligible. The advantage of doing this is that we do not let the algorithm push too much excess into the network, reducing in this way the runtime of the second phase. We have found instances showing that without this procedure the running time was significantly bigger due to the long second phase.

4. Experimental results

For the experiments, we used the DIMACS suggested problems, and the generators GENRMF, WASHINGTON, and AC-MAX [**9, 8, 10**]. (See the DIMACS document "The Core Experiments"). The families of networks we report on include the ones suggested by "The Benchmark Experiments", and two classes of problems made intentionally very difficult for Goldberg's preflow-push algorithm.

The input parameters for the DIMACS benchmark problems were chosen, according to the structure of the particular networks, such that the number of nodes of the generated example increase approximately by multiples of 2 up to the memory limitations of our workstation.

4.1. Description of the problem classes. The generator GENRMF takes 4 numbers $(a, b, c_1 and c_2)$ as input parameters. The resulting network consists of b layers of $(a \times a)$ grids, in one of the corners of the first layer is the source, and in the opposite corner of the last layer is the sink. Arcs are drawn between each neighboring grid nodes in both directions with capacities $c_2 a^2$, and two consecutive layers are connected along a randomly generated perfect matching with capacities chosen randomly in the range (c_1, c_2).

The generator WASHINGTON takes several numbers as input parameters. The first argument chooses the type of the network, and the rest are specific to the type chosen.

The generator AC-MAX generates fully dense or sparse random acyclic networks. The only parameter is n, the number of nodes in the network. (see "The Core Experiments")

The following is a description of the network families what we tested on. Exact parameters for the families GL, GW, WLM and AD are given in Table 1.

- **GL - Genrmf-Long**. Generated by the GENRMF code. They are called "long", because b is approximately a^2.
- **GW - Genrmf-Wide**. Generated by the GENRMF code. They are called "wide", because a is approximately b^2.
- **WLM - Washington-Line-Moderate**. Fairly dense networks, generated by the WASHINGTON code. The network type is "6", and the type specific parameters are n, m, deg. The resulting network has $n \times m$ nodes in a grid, plus two additional nodes for the source and sink. The source is connected to the first m nodes of the grid, and the last m nodes are connected to the sink by arcs with infinite capacities. For each node

in the grid, *deg* arcs are generated with random capacities pointing to *deg* randomly selected nodes from the next *deg* columns.

- **AD - Acyclic-Dense.** These instances are fully dense acyclic graphs with random capacities, generated by the AC-MAX code.

TABLE 1. Parameters used for generating the DIMACS Benchmark instances.

NAME	PARAMETERS	NODES	ARCS
GL1	a=6 b=31 c_1=1 c_2=10000	1116	4800
GL2	a=7 b=42 c_1=1 c_2=10000	2058	9065
GL3	a=8 b=64 c_1=1 c_2=10000	4096	18368
GL4	a=9 b=100 c_1=1 c_2=10000	8100	36819
GL5	a=11 b=128 c_1=1 c_2=10000	15488	71687
GL6	a=13 b=194 c_1=1 c_2=10000	32786	153673
GW1	a=16 b=4 c_1=1 c_2=10000	1024	4608
GW2	a=21 b=5 c_1=1 c_2=10000	2205	10164
GW3	a=28 b=5 c_1=1 c_2=10000	3920	18256
GW4	a=37 b=6 c_1=1 c_2=10000	8214	38813
GW5	a=48 b=7 c_1=1 c_2=10000	16128	76992
GW6	a=64 b=8 c_1=1 c_2=10000	32768	157696
WLM1	6 64 4 5	258	1236
WLM2	6 128 4 8	514	3975
WLM3	6 256 4 8	1026	8069
WLM4	6 512 4 11	2050	22279
WLM5	6 1024 4 16	4098	65019
WLM6	6 2048 4 22	8194	179258
AD1	fully dense	256	32640
AD2	fully dense	321	51360
AD3	fully dense	403	81003
AD4	fully dense	507	128271
AD5	fully dense	636	201930
AD6	fully dense	800	319600

- **GB - Goldberg-Bad-Case**. A deterministic family, generated by the WASHINGTON code. The type is "10", and has one parameter n. The structure is described in Figure 1. This family was designed especially difficult for Goldberg's algorithm.
  We generated 30 networks of this kind, with changed the value of n along a geometric series from 500 to 25000.

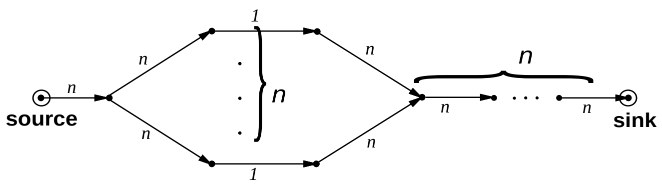

FIGURE 1. Goldberg-Bad-Case network

- **CL - Cheriyan-Graph-Long**. A deterministic family, generated by the WASHINGTON code. The type is "11", and has three parameters n, m, c. The structure is described in Figure 2. This family was designed by J. Cheriyan, and also considered as a hard case for Goldberg's algorithm.
  For CL networks, we generated 30 instances with $n = 1000$, $m = 10$ and c changed along a geometric series from 10 to 1000.
- **CW - Cheriyan-Graph-Wide**. The same family as CL, except the generated 30 instances have parameters $n = 1000$, m changed along a geometric series from 10 to 1000 and $c = 10$.

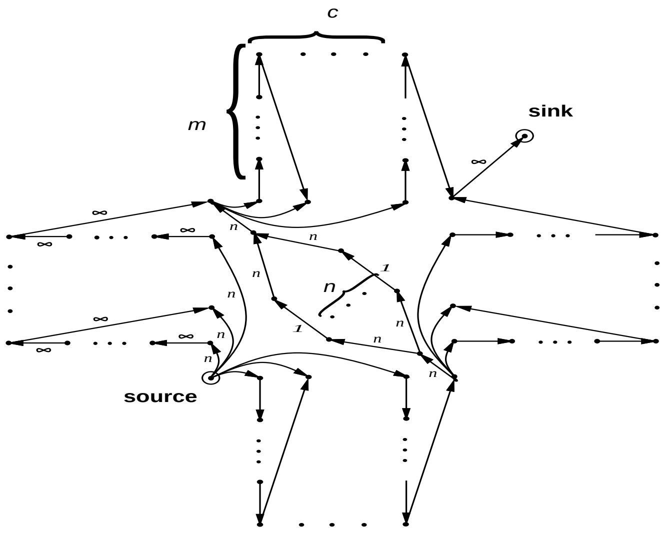

FIGURE 2. Cheriyan-Graph

The following three families were all generated by the code GENRMF, with different parameter settings. Each contains 30 instances. Capacities c_1, c_2 were set to 100 and 10000 respectively in each case.

- **GXL - Genrmf-eXtra-Long.** Extreme "long" networks, with $a = 4$, and b changed along a geometric series from 20 to 2430.
- **GM - Genrmf-Medium.** Cubic like networks, with $a = b$, and both were incremented by 1 from 5 to 34.
- **GXW - Genrmf-eXtra-Wide.** Extreme "wide" networks, with a changed along a geometric series from 20 to 94, and $b = 4$.

TABLE 2. Benchmark times (sec) on the SPARCstation 1+ used for testing.

FORTRAN	**TEST 1**			**TEST 2**		
(netflow)	**real**	**user**	**sys**	**real**	**user**	**sys**
f77	5.7	4.7	0.5	9.8	8.6	0.6
f77 -O	4.8	3.6	0.4	7.3	6.1	0.6
f77 -O1	5.4	4.6	0.5	9.7	8.5	0.6
f77 -O2	4.0	3.4	0.4	6.7	5.7	0.6
f77 -O3	4.4	3.6	0.5	7.0	6.3	0.4
f77 -O4	3.7	2.9	0.5	5.8	5.0	0.6
C	**TEST 1**			**TEST 2**		
(wmatch)	**real**	**user**	**sys**	**real**	**user**	**sys**
cc	5.5	5.2	0.2	46.2	44.7	0.5
cc -O	4.4	4.0	0.2	35.9	34.2	0.6
gcc	5.9	5.5	0.2	49.3	47.5	0.5
gcc -O	4.4	4.1	0.2	34.2	32.7	0.3

4.2. Description of the reported data. In all reported experiments we used the **Gap** switch on , **Relab-freq** $= 0.5M$, **Cut-freq** $= 0$ and the procedure to generate permutations was turned off.

Table 3. contains the mean running times of 10 random instances and the relative variance for the DIMACS Benchmark problems. All reported times are in CPU seconds on a Sun Sparc 1+ Workstation and do not include I/O time. We attempted to measure the user CPU time what the program uses during the execution of the algorithm. Since we used a multi-user workstation under Unix, our obtained running times slightly depended on the workload of the system. This can explain a part of the large variances.

Figures 3–9 contain graphical comparison of the tested algorithms on the GB, CL, CW, GXL, GM, GXW families.

The codes were written in ANSI C language, employing structures, pointers in order to handle the various operations efficiently. Memory requirements are

24 bytes/edge for all codes and 64 bytes/vertex for PLED, 20 bytes/vertex for GOLD, and 40 bytes/vertex for GOLDYN. All integers are of 4 bytes. We used the Gnu C-compiler (gcc) with -O optimization option.

Table 2. contains some benchmark results which were obtained by executing the DIMACS provided Fortran and C codes on our machine.

TABLE 3. Mean running times (secs) and relative variances (%) of the instances.

NAME	PLED		GOLD		GOLDYN	
	mean	var %	mean	var %	mean	var %
GL1	2.11	19.9	0.68	19.6	1.14	22.3
GL2	4.82	11.9	1.57	12.4	2.59	15.3
GL3	11.20	9.9	3.76	6.9	5.71	9.9
GL4	27.89	9.2	8.83	11.3	12.74	5.4
GL5	64.69	9.3	20.94	10.1	28.98	7.3
GL6	275.75	7.0	54.43	10.2	144.85	7.6
GW1	3.38	10.2	1.27	9.0	2.42	11.4
GW2	8.91	7.5	3.45	5.7	6.55	3.9
GW3	20.62	6.4	8.46	4.0	15.47	9.7
GW4	61.92	7.6	23.49	6.1	43.86	6.3
GW5	158.01	4.7	59.84	4.7	106.44	6.6
GW6	979.44	6.5	149.21	8.0	314.10	6.0
WLM1	0.20	5.3	0.07	4.2	0.13	5.0
WLM2	0.53	8.7	0.18	6.2	0.30	7.8
WLM3	1.38	7.5	0.37	11.8	0.54	8.5
WLM4	3.21	10.2	0.87	11.2	1.19	13.7
WLM5	8.05	6.5	2.28	5.8	2.79	9.5
WLM6	22.75	5.3	5.94	9.7	7.14	6.1
AD1	1.32	18.9	1.11	10.5	1.34	18.8
AD2	2.42	19.8	1.80	13.2	2.22	15.1
AD3	3.52	16.1	2.82	12.4	3.38	11.1
AD4	5.78	16.5	4.65	11.3	5.68	11.6
AD5	8.81	16.4	7.09	16.1	8.49	16.9
AD6	14.43	12.6	14.01	8.1	14.27	10.2

4.3. Evaluation of the DIMACS Benchmark problems. From Table 3. at first sight we can say that the simple preflow-push code (GOLD) performs much better on these families than our PLED or GOLDYN implementation. The only exception were the AD instances, where all codes performed about the same way. Moreover the ratio of the running times of PLED over GOLD is getting bigger in almost all the cases as the number of nodes increases. This ratio is very much depending on the structure of the network.

Having a closer look of the profile information of PLED, we found that PLED used about 45% of the execution time for heap operations and only 10-12% for dynamic trees operations. We have also found (see Table 5.) that the number of selections of the vertices for the same problem were about the same in PLED as in GOLD. Since GOLD uses only a simple FIFO policy, while PLED has a "sophisticated" minimum label selection rule, GOLD can be much faster in these cases. This means that using the global relabeling technique, in average both algorithms take approximately the same number of steps.

The running times of PLED in Table 3. corresponds to the "considered to be the best" parameter settings. From early experiments we learned that using random or deterministic permutations instead of just leaving the edgelists untouched has no practical advantage. In some cases the number of operations decreased slightly if the edgelists were permuted, but the cost of such a procedure is high, and the running times became larger. We even encountered examples, in which the number of steps taken was more with using permutations than without them. Let us remark also since all the examples were generated randomly, an additional random permutation of the adjacency lists can not add much to the randomness of these lists.

Our experiments also showed, that perhaps the best setting for the parameter **Relab-freq** (, and **Cut-freq**) is in the range $[0.4N..4N]$ ($[0.1N..N]$), however, we could not make much difference between the values in this range. For several different settings from this range we obtained almost identical running times. On the other hand, of course, too small values for these parameters result in too many big-relabel calls, while too large values do not make profit of the big-relabel procedure. As we mentioned earlier, the effect of big-relabel is so significant, that its examination could be a subject of later studies.

It is interesting that problems with short paths between the source and the sink (eg. the Acyclic-dense problems which are full graphs with diameter 1 or 2) need much less effort to solve than other problems. A reason for this perhaps, is that the maximum flow can be pushed over much sooner in these cases.

TABLE 4. Number of Saturating and Nonsaturating pushes.

NAME	SATURATING			NONSATURATING		
	GOLD	GOLDYN	PLED	GOLD	GOLDYN	PLED
WLC1	895	1036	899	2735	2172	3451
WLC2	1838	2231	1878	7444	5889	10282
WLC3	3550	4392	3775	17720	13869	26400
WLC4	7397	9629	8465	45568	36075	71340
WLC5	14001	18996	17646	104444	81626	182883
WLC6	27666	39759	38472	252319	193867	461418
GL1	1076	1217	1004	10783	8619	12343
GL2	2151	2514	2072	24849	19618	27864
GL3	4229	4890	4065	60679	45268	71501
GL4	8228	9378	8078	146039	104902	201734
GL5	17065	19465	16867	343387	245319	476582
GL6	36856	42414	38370	900646	624328	1364934
GW1	1899	3038	2598	19883	20123	15289
GW2	4858	7826	6759	51479	57841	42577
GW3	10453	17456	15319	124369	141565	95288
GW4	26700	44745	42481	342545	408457	287406
GW5	61309	102140	103667	826702	1037573	765610
GW6	138916	236455	259357	2068616	2699497	2268190
WLM1	247	251	240	652	611	841
WLM2	490	503	459	1264	1179	1972
WLM3	938	978	882	3504	2651	7603
WLM4	1773	1857	1595	6367	4747	17397
WLM5	3621	3822	3149	12077	8594	39231
WLM6	6970	7480	5978	23030	15110	91198
AD1	358	628	586	415	142	258
AD2	499	838	773	537	190	402
AD3	618	1040	936	670	235	443
AD4	759	1323	1186	858	292	534
AD5	954	1631	1483	1052	376	683
AD6	1183	1977	1902	1283	495	884

TABLE 5. Number of Vertex Selections and Relabels.

NAME	SELECT			RELABEL		
	GOLD	GOLDYN	PLED	GOLD	GOLDYN	PLED
WLC1	3374	2470	3217	713	727	736
WLC2	9108	6601	8821	1828	1940	2112
WLC3	21674	15208	21884	4264	4477	5467
WLC4	55836	40074	59103	10953	12069	15545
WLC5	126935	88635	146368	23762	26680	40123
WLC6	307239	210937	365618	57558	63994	102588
GL1	13454	8814	10152	2944	2954	3132
GL2	30803	20046	22795	6610	6844	7181
GL3	73657	43938	53240	14367	14884	16559
GL4	174621	96398	131540	31354	32443	40588
GL5	407952	218523	305694	72237	73552	95535
GL6	1054634	531145	811533	170055	178095	256119
GW1	25255	19371	17671	5773	7095	6685
GW2	65811	51368	45786	15393	19091	17494
GW3	157419	119001	106912	35319	44203	40563
GW4	435163	329006	317527	98618	123425	120234
GW5	1054655	798915	810822	241743	302175	306952
GW6	2615522	1969943	2124932	578402	752522	804951
WLM1	787	582	734	140	142	151
WLM2	1508	1143	1649	245	259	330
WLM3	4165	2511	5194	682	597	1092
WLM4	7506	4503	11088	1195	1035	2301
WLM5	14035	8046	24355	1960	1593	4975
WLM6	26470	14168	52860	3442	2572	10574
AD1	590	567	556	188	187	307
AD2	788	761	864	270	270	470
AD3	974	915	944	326	318	518
AD4	1246	1198	1152	415	418	639
AD5	1513	1465	1471	497	499	815
AD6	1864	1805	1819	576	583	1021

4.4. Testing on the GXL, GM, GXW problems. Since the DIMACS Benchmark problems had only a limited set of networks, we decided to compare our codes on some "extreme" families too. We assumed that the length of paths from the source to the sink could be relevant for Goldberg's algorithm, so we generated the earlier described GXL, GM, GXW network families. Figures 3., 4., and 5. show graphs of running times obtained on these networks. All the scales are linear and the runtimes are shown as functions of the number of vertices.

Among these problems the GXW family seems to be the hardest to solve. Even the growth rate is the highest on GXW networks for all the codes. Again GOLD proved to be the fastest in the GM and GXW case, while PLED is the slowest in all three cases. However, on the GXL family the dynamic tree version of Goldberg's algorithm performed the best. This family has the property that the sink can be reached from the source using only relatively long paths, (compared to the size of the network). So the strategy of pushing excess along a longer path seems to have advantage in this case. GOLDYN pushes excesses as far as possible without controlling the size of this path or the amount to be pushed. On the other hand PLED does not use this "greedy" strategy (e.g. by limiting the amount to be pushed).

This experiment also suggests that the longer the relative distance between the source and the sink, the more useful the GOLDYN algorithm could be.

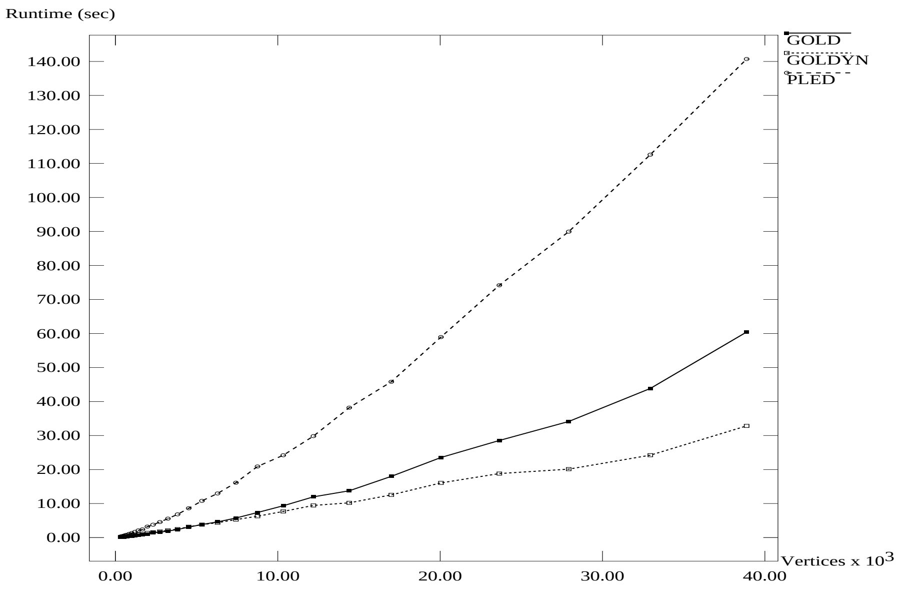

FIGURE 3. Runtimes on the Genrmf-eXtra-Long problems.

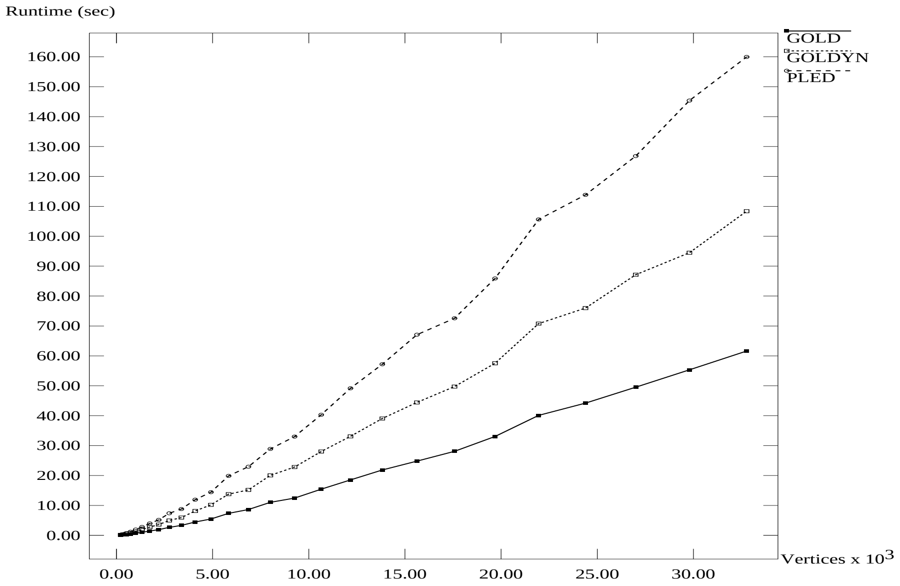

FIGURE 4. Runtimes on the Genrmf-Medium problems.

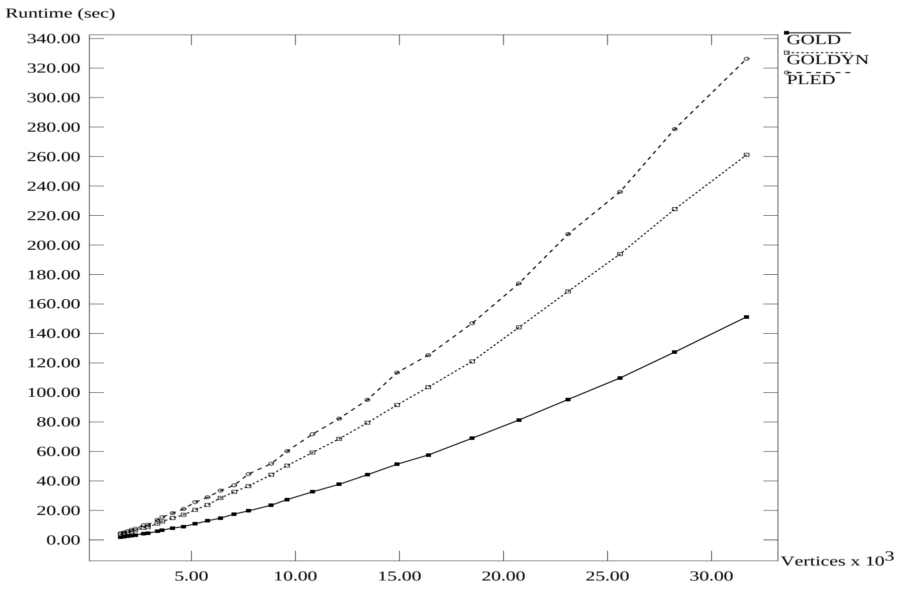

FIGURE 5. Runtimes on the Genrmf-eXtra-Wide problems.

4.5. The effect of dynamic trees. Figures 6., 8. and 9. show the experimental results obtained on the GB, CL and CW families. To examine the effect of using the dynamic trees data structure, we also ran our NPLED and NGOLDYN code on these examples. Figure 7. contains the same runtimes as Figure 6., except the scale is different since it shows PLED and GOLDYN only.

It is clear that GOLD, even with using the big-relabel strategy, is far the slowest on the GB problems. What is more interesting is that this family was the only case where PLED and GOLDYN outperformed their "non-dynamic tree" versions. While the running time of NPLED and NGOLDYN is exponentially increasing, PLED and GOLDYN has a nice linear behavior (see Figure 7.). The difference can be as large as 16 second versus 3192 second for GOLDYN vs. NGOLDYN and 73 second vs. 1862 second for PLED vs. NPLED on the largest example. This emphasizes again that pushes along long paths can be handled by the dynamic trees structure much more efficiently.

We counted the average length of a push during the execution of PLED and GOLDYN, and found that in the case of the GB family it was significantly longer than in any other families. Significant means several hundreds or thousands for GB and less than 10 (ten) for the rest, including all the tested families. On the CL family both NPLED and NGOLDYN were faster than their counterparts, and on the CW family PLED could beat NPLED. Looking at the average push length in these cases we again found that it was very small (under 3) in the CL family for both algorithms, and slightly increasing, but still under 11 on the CW problems for GOLDYN. This length was the longest (≈ 15) on the CW family for PLED, what may explain why it could perform better then NPLED.

It is worth to mention that in all the cases PLED tends to push along longer paths as we have found that the average length of a push was always longer for PLED than for GOLDYN. This may come from the fact that PLED links trees together only if the link has enough residual capacity, while GOLDYN can link trees with a small capacity edge, and hence by pushing along a path GOLDYN may have to cut more often than PLED.

The second longest average length for pushes were found in the GXL case. This supports our conjecture that algorithms using dynamic trees like PLED or GOLDYN can have practical advantage on networks with extremely large distance between the source and the sink.

Figures 8. and 9. also show that the actual structure of the network can change completely the ranking of the execution times of these algorithms.

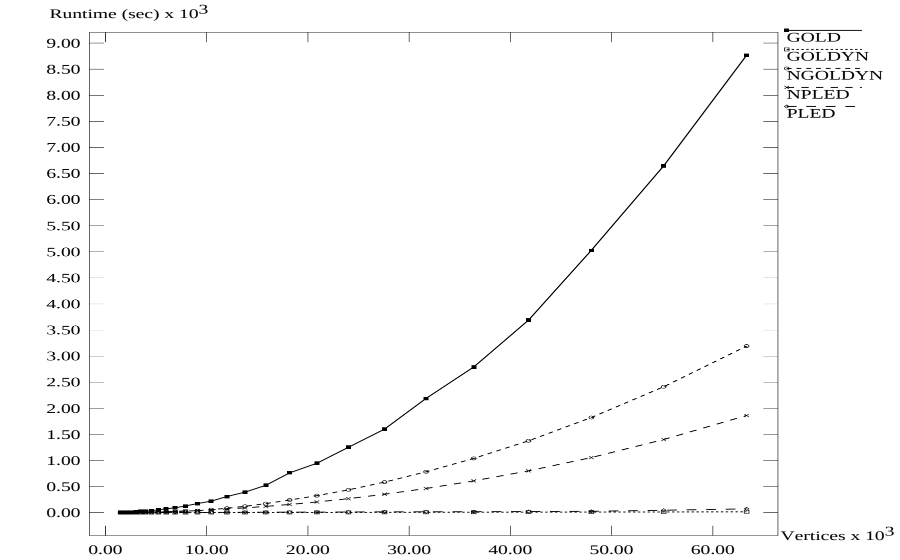

FIGURE 6. Runtimes on the Goldberg-Bad-Case problems.

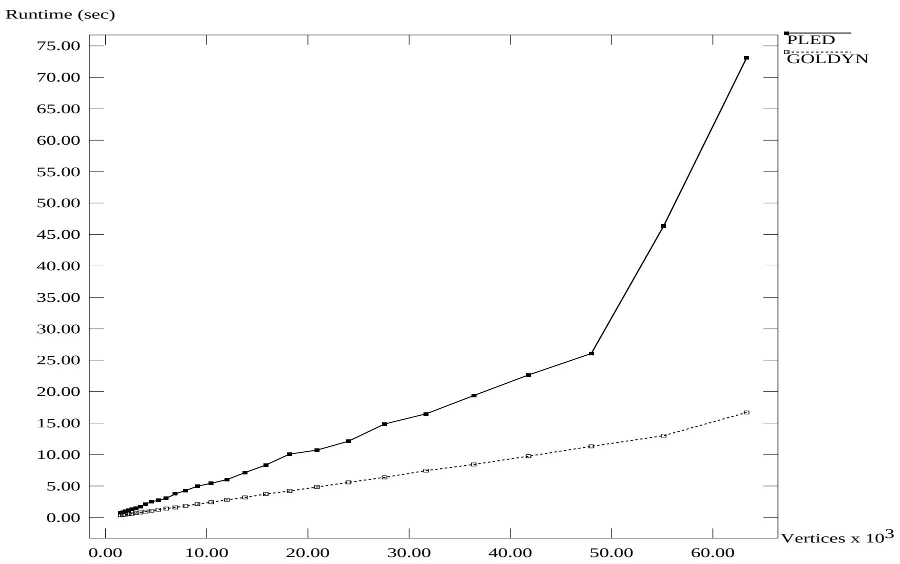

FIGURE 7. Runtimes of PLED and GOLDYN on the Goldberg-Bad-Case problems. (Note that the last to sample points for the time of PLED shows that the workstation started to swap memory at these sizes, since PLED uses the memory most extensively.)

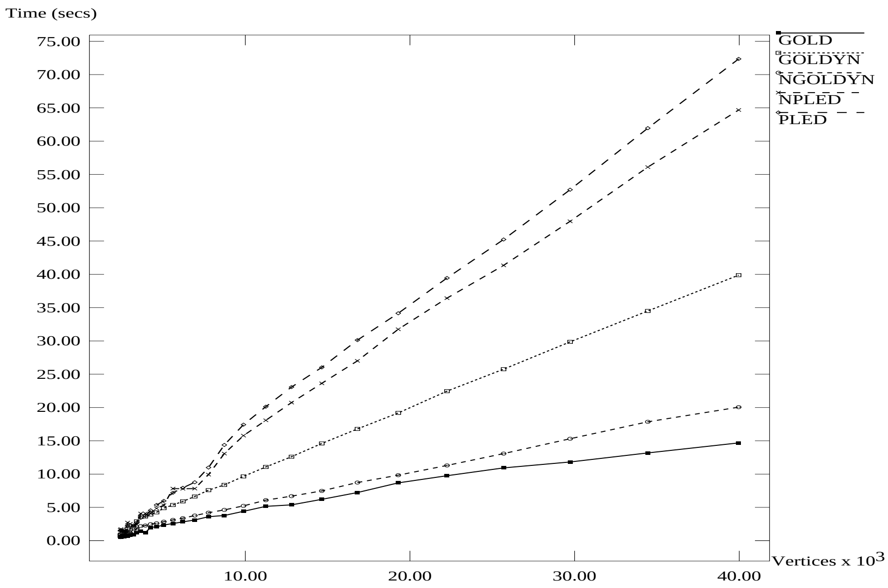

FIGURE 8. Runtimes on the Cheriyan-Long problems.

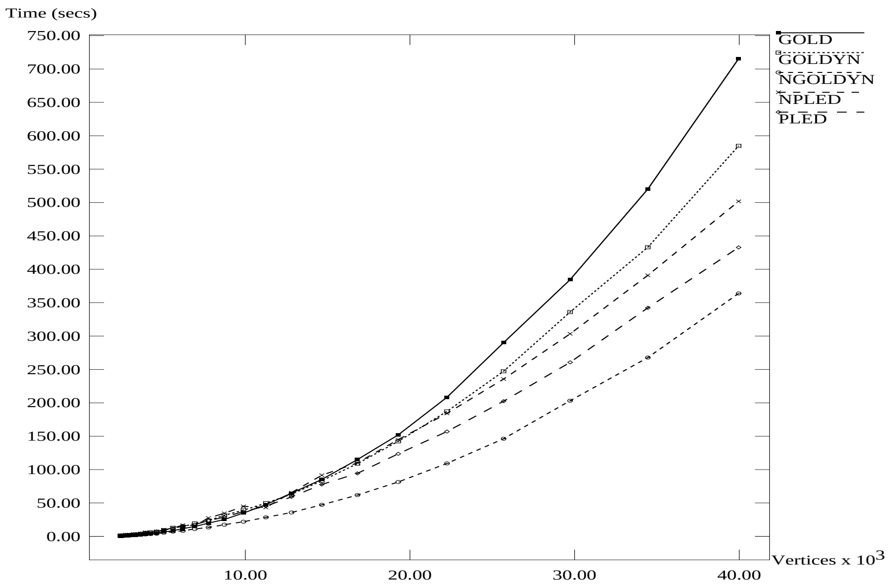

FIGURE 9. Runtimes on the Cheriyan-Wide problems.

5. Conclusions

Summarizing our work, we can conclude that although the PLED algorithm has a very good theoretical worst case bound, in practice Goldberg's simple preflow-push algorithm outperforms it on most of the examples of this study.

Our study shows that the structure of the networks is the most important factor in ranking the algorithms. (E.g. see the examples shown on Figures 8. and 9.) One such parameter to be considered, reflecting the structure of the network, could be the relative distance between the source and the sink.

In this study we were particularly interested in the effectiveness of dynamic trees. Our experiments show clearly that there are families of problems for which dynamic trees improved the performance of our code at a small cost. To determine the properties of network classes on which the algorithms GOLDYN or PLED are the best would be an interesting topic of later works.

Let us remark finally that Fibonacci heaps did not help much in these examples. They did not improve neither the running time nor the number of selection steps of PLED.

Acknowledgment: The authors are thankful to Joseph Cheriyan and to the anonymous referee for the many helpful comments and suggestions.

References

1. J. Cheriyan and T. Hagerup: "A randomized maximum-flow algorithm". in Proc.IEEE FOCS (1989), 118-123.
2. J. Cheriyan and T. Hagerup: "A randomized maximum-flow algorithm". Tech. Report 988, O.R.I.E, Cornell Univ., Ithaca, NY, Oct. 1991.
3. J. Cheriyan, T. Hagerup and K. Mehlhorn: "Can a maximum flow be computed in $o(nm)$ time?". Proc. 17th Internat. Colloquium on Automata, Languages and Programming, 1990, pp.235-248.
4. J. Cheriyan, T. Hagerup and K. Mehlhorn: "An $o(n^3)$-Time Maximum-Flow Algorithm". Tech. Report MPI-I-91-120, Max Planck Institut für Informatik, Saarbrücken, Germany, Nov. 1991.
5. N. Alon: "Generating pseudo-random permutations and maximum flow algorithms" *Information Processing Letters* **35** (1990), 201-204. North-Holland
6. R. K. Ahuja and J. B. Orlin: "A Fast and Simple Algorithm for the Maximum Flow Problem". *Operations Research* **37** (1989), 748-759.
7. R. K. Ahuja, J. B. Orlin and R. E. Tarjan: "Improved Time Bounds for the Maximum Flow Problem". *SIAM J. Computing* **18** (1989), 939-954.
8. R. Anderson, et al.: Program washington.c for creating maximum flow problem instances. Available through Dimacs.
9. T. Badics: Program genrmf.c for creating maximum flow problem instances. Available through Dimacs.
10. G. Waissi: Program ac-max.c for creating maximum flow problem instances. Available through Dimacs.
11. U. Derigs and W. Meier: "Implementing Goldberg's max-flow-algorithm — A computational investigation". *ZOR - Methods and Models of Operations Research* **33** (1989), 383-403.
12. A. V. Goldberg: "A New Max-Flow Algorithm". Tech. Rep. MIT/LCS/TM-291, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1985.
13. A. V. Goldberg and R. E. Tarjan: "A New Approach to the Maximum-Flow Problem". J. ACM 35 (1988), 921-940.

14. A. V. Goldberg, É. Tardos and R. E. Tarjan: *Network Flow Algorithms.* Tech. Rep. No. 860, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY, 1989.
15. D. D. Sleator and R. E. Tarjan: "A Data Structure for Dynamic Trees". *Journal of Computer and System Sciences* **26** (1983), 362-391.
16. R. E. Tarjan: *Data Structures and Network Algorithms.* SIAM publications, Philadelphia, PA, 1983.
17. M. L. Fredman and R. E. Tarjan: "Fibonacci Heaps and Their Uses in Improved Network Optimization Algorithms". *Journal of ACM* **34** (1987), 596-615.
18. M. L. Fredman, R. Sedgewick, D. D. Sleator and R. E. Tarjan: "The Pairing Heap: A New Form of Self-Adjusting Heap". *Algorithmica* **1** (1986), 111-129.
19. Gy. Novotny: "Comparison of Different Data Structures". *Thesis.* Eötvös Loránd University, Budapest, 1989.

RUTCOR, Rutgers University, New Brunswick, NJ 08903-5062.
E-mail address: badics@rutcor.rutgers.edu, boros@rutcor.rutgers.edu