

Pairing Heaps with $O(\log \log n)$ decrease Cost *

Amr Elmasry
Max-Planck Institut für Informatik
Saarbrücken, Germany
elmasry@mpi-inf.mpg.de

Abstract

We give a variation of the pairing heaps for which the time bounds for all the operations match the lower bound proved by Fredman for a family of similar self-adjusting heaps. Namely, our heap structure requires $O(1)$ for insert and find-min, $O(\log n)$ for delete-min, and $O(\log \log n)$ for decrease-key and meld (all the bounds are in the amortized sense except for find-min).

1 Introduction

A self-adjusting data structure is a structure that does not maintain structural information (like size or height) within its nodes, still can adjust itself to perform efficiently and theoretically compete with other structures. Avoiding the restrictions governing other structures and the necessity of maintaining structural information, self-adjusting structures showed practical superiority over their counterparts.

Following splay trees [12], a self-adjusting alternative to balanced trees with many other interesting properties, the next move was a self-adjusting heap. Fredman et al. [6] introduced the pairing heaps as a self-adjusting alternative to Fibonacci heaps. Despite the ingenuity of their structure and proofs, they were able to illustrate an amortized $O(\log n)$ bound for various heap operations. Lagging behind the Fibonacci heaps, the asymptotic time bounds for various heap operations except for delete-min were to be improved.

The pairing heap [6] is a heap-ordered general tree. The values in the heap are stored one value per node. The basic operation on a pairing heap is the linking operation in which two trees are combined by linking the root with the larger key value to the other as its leftmost child. The following operations are defined for the standard implementation of the pairing heaps:

- *find-min*. Return the value at the root of the heap.
- *insert*. Create a single-node tree and link it with the tree of the heap.

- *decrease-key*. Decrease the value of the corresponding node. If this node is not the root, cut its subtree and link the two trees resulting from the cut.
- *delete-min*. Remove the root of the heap and return its value. The resulting trees are then combined to form a single tree. Several variants have been suggested, each with a different way of combining these trees. For the standard two-pass variant, the linkings are performed in two passes. In the first pass, called the pairing pass, the trees are linked in pairs from left to right (pairing these trees from right to left achieves the same amortized bounds). In the second pass, called the right-to-left incremental pass, the resulting trees are linked in order from right to left, where each tree is linked with the tree resulting from the linkings of the trees to its right.
- *meld*. Link the two trees representing the two heaps.

Another variant of the pairing heaps is called the multi-pass variant [6]. In the multi-pass variant, the delete-min operation proceeds as follows. The children of the deleted root are linked in pairs; then, instead of the second right-to-left incremental pass the resulting trees are linked in pairs again, and the process repeats until a single tree remains. The $O(\log n)$ bound does not apply for the multi-pass variant of the pairing heaps. Other variants with different linking strategies for the delete-min operation were also given [1, 6]. Iacono [7] and Elmasry [2] studied some distribution-sensitive properties of the pairing heaps.

Around the same time of introducing the pairing heaps, Sleator and Tarjan [11] also introduced the skew heaps, another self-adjusting heap structure. As first introduced, the skew heaps did not support the decrease-key operation, but can be extended to support it in $O(\log n)$ time. The amortized asymptotic time bounds for skew-heap operations are $O(\log n)$, still lagging behind those of Fibonacci heaps.

*Supported by Alexander von Humboldt Fellowship.

One year after, Stasko and Vitter [13] were able to improve the amortized time bound for insert to $O(1)$ by modifying the standard implementation of the pairing heaps using an auxiliary buffer for insertions. With every insertion, a new single-node tree is added to the buffer. Accompanying a delete-min operation, the nodes in this buffer are first combined into one tree using the multi-pass pairing, this tree is then linked with the main heap.

Despite the inability to prove better bounds for decrease-key, several experiments [8, 9, 13] were conducted illustrating that the pairing heaps are practically superior to other heaps, including the Fibonacci heaps, especially for applications that involve decrease-key operations! Other experiments [3] were also conducted with an attempt to construct a worst-case scenario for the decrease-key operation, but the outcome was that decrease-key takes a constant time in practice!

An important step was taken by Fredman [5], when he showed that a constant amortized cost of decrease-key is precluded by establishing a lower bound of $\Omega(\log \log n)$ for a family of heap structures that generalize the standard implementation of the pairing heaps. He [5] also conducted experiments illustrating that this lower bound is subject to experimental detection by constructing practical scenarios where the pairing heap requires more than constant amortized time to perform decrease-key. Though practically efficient, pairing heaps is therefore not asymptotically as efficient as Fibonacci heaps. Left with skew heaps, one would think about the possibility of improving the bound for decrease-key. This was also precluded by Fredman [4] who introduced a transformation, referred to as depletion, which when applied to skew heaps induces the skew-pairing heaps. This implies that his lower bound, as applying to skew-pairing heaps, also applies to skew heaps.

In the same paper [5], Fredman stated that the cost of m pairing-heap operations, including n delete-min operations, is $O(m \log_{2m/n} n)$. This bound implies a constant cost for the decrease-key operation when $m = \Omega(n^{1+\epsilon})$, for any constant $\epsilon > 0$.

The gap between the upper bound and the lower bound for decrease-key lasted for six years, until Pettie [10] proved an $O(2^{2\sqrt{\log \log n}})$ upper bound, the first sub-logarithmic bound for operations other than delete-min. He also conjectured that Fredman's lower bound is tight.

In this paper, we give a variation of the pairing heaps that accomplishes an $O(\log \log n)$ bound for the decrease-key operation. We describe the data structure in the next section, then we prove the time bounds, and finally we conclude the paper with some remarks.

2 The data structure

There are two basic ideas behind our new implementation for the pairing heaps: The first idea is that, when applying a decrease-key operation on a node x , we not only cut x 's subtree from its parent but also cut the subtree of the leftmost child of x from x 's subtree and glue this leftmost subtree in the place of x 's subtree. The intuition is that cutting subtrees would result in holes implying bad heap structures, while replacing a subtree with the subtree of the leftmost child of its root would hopefully fill the hole with a subtree that is comparable in size. The second idea is to extend the method of Stasko and Vitter [13] of performing lazy insertions, using an auxiliary insertion buffer, to involve the subtrees (without the leftmost subtrees of their roots) cut by decrease-key operations. Our heap structure will have, in addition to the insertion buffer, another pool of trees which contains these cut subtrees. Once the number of trees of this pool reaches some threshold, we apply a sorting subroutine on the roots of these trees and combine them according to such order. The intuition is that since we do not know the sizes of these trees, there is no guarantee on how to combine them in an efficient structure. Linking the trees in order, each as the leftmost child of the other, will result in such a good structure. Because sorting is expensive, we have to involve a bounded number of such trees. The details follow.

After a delete-min operation, we combine all the trees of the heap in one tree that we consider as the main tree of this phase. During the execution, the heap structure is composed of two components: the insertion buffer and the tree pool. The insertion buffer holds single-node trees that have been inserted in the heap since the last delete-min operation. The pool holds at most $\lceil \log n \rceil$ ($n \geq 2$) heap-ordered trees of various sizes including the main tree together with the subtrees that are cut by decrease-key operations as well as the trees that are added by meld operations. We also maintain a pointer to the root of the tree that has the minimum value among those in the insertion buffer and the pool.

The detailed implementation of various heap operations are as follows:

- *find-min*. Return the value of the root pointed to by the minimum pointer.
- *insert*. Create a single-node tree and add it to the insertion buffer. Update the minimum pointer to point to this node if necessary.

We use the following procedure in implementing the upcoming operations:

- *combine-heap*: As in [13], combine the nodes of the insertion buffer into one tree using the multi-pass

pairing. Combine the trees of the pool in one tree by sorting the values of the roots of these trees, and linking the trees in this order such that their roots form a path of nodes in the combined tree (make every root the leftmost child of the root with the next smaller value). Link the combined tree of the insertion buffer and the combined tree of the pool.

- *decrease-key*. Decrease the value of the corresponding node x . Make the minimum pointer point to x if necessary. If x is not a root: Cut x 's subtree and the subtree of the leftmost child of x . Glue the subtree of the leftmost child of x in place of x 's subtree, and add the rest of x 's subtree (excluding the subtree of x 's leftmost child that has just been cut) to the pool as a stand-alone tree. If the number of trees of the pool is now $\lceil \log n \rceil$, call combine-heap.
- *delete-min*. Remove the root pointed to by the minimum pointer. Combine the subtrees of this root in one tree using the standard two-pass implementation of the pairing heaps [6]. Combine the trees of the heap in one tree by calling combine-heap. Update the minimum pointer to point to the root of this combined tree.
- *meld*. Combine the trees of the smaller heap in one tree by calling combine-heap. Add the resulting tree to the pool of the larger heap, and destroy the smaller heap. Update the minimum pointer to point to the root of this tree if necessary. If the number of trees of the pool is now $\lceil \log n \rceil$, call combine-heap.

3 Analysis

For our amortized analysis, we use a combination of the potential function and the accounting methods [14].

3.1 The potential function In [6, 13] the potential function $\Phi = \sum_x \log s(x)$ is used, where $s(x)$ is the size (number of nodes) of the subtree rooted at x in the binary representation of the pairing heaps. Looking at the same thing in a different way, in [1] the potential function $\Phi = \sum_l \log(s_1(l) + s_2(l))$ is used, where $s_1(l)$ and $s_2(l)$ are the sizes (number of nodes) of the linked subtrees at the moment when link l took place.

The drawback of these potential functions is that they do not reflect the efficiency of the links performed. In other words, we want to reflect the fact that when a tree is linked to a smaller tree, this link is efficient with respect to the amount of information gained by the link. To accomplish this, we use the potential function $\Phi = \sum_l \log \frac{s_1(l) + s_2(l)}{s_2(l)}$, where a tree of size $s_2(l)$ was

linked to a tree of size $s_1(l)$ at the moment when link l took place. This potential function was used in [10].

Despite the fact that the potential on a link may reach $\log n$, there is good news concerning our potential function. When two trees of equal sizes are linked, the potential on the formed link is 1. Therefore, the sum of the potential on the links of a binomial tree that has k nodes is $k - 1$. The potential on a link is even smaller than 1 when a tree is linked to a smaller tree. More interesting is that the sum of potentials on any path of links from a node x to a leaf telescopes to at most $\log \Delta$, where Δ is the number of descendants of x . If the path is the left spine of the subtree of x , the sum of potentials telescopes to exactly $\log \Delta$. These properties of the potential function are the key points that motivate our construction and analysis.

3.2 Credits We maintain the following credits in addition to the potential function: For every tree in the insertion buffer, we maintain an $\Theta(1)$ credits, and call these credits the insertion-buffer credits. For every tree in the pool, we maintain an $\Theta(\log \log n)$ credits, and call these credits the pool credits. We also maintain an $\Theta(\log n)$ credits for the whole heap, and call these credits the heap credits.

3.3 The time bounds We will make use of the following proposition:

PROPOSITION 3.1. $\log n \cdot (\log \log 2n - \log \log n) = O(1)$.

Proof.

$$\begin{aligned} \log n \cdot (\log \log 2n - \log \log n) &= \log n \cdot \log \left(\frac{\log n + 1}{\log n} \right) \\ &= \log \left(1 + \frac{1}{\log n} \right)^{\log n} \end{aligned}$$

But $(1 + \frac{1}{\log n})^{\log n} < e$ when $n \geq 2$, where e is the base of the natural logarithm.

Next, we analyze the time bounds for heap operations. Each operation must maintain the credits, the potential function, and pay for the work it has performed.

find-min. No credits or potential function changes are required. The actual work of find-min is $O(1)$. It follows that the worst-case cost of find-min is $O(1)$.

insert. An $\Theta(1)$ credits are given to the inserted node to maintain the insertion-buffer credits. Since the number of trees of the pool is at most $\lceil \log n \rceil$, Proposition 3.1 indicates that an $O(1)$ credits are enough to adjust the pool credits resulting from incrementing the size of

the heap. The $\Theta(\log n)$ heap credits may require an adjustment of $O(1)$. No potential changes are required. The actual work to insert a new node is $O(1)$. It follows that the amortized cost of insert is $O(1)$.

combine-heap. The following lemma shows that the insertion-buffer credits, in addition to being enough to pay for the actual cost of combining the nodes of the insertion buffer using the multi-pass pairing, will also compensate for the resulting change in potential.

LEMMA 3.1. *The extra potential as a result of performing a multi-pass pairing on the nodes of the insertion buffer is $\Theta(b)$, where b is the size of the insertion buffer.*

Proof. The worst-case scenario that maximizes the potential function is when all the small trees are linked to the large trees whenever two trees of different sizes are linked. Performing a multi-pass pairing on the nodes of the insertion buffer, and using this worst-case scenario, the resulting tree will be in the form of a set of binomial trees each linked as the leftmost child of the one larger in size. Other than the links of the leftmost path in this combined tree, the potential on the other links adds up to at most $b - 1$. Since the sum of the potential values on a path telescopes, the sum of these values on the leftmost path of the combined tree is $\log b$. Therefore, the total potential on the links of the combined tree is at most $b + \log b - 1 = \Theta(b)$ (in fact the maximum value is $b + \log b - 2$).

The trees of the pool are combined by sorting the values in their roots and linking them accordingly in order. This will result in a new path of links. Since the sum of the potential values on a path telescopes, the increase in potential will be $O(\log n)$. The actual work done in sorting the $r \leq \lceil \log n \rceil$ trees of the pool is $O(r \cdot \log \log n)$. The increase in potential as a result of linking the combined tree of the insertion buffer and that of the pool is $O(\log n)$. As a consequence, it follows that the amortized cost of the combine-heap operation is $O(r \cdot \log \log n + \log n)$.

decrease-key. Performing a decrease-key on a node x results in decreasing the number of descendants of all the ancestors of x . Accordingly, there should be an increase in potential for the links along the path of the ancestors of x . However, we are still on the safe side. Before the decrease-key operation, consider the path of nodes from the root including all the ancestors of x followed by the nodes on the left spine of the subtree of x . Since we cut the subtree of x and replace it with the subtree of its leftmost child, the nodes of the above path remain the same except for x . Let P be the

sum of the potentials on this path before the decrease-key. Then, $P = \log(\frac{\tau_1}{\tau_2} \cdot \frac{\tau_3}{\tau_4} \dots \Delta)$, where $\tau_i \geq \tau_{i+1}$, Δ is the number of descendants of x and $\Delta < \tau_i$. Let P' be the sum of the potentials on this path after the decrease-key. Then, $P' = \log(\frac{\tau_1 - \delta}{\tau_2 - \delta} \cdot \frac{\tau_3 - \delta}{\tau_4 - \delta} \dots (\Delta - \delta))$, where δ is the number of the descendants of x excluding those of the subtree of the leftmost child of x i.e. $\delta < \Delta$. As stated in the above text, the change in potential for a link on the path of the ancestors of x is $\log(\frac{\tau_i - \delta}{\tau_{i+1} - \delta} / \frac{\tau_i}{\tau_{i+1}})$, which is positive. Still, we can write $P' - P = \log(\frac{\tau_1 - \delta}{\tau_1} \cdot \frac{\tau_3 - \delta}{\tau_2 - \delta} / \frac{\tau_3}{\tau_2} \cdot \frac{\tau_5 - \delta}{\tau_4 - \delta} / \frac{\tau_5}{\tau_4} \dots)$, which is negative, indicating that the total potential of this path decreases. Since the potentials on all the other links either decrease or remain the same, it follows that the total potential on all the links decreases as a result of the cuts accompanying the decrease-key operation.

When a tree is cut and added to the pool, an extra $\Theta(\log \log n)$ credits are needed for such tree to maintain the pool credits. The combine-heap operation will be called only when the number of trees of the pool is $\lceil \log n \rceil$. In such case, the $O(\log n \cdot \log \log n)$ cost of combine-heap will be paid for from the pool credits (each of the $\lceil \log n \rceil$ trees has $\Theta(\log \log n)$ credits). The actual work done by decrease-key, other than the call to combine-heap if performed, is $O(1)$. It follows that the amortized cost of decrease-key is $O(\log \log n)$.

meld. The cost of the combine-heap operation performed on the smaller heap is paid for from the pool and heap credits of this heap before it is destroyed. When the combined tree of the smaller heap is added to the pool of the larger heap, an extra $\Theta(\log \log n)$ credits are needed for such tree to maintain the pool credits. The size of the larger heap now increases, but at most doubles. Using Proposition 3.1, an $O(1)$ credits are needed to adjust the pool credits for this increase in size. The heap credits of the melded heap require an adjustment of $O(1)$ as well. The combine-heap operation for the melded heap will be called only when the number of trees of the pool of the melded heap is $\lceil \log n \rceil$. In such case, the $O(\log n \cdot \log \log n)$ cost of combine-heap will be paid for from the $\Theta(\log n \cdot \log \log n)$ pool credits of such melded heap. The actual work done by meld, other than the calls to combine-heap, is $O(1)$. It follows that the amortized cost of meld is $O(\log \log n)$.

delete-min. We will think about the two-pass pairing as being performed in steps. In the i -th step, the i -th pair from the right among the subtrees of the deleted root are linked, then the resulting tree is linked with the accumulated result of the linkings of all the previous steps. Each step will then be involving three trees and two links. Let a_i be the tree resulting from the linkings

of the previous step, and let A_i be the size of this tree. Let b_i and c_i be the i -th pair from the right among the subtrees of the deleted root to be linked in the i -th step, and let B_i and C_i be their respective sizes. It follows that $A_{i+1} = A_i + B_i + C_i$. During the delete-min, the two links that are cut in the i -th step have a potential value of

$$\log \frac{A_i + B_i}{B_i} + \log \frac{A_i + B_i + C_i}{C_i}.$$

Four cases are possible for the linkings of b_i, c_i and a_i depending on which trees are linked to which. Next, we show that the change in potential for all the cases is at most $2 \log \frac{B_i + C_i}{A_i}$.

Case 1. The tree c_i is linked to b_i , then the resulting tree is linked to a_i :

The potential on the new links is:

$$\log \frac{B_i + C_i}{C_i} + \log \frac{A_i + B_i + C_i}{B_i + C_i} = \log \frac{A_i + B_i + C_i}{C_i}.$$

The difference in potential is:

$$\log \frac{B_i}{A_i + B_i} < 2 \log \frac{B_i + C_i}{A_i}.$$

Case 2. The tree b_i is linked to c_i , then the resulting tree is linked to a_i :

The potential on the new links is:

$$\log \frac{B_i + C_i}{B_i} + \log \frac{A_i + B_i + C_i}{B_i + C_i} = \log \frac{A_i + B_i + C_i}{B_i}.$$

The difference in potential is:

$$\log \frac{C_i}{A_i + B_i} < 2 \log \frac{B_i + C_i}{A_i}.$$

Case 3. The tree c_i is linked to b_i , then a_i is linked to the resulting tree:

The potential on the new links is:

$$\log \frac{B_i + C_i}{C_i} + \log \frac{A_i + B_i + C_i}{A_i}.$$

The difference in potential is:

$$\log \frac{(B_i + C_i) \cdot B_i}{A_i \cdot (A_i + B_i)} < 2 \log \frac{B_i + C_i}{A_i}.$$

Case 4. The tree b_i is linked to c_i , then a_i is linked to the resulting tree:

The potential on the new links is:

$$\log \frac{B_i + C_i}{B_i} + \log \frac{A_i + B_i + C_i}{A_i}.$$

The difference in potential is:

$$\log \frac{(B_i + C_i) \cdot C_i}{A_i \cdot (A_i + B_i)} < 2 \log \frac{B_i + C_i}{A_i}.$$

If $\frac{B_i + C_i}{A_i} \leq \frac{1}{2}$, then the i -th step results in a decrease in potential of more than 2 units. This released potential is used to pay for the work done in this step.

If $\frac{B_i + C_i}{A_i} > \frac{1}{2}$, we call the i -th step a bad step. The total change in potential resulting from all bad steps is at most $2 \sum_i \log \frac{B_i + C_i}{A_i}$, taking the summation over these bad steps only. Since $A_{i'} > B_i + C_i$ when $i' > i$, the sum of the changes in potential for all such bad steps telescopes to an $O(\log n)$. It remains to account for the actual work done in the bad steps. Since $A_{i+1} = A_i + B_i + C_i$, a bad step i results in $A_{i+1} > \frac{3}{2} A_i$. Since $A_3 > (\frac{3}{2})^3$, it follows that after the j -th bad step (this may be any j' -th step where $j' \geq j$), $A_{j'} > (\frac{3}{2})^j$. Then, the number of bad steps $j < \log_{3/2} n = O(\log n)$. Since the actual work done per step is $O(1)$, the actual work done in the bad steps is $O(\log n)$.

The cost of the combine-heap operation is $O(r \cdot \log \log n + \log n)$, r is the number of trees in the pool at the time when the delete-min is performed. If $r \cdot \log \log n > \log n$, this cost is paid for using the $\Theta(r \cdot \log \log n)$ pool credits. Otherwise, the cost of combine-heap is $O(\log n)$, which is paid for by the delete-min operation. Summing up, it follows that the amortized cost of delete-min is $O(\log n)$.

4 Conclusions

We have given a pairing-heap implementation that achieves an $O(\log \log n)$ cost per decrease-key. The new variation performs a sequence of m operations, including n delete-min operations, in $O(m \log \log n + n \log n)$ time. This implies a constant cost per decrease-key when $m = O(n \log n / \log \log n)$. On the other hand, Fredman also proved a constant cost per decrease-key, for the standard implementation of the pairing heaps, when $m = \Omega(n^{1+\epsilon})$. This efficiency of the pairing heaps when the ratio of the decrease-key to delete-min operations is small or large (asymptotically) illustrates the practically efficient performance for the decrease-key operation [3].

We use sorting to implement our heap operations. Beyond the decision-tree model, breaking the $\Omega(n \log n)$ bound for sorting n integers can be done under several circumstances. Applying these sorting methods to our heap structure would break the $O(\log \log n)$ bound for decrease-key. For example, if the number of bits of the integers representing the elements of the heap is a multiple of $\log n$, radix sort runs in linear time. In such case, we would achieve a constant cost per decrease-key.

It is still open whether the two-pass variant of the pairing heaps, as introduced in [6], achieves the same bounds as the variation we introduced in this paper.

We need to point out that the possibility of inventing a self-adjusting heap structure that achieves an asymptotically better bound of $o(\log \log n)$ for the

decrease-key operation, while having the same optimal bounds for other operations, is still in question. Such self-adjusting heap structure should not follow the settings of Fredman's lower bound, though. One of the settings for Fredman's lower bound to apply is that a comparison is followed by a link between the nodes containing the compared values. Using sorting to combine the pool trees breaks such setting.

Acknowledgment Thanks to Seth Pettie and Irit Katriel for several valuable discussions during my visit to Max-Planck Institut für Informatik in Summer 2004.

References

- [1] A. Elmasry. *Parametrized self-adjusting heaps*. Journal of Algorithms 52(2) (2004), pp. 103-119.
- [2] A. Elmasry. *Adaptive properties of pairing heaps*. Technical Report 2001-29, DIMACS (2001).
- [3] A. Elmasry and M. Fredman. *Unpublished experiments*. Rutgers University (1997).
- [4] M. Fredman. *A priority queue transform*. 3rd Workshop on Algorithms Engineering, LNCS 1668 (1999), pp. 243-257.
- [5] M. Fredman. *On the efficiency of pairing heaps and related data structures*. Journal of the ACM 46(4) (1999), pp. 473-501.
- [6] M. Fredman, R. Sedgewick, D. Sleator, and R. Tarjan. *The pairing heap: a new form of self-adjusting heap*. Algorithmica 1(1) (1986), pp. 111-129.
- [7] J. Iacono. *Improved upper bounds for pairing heaps*. Scandinavian Workshop on Algorithms Theory, LNCS 1851 (2000), pp. 32-45.
- [8] D. Jones. *An empirical comparison of priority-queues and event-set implementations*. Communications of the ACM 29(4) (1986), pp. 300-311.
- [9] B. Moret and H. Shapiro. *An empirical assessment of algorithms for constructing a minimum spanning tree*. DIMACS Monographs in Discrete Mathematics and Theoretical Computer Science 15 (1994), pp. 99-117.
- [10] S. Pettie. *Towards a final analysis of pairing heaps*. 46th IEEE Symposium on Foundations of Computer Science (2005), pp. 174-183.
- [11] D. Sleator and R. Tarjan. *Self-adjusting heaps*. SIAM Journal on Computing 15(1) (1986), pp. 52-69.
- [12] D. Sleator and R. Tarjan. *Self-adjusting binary search trees*. Journal of the ACM 32(3) (1985), pp. 652-686.
- [13] J. Stasko and J. Vitter. *Pairing heaps: experiments and analysis*. Communications of the ACM 30(3) (1987), pp. 234-249.
- [14] R. Tarjan. *Amortized computational complexity*. SIAM Journal on Algebraic Discrete Methods 6 (1985), pp. 306-318.