Linear-time In-place Selection in Less than $3n$ Comparisons

Svante Carlsson* and Mikael Sundström*

Abstract. By developing and exploiting new in-place techniques, we show that finding the element with the median value out of n elements stored in an array can be performed in-place in $(2.95 + \epsilon)n$ (for any $\epsilon > 0$) comparisons and in linear time. This is arbitrarily close to the upper bound for the same problem without space-restrictions.

To make the algorithm competitive we also try to minimize the number of element moves performed by the algorithm since this is the other critical operation. This has resulted in a trade-off between the number of comparisons and the number of moves. By minimizing the sum of the critical operations we achieve an algorithm that uses at most $3.75n$ comparisons and $9n$ moves for finding the median in-place. This is, in principle, twice as good as earlier attempts on implicit selection for both of the operations.

1 Introduction

The problem of selecting the element with a given rank in a set of n elements was shown to be solvable in $\Theta(n)$ time by Blum et al. in 1973 [2]. This was a surprise to the research community since it was believed that the general selection problem was as difficult as sorting. They showed that the number of comparisons needed for the problem of median finding is at most $5.43n + o(n)$. This result was improved in 1976 by Schönhage, Paterson, and Pippenger [7] to $3n + o(n)$ comparisons. It was first in 1994 anyone could improve the $3n$ upper bound despite many attempts from researchers all over the world. By modification, of the algorithm by Schönhage et al., Dor and Zwick achieved an upper bound of $2.95n$ comparisons for the problem [5]. There is still, however, a large gap between the upper bound and the lower bound on $2n$ comparisons that is due to Bent and John [1], also an old result from 1985.

In this context, we would like to study how well this can be implemented in-place. This is interesting both from a theoretical and from a practical point of view. A theoretically interesting question is how much of the information gathered by the algorithm can be implicitly stored in the in-place ordering of the elements and how much information needs to be recomputed or stored externally. From a practical point of view we would also like to minimize the extra storage used. One reason is that by limiting the extra storage we will be able to keep a larger part of the data set in faster memory, which will result in a faster

* Division of Computer Science, Luleå University of Technology, S-971 87 LULEÅ, Sweden. E-mail: {Svante.Carlsson, Mikael.Sundstrom}@sm.luth.se

algorithm. Another reason is that in an implicit data structure we are able to take advantage of localities of memory accesses. When we are using pointer representations we can never be sure where elements that are going to be accessed close in time are going to be stored, which will lead to several page faults that will slow the algorithm down.

By implicit (or in-place) selection we mean that we are only allowed to use a constant amount of additional space, apart from the array storing the elements. In this paper, we show that the heavy space restriction only have a marginal effect on the number of comparisons for finding the median and still only using linear time for the algorithm. We can get arbitrarily close to Dor and Zwick's upper bound on $2.95n$ comparisons. The best algorithm, in the sum of the number of comparisons and moves, that we have found takes $3.75n$ comparisons and $9n$ moves, which is about twice as good for both operations as for the previously best result of slightly less than $7n$ comparisons and $19n$ data moves by Lai and Wood [6].

2 Overview of the Algorithm

Our in-place selection algorithm accepts as input an n elements array $A = a_0, a_2, \dots, a_{n-1}$ of elements drawn from a totally ordered universe. We will assume, in the rest of the paper, that the elements in A are distinct. However, by using three way comparisons, our algorithm, with minor modifications, also handles repetitions among the elements in A , at no extra cost.

Based on similar ideas as the selection algorithm by Schönhage, Paterson and Pippenger [7], our algorithm relies heavily upon a *spider factory* used in the mass production of k -spiders. A k -spider is a partial order consisting of a special element c called the center, k elements that are smaller than c , and k elements that are larger than c . We will locate the spider factory in the leftmost part of the input array. The rest of the array will be organized in as many *spider sites* of size $2k + 1$ as possible. A spider site can be either *prepared*, *in use* or *eliminated*. A prepared spider site contains raw material to the spider factory, a spider site in use contains the elements of a k -spider while an eliminated spider site contains elements that no longer are considered as median candidates.

Fig. 1. The state of A : a. after preparation, b. during propagation, c. at the end of propagation, and d. during elimination.

The first phase of our algorithm is the *preparation phase*, in which the spider factory is built and all spider sites are prepared (Fig. 1 a). When a spider site

is prepared, the elements stored in it are refined into the kind of raw material (e.g. pairs) accepted by the factory.

The next phase of the algorithm is the *propagation phase*, in which k -spiders are produced until all the prepared spider sites are used. A k -spider is produced by *extracting* elements from the factory and swapping them with the elements from the leftmost prepared spider site, followed by a *reconstruction* of the factory. By using the center as key, the produced k -spider is then inserted into a double ended priority queue of previously produced k -spiders. Since we, logically, have an *array* of k -spiders, we will use an implicit priority deque called deap [4]. In Fig. 1 b-c, we see how the prepared spider sites are consumed while the deap grows.

In the final phase of our algorithm, the *elimination phase*, we will eliminate elements until only a few are still in consideration as median candidates. Elimination is accomplished by moving the smallest and largest k -spiders to the spider sites at the right end of the deap as they are deleted from the deap. By swapping elements between the spider sites, the right spider site becomes eliminated. The left spider site is then prepared and used in the production of a new k -spider, which is then inserted into the deap. As one k -spider is inserted for every two deleted, the deap shrinks during this phase Fig. 1 d. The $o(n/\lg n)$ elements remaining when elimination can not proceed are sorted with heapsort to find the median.

To make the selection algorithm implicit there are mainly two problems that need to be solved, namely, how to construct an efficient spider factory in-place and how to implement the priority deque of k -spiders while limiting the number of moves.

3 The Implicit Spider Factory

The key to our main result is the implementation of the spider factory. It consists of a finite partial order, called *hyperpair*, with a distinguished element, the *center*, defined recursively by: H_0 is a single element, and H_{i+1} is obtained from two disjoint copies of H_i by comparing their centers and taking the smaller ($i+1 = 1, 4, 6, 8, \dots$) or larger ($i+1 = 2, 3, 5, 7, \dots$) of these as the new center (Fig. 2).

Fig. 2. A H_6 hyperpair with the center shown as a square and the related elements shown as larger dots.

The hyperpair was also used by Schönhage et al. [7], and Dor and Zwick [5]. We use it in a slightly different fashion by relying on two basic operations, *pair* and *unpair*, to build a H_{2^h} , $2^{h-1} < k \leq 2^h - 1$, and then produce k -spiders by extracting them from the H_{2^h} while simultaneously reconstructing the hyperpair.

The basic idea behind our representation of hyperpairs is to store the elements of a H_i in 2^i consecutive locations of A , with one H_{i-1} sub-hyperpair stored, recursively, in the 2^{i-1} first locations, and the second in the last 2^{i-1} locations. By using a *bit array* $B = b_1, b_2, \dots$, we record the relation between the centers of the sub-hyperpairs in the bit with the same index as the first element of the second sub-hyperpair.

The problem with the above *basic representation* is that the center of a H_i can not be found in constant time. By examining one bit we can decide in which half of the hyperpair the center resides, and by repeating this i times we will find the center. However, if we, after comparing the centers of the H_{i-1} s, store the one that is taken as the center of the H_i being under construction in the first location (by swapping the centers of the H_{i-1} s, if necessary), we will be able to find the center of the H_i without examining any bits. We are also able to restore the sub-hyperpairs since we have recorded the outcome of the comparison, and, hence, also if we have performed a swap or not. This is called the *swap representation*.

In the swap representation, a pair (i, j) is sufficient to identify a H_i hyperpair stored at position j (the position of the center is j). Also, it is necessary that the two H_i s used to build a H_{i+1} hyperpair are adjacent at positions j_1 and j_2 satisfying $|j_1 - j_2| = 2^i$ and $\min(j_1, j_2) \equiv 0 \pmod{2^{i+1}}$. We define the basic operations as follows:

- *pair* $(i + 1, j)$ builds $(i + 1, j)$ by comparing the centers a_j and a_{j+2^i} of (i, j) and $(i, j + 2^i)$ respectively. If a_j is taken as the new center, we flip b_{j+2^i} to record this. Otherwise, a_j and a_{j+2^i} are swapped.
- *unpair* $(i + 1, j)$ reverses the effect of a previous call *pair* $(i + 1, j)$. If $b_{j+2^i} = 0$, it is flipped and $\langle j, j + 2^i \rangle$ is returned. Otherwise, a_j and a_{j+2^i} are swapped and $\langle j + 2^i, j \rangle$ is returned.

Since our factory is a hyperpair with 2^{2^h} elements we will need a bit array of size $2^{2^h} - 1$ for B . We will use a pair of adjacent elements in A to emulate a bit, by placing them in ascending or descending order. This standard technique for encoding bits in-place results in one comparison for each bit examined, and one swap for each bit flip. We extend our *factory space* to $3 \cdot 2^{2^h}$ locations in A to contain also the bit array representation.

Lemma 1. *Performing pairing in the swap representation requires one element comparison and two moves, while unpairing requires one bit comparison and two moves. The factory is built in $2(2^{2^h} - 1)$ comparisons and $4(2^{2^h} - 1)$ moves.*

4 k -spider Production

In this section we describe how a k -spider is produced and also present an analysis, which amounts in the following lemma.

Lemma 2. For any non negative integer, x , there exists a spider factory of size $\Theta(k^2)$, in A , that, given pairs as raw material, produces a k -spider in $(4 + 6 \cdot 2^{-x})k + o(k)$ comparisons and $(2^{x+1} + 2 + 24 \cdot 2^{-x})k + o(k)$ moves.

We begin by describing how a k -spider is produced using a simple factory that accepts singletons as raw material.

Production is performed by first *decomposing* the H_{2h} constituting the factory into its center c and a disjoint set of hyperpairs $H_0, H_1, \dots, H_{2h-1}$, followed by swapping c with the element at the first location of the spider site. The factory is decomposed in $\Theta(h)$ time by repeatedly calling *unpair*.

To obtain k elements that are smaller than c in the k -spider, we will *downward process* the hyperpairs $H_1, H_2, H_4, \dots, H_{2(h-1)}$, which all have centers smaller than c . Downward processing a H_i at position j is accomplished by calling *d_process*(i, j),

$$d_process(0, j) = \text{swap } a_j \text{ with an element from the spider site} \quad (1.1)$$

$$d_process(i, j) = \langle j', j'' \rangle \leftarrow \text{unpair}(i, j) \quad (2.1)$$

$$d_process(i-1, j') \quad (2.2)$$

$$\text{if } i = 2, 3, 5, 7, \dots \text{ then } d_process(i-1, j'') \quad (2.3)$$

$$\text{pair}(i, j) \quad (2.4)$$

which recursively swaps its center and the elements (guaranteed to be) smaller than its center with raw material from the spider site. As the recursion rolls up, a new H_i is built at the same position. To complete the k -spider we extract k elements larger than c by *upward processing*. This is accomplished by applying a procedure similar to *d_process* to the hyperpairs $H_0, H_3, H_5, \dots, H_{2h-1}$, which have centers larger than c .

Lemma 3. By downward processing, 2^i elements that are smaller than c can be extracted from the $H_{2i}, i > 0$, hyperpair, and by upward processing, 2^i elements that are larger than c can be extracted from the $H_{2i+1}, i > 0$, hyperpair.

Proof. Let $f_e(i)$ be the number of elements extracted by downward processing a H_i hyperpair. In (1.1), the singleton which is smaller than c is simply swapped with an element (raw material) at an appropriate location in the spider site. A hyperpair (i, j) , $i > 0$, with center $c' < c$, is split in (2.1). The restored sub-hyperpair $(i-1, j')$, which, by the definition of *unpair*, has c' as center, is then recursively processed in (2.2) obtaining $f_e(i-1)$ elements. If the condition in (2.3) is satisfied, the center of $(i-1, j'')$ is smaller than c' , by the definition of *pair*, and, hence, $(i-1, j'')$ can also be recursively processed to extract another $f_e(i)$ elements. A new (i, j) hyperpair is built in (2.4). Solving the recurrence $f_e(2) = 2, f_e(2i) = 2f_e(2(i-1))$, yields the lemma for downward processing. The proof for upward processing is similar. $\square$

From Lemma 3, it follows that k smaller and k larger elements can be extracted, provided $k \leq 2^h - 1$. We show in [3] how our way of representing hyperpairs makes it possible to implement the processing procedures iteratively with constant extra storage.

The k -spider now completed but the hyperpairs $H_0, H_0, H_1, \dots, H_{2h-1}$ must be assembled into a H_{2h} before the next k -spider can be produced. This is achieved in $\Theta(h)$ time, by repeatedly calling *pair*.

In the remainder of this section we will investigate the processing cost while extracting elements as pairs instead of singletons and by using pairs as raw material. The raw material in a prepared spider site will be organized in k pairs of adjacent elements stored in sorted order. That implicit representation of H_1 s can be used also in the spider factory to avoid bit comparisons at the lowest level in the recursion. We can, as shown in the following lemma, store even larger hyperpairs in an implicit representation of a hyperpair of that size to reduce the number of bit comparisons arbitrarily close to 0.

Lemma 4. *If we represent H_{2x} s implicit in the spider factory we get a processing cost of $2 + 3 \cdot 2^{-x}$ comparisons and $2^x + 1 + 12 \cdot 2^{-x}$ moves per extracted element.*

Proof. The number of element comparisons performed in the downward processing of a H_{2i} and the upward processing of a H_{2i+1} are given by the recurrence $f_c(i) = 2 + 2f_c(i-1)$, since *pair* is called once at each level in the recursion and the recursion branches every second level. We now consider the downward and upward processing of a H_2 , $\alpha < \beta < \gamma$, $\alpha < \delta$. In the former case, two singletons γ and δ are left after the pair $\alpha < \beta$ has been extracted, while, in the latter case, the pair $\beta < \gamma$ is extracted, leaving the pair $\alpha < \delta$. It follows that the cost for building a new H_2 from the elements left and a new pair from the spider site is 2 element comparisons for downward processing and 1 element comparison for upward processing. Also accounting for one element comparison used to pair two H_2 s, during the upward processing of a H_3 , yields the base case $f_c(1) = 2$ for both kinds of processing.

We get a slightly higher cost for upward processing a H_{2x+1} than for downward processing a H_{2x} (2^x elements are obtained in both cases) since the H_{2x+1} must be unpaired before the H_{2x} can be processed. It follows that the cost for upward processing is slightly higher than the cost for downward processing.

Let $f_b(i)$ be the number of bit comparisons and $f_m(i)$ be the number of moves required to upward process a H_{2i+1} . For comparisons, the base case is $f_b(x) = 1$, since one call to *unpair* is made when a H_{2x+1} is processed. For moves, we first consider the processing of a H_{2x} . Exactly 2^x elements are extracted. These are swapped with 2^x elements from the spider site. Since the H_{2x} is stored in an implicit representation, the reconstruction of the H_{2x} , from the elements left and the new elements from the spider site, requires, in the worst case, a complete reorganization of the $2^{2x} - 2^x$ elements of the H_{2x} that are left in the factory. It follows that the number of moves to process a H_{2x} is bounded by $2^{2x} + 2^x$. Also accounting for the calls to *unpair* and *pair*, performed in the processing of the H_{2x+1} , yields the base case $f_m(x) = 4 + 2^{2x} + 2^x$.

At each level in the recursion *unpair* and *pair* are called once, and, since the recursion branches ever second level, we get the recurrences $f_b(i) = 2 + 2f_b(i-1)$ and $f_m(i) = 8 + 2f_m(i-1)$. It is easily shown that $f_c(i) = 2 \cdot 2^i$, $f_b(i) \leq 2^i \cdot 3 \cdot 2^{-x}$ and that $f_m(i) \leq 2^i (2^x + 1 + 12 \cdot 2^{-x})$. Since 2^i elements are extracted we get a cost of $2 + 3 \cdot 2^i$ comparisons and $2^x + 1 + 12 \cdot 2^{-x}$ moves per element. $\square$

We may choose x as an arbitrarily large value, and still only have a linear number of moves for extraction of k elements. Since $2k$ elements are extracted, and the decomposition and composition of the factory takes $\Theta(h)$ time, Lemma 2 follows from Lemma 4.

5 The Priority Deque and the Spider Sites

As described in Section 2 the centers of the k -spiders are inserted into a priority deque after the k -spider have been produced. Any implicit priority deque that supports insertion, extract-min and extract-max in $O(\lg t)$, where t is the number of elements in the deque, can be used. We have chosen the deap as our priority deque [4].

A question that arises, and must be answered, at this point is exactly what to insert in the deap. We can not afford to insert the whole k -spider since it would impose a severe overhead in the number of moves performed. On the other hand, if the center alone is inserted it would be difficult to remember the position of the remaining $2k$ elements without external pointers. Our approach is something in between these two extremes.

We store a k -spider in two parts called the *head* and the *tail*, where the head consist of the center and $\lceil \lg n \rceil$ pairs of the k -spider while the remaining $k - \lceil \lg n \rceil$ pairs constitutes the tail. By using the same technique for encoding bits, as used in the spider factory, we will encode the binary representation of the first location of the tail among the $\lceil \lg n \rceil$ pairs of the head. In this way, the head is provided with enough intelligence to find its own tail.

Now, consider the preparation of spider sites in the *preparation phase*. As mentioned in Section 2, we prepare all spider sites. A spider site is prepared by building k pairs, at a cost of k comparisons and $2k$ moves. Since the number of spider sites prepared is bounded by $n/(2k+1)$, and the spider factory is built in $\Theta(2^{2h})$ time, it follows that $0.5n + \Theta(2^{2h})$ comparisons and $n + \Theta(2^h)$ moves are performed during the preparation phase.

Now we turn our attention to the *propagation phase*. When a new k -spider is to be produced, the leftmost prepared head and prepared tail, are designated as spider site. When the k -spider is completed, a pointer to the tail is encoded in the head, followed by an insertion of the head in the deap.

Finally, we give a more detailed description of the behaviour of our algorithm during the *elimination phase*. In this phase, k -spiders are alternately produced and destroyed. In fact, one k -spider is produced for each pair of k -spiders destroyed.

It can be shown that the smallest center and the largest center, in the deap, are too small and too large, respectively, to be the median if the number of elements left in the algorithm is larger than $7k^3$. It follows that the k elements smaller than the smallest center and the k elements that are larger than the largest center are also too small or too large to be the median. These $2k+1$ elements can, therefore, be eliminated. Elimination is accomplished by swapping the pairs from the smallest k -spider that are smaller than the smallest center,

with the pairs from the largest k -spider that are smaller than the largest center. In this way, the site of the larger k -spider becomes eliminated, while the site of the smaller k -spider becomes prepared. The prepared spider site is then used in the production of a new k -spider. The cost for insertion, deletion and elimination is $\Theta(\lg n)$ comparisons and $k + \Theta(\lg^2 n)$ moves per k -spider.

Choosing $k = 2^h - 2$, where $h = \lceil 0.25 \lg n \rceil$, yields a negligible cost for sorting the $\Theta(k^3)$ elements left (when elimination can not proceed). The costs for inserting, deleting and eliminating a k -spider also becomes negligible except for k moves during elimination.

6 Improving the Factory with Grafting

Schönhage et al. obtained a major improvement in their algorithm by a process called *grafting*. In this section we will incorporate this process in our algorithm.

Grafting is performed prior to processing, and the idea is to repeatedly compare elements with the center until either k elements smaller than the center or k elements larger than the center are found. The grafted elements are obtained at a low cost compared to the, at most, k elements that have to be extracted from the factory to complete the k -spider.

As Schönhage et al., we will use pairs for grafting, and, in order to reduce the number of moves, performed by the grafting process, we will use the pairs that already resides in the prepared spider site currently in use. Grafting the pair $v < w$, to the center c , yields one of three outcomes. We either have (i) $v < w < c$, (ii) $c < v < w$, or (iii) $v < c < w$. Three counters, p_0, p_1 , and s , are used to record the number of occurrences of outcomes (i), (ii), and (iii), respectively. Prior to the grafting, the k pair locations, in the spider site, are occupied by k pairs (raw material). We dedicate $\frac{k}{2}$ of these pairs and pair locations as *smaller pair candidates* and *smaller pair locations*, respectively. The remaining pairs and locations are referred to as *larger pair candidates* and *larger pair locations*, respectively. The grafting proceeds according to the following description until $s + 2 \max(p_0, p_1) \geq k - 1$. If $p_0 > p_1$, let $v < w$ be the $p_0 + 1$ st smaller pair candidate and compare w to c , and if $w > c$, compare v , also, to c . Otherwise, let $v < w$ be the $p_1 + 1$ st larger pair candidate and compare v to c , and then compare w and c , also, if v turns out to be smaller than c . If outcome (i) occurs when $p_0 \leq p_1$ or outcome (ii) occurs when $p_0 < p_1$, the $p_0 + 1$ st smaller pair candidate and the $p_1 + 1$ st larger pair candidate are swapped. When outcome (iii) occurs, the grafted pair is swapped with the $k - \frac{s}{2}$ th smaller pair candidate if s is even, or swapped with the $k - \lfloor \frac{s}{2} \rfloor$ th larger pair candidate if s is odd.

When the grafting terminates, the pairs from outcome (i) and (ii) occupies the first p_0 th smaller pair locations and the first p_1 th larger pair locations, respectively, whereas the singletons resulting from outcome (iii) occupies the last $\lceil \frac{s}{2} \rceil$ smaller pair locations and the $\lfloor \frac{s}{2} \rfloor$ last larger pair locations. The pairs that are not related to the center constitutes the raw material that is used in the completion of the k -spider by processing hyperpairs in the factory. To enable recycling of pairs and singletons, the binary representation of s is encoded in

the k -spider itself, by using $\lceil \lg k \rceil$ pairs of elements. For an extended description please refer to [3].

Lemma 5. *Selection in-place can be performed in $(3 + 3 \cdot 2^{-x}) \cdot n + o(n)$ comparisons.*

Proof. Every comparison performed by the grafting process yields at least one element related to the center of the k -spider. The cost for increasing $\max(p_0, p_1)$ by one during the grafting process is $\max(p_0, p_1) - \min(p_0, p_1) + O(1)$ comparisons. Hence, the cost for obtaining $2s + 2p_0 + 2p_1$ elements by grafting is $2 \min(p_0, p_1) + p_0 + p_1 + 2s + O(1)$ comparisons. For every *two* occurrences of outcome (iii) we must charge one extra comparison for building a pair when these singletons are recycled, yielding an additional $0.5s$ comparisons. By Lemma 4, the $2k - 2s - 2p_0 - 2p_1$ elements, needed to complete the k -spider, are extracted from the factory at a unit cost of $2 + 3 \cdot 2^{-x}$ comparisons. Since $2 + 3 \cdot 2^{-x} \geq 2$, $s + 2 \max(p_0, p_1) \geq k - 1$, and the number of k -spiders produced is bounded by $\frac{n}{k}$, we get a total production cost of $(2.5 + 3 \cdot 2^{-x})n + o(n)$ comparisons for the algorithm. Adding the $\frac{n}{2} + o(n)$ comparisons performed during the preparation phase yields the lemma. $\square$

Obtaining a bound of $(2^x + 3 + 12 \cdot 2^{-x}) \cdot n + o(n)$ moves for the described algorithm is straight forward. The recycling cost is $1.5s$ moves and the proof is essentially the same as for comparisons. However, to obtain our best upper bound of $(2^x + 2 + 12 \cdot 2^{-x}) \cdot n + o(n)$ moves, while keeping the same bound for comparisons, we have used much more sophisticated techniques of moving elements. The description and analysis of these techniques takes about five pages and are, therefore, left out due to space restrictions [3].

By inserting $x = 2$ in the expressions above we get the best sum of comparisons and moves.

Theorem 6. *In-place selection can be performed in $3.75n + o(n)$ comparisons and $9n + o(n)$ moves.*

This result improves the previously best bounds for implicit selection of $6.7756n + o(n)$ comparisons and $18.6873n + o(n)$ moves by Lai and Wood [6].

7 Getting Below $3n$ Comparisons

Recently Dor and Zwick [5] improved the upper bound for selection to $2.95n + o(n)$ comparisons with an algorithm that is based on Schönhage et al. but with some new features. We will indicate how these new features can be incorporated in our in-place selection algorithm.

The algorithm uses sub-factories that accepts different kinds of partial orders as raw material. Partial orders of the same kind can be stored in sub-arrays called *stockpiles* in the in-place version. Each stockpile can be moved in time proportional to the distance by moving elements from one end of the stockpile to the other end. This can be achieved since the order between the partial orders

in a stockpile is irrelevant, and since we can allow one partial order to be stored with one part in each end of the stockpile. If space, in some direction, must be made available for a stockpile (to move or to grow) the next stockpile in that direction is moved. This may, of course, result in that all stockpiles are moved, but can still be performed in constant time since both the number of stockpiles and the size of stored partial orders are bounded by a constant.

Another feature is that the spiders produced does not necessarily have k elements on each side of the center. This results from the use of a more powerful grafting, which yields many outcomes, and that, for each outcome, an optimal number of elements are extracted from the factory. The different kinds of outcomes from the grafting are managed by using stockpiles also in the spider sites. When a k -spider has been completed, the size of each stockpile is recorded in the k -spider using the same technique as described in Section 5.

Since the new features of Dor and Zwick's algorithm can be incorporated with our algorithm and since we can reduce the number of bit comparisons to an arbitrarily small value, by Lemma 4, our main theorem follows.

Theorem 7. *There exists an linear time in-place selection algorithm that finds the median in $(2.95 + \epsilon) \cdot n + o(n)$ comparisons and $\Theta(n)$ moves.*

8 Acknowledgement

We would like to thank Dr. Jingsen Chen for valuable discussions on the problem and for constructive comments on the presentation of these results.

References

1. S. W. Bent, J. W. John, Finding the median requires $2n$ comparisons, In *Proceedings of the 17th Annual Symposium on Theory of Computing*, pp. 213–216, 1985.
2. M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, Time bounds for selection, *Journal of Computer and System Sciences*, 7:448–461, 1973.
3. S. Carlsson and M. Sundström, Linear-time In-place Selection in Less than $3n$ Comparisons, extended version available at [URL:http://www.sm.luth.se/~msm/reports/in-place.ps](http://www.sm.luth.se/~msm/reports/in-place.ps)
4. S. Carlsson, The Deap – A double-ended heap to implement double-ended priority queues, *Information processing Letters* 26 (1):33–36, 1987.
5. D. Dor and U. Zwick, Selecting the median, In *Proceedings of the 6th ACM-SIAM Symposium on Discrete Algorithms (SODA'95)*, San Francisco, California, January 1995.
6. T. W. Lai and D. Wood, Implicit Selection, In *SWAT 88*, pp. 14–23, 1988.
7. A. Schönhage, M. Paterson, and N. Pippenger, Finding the median, *Journal of Computer and System Sciences*, 13:184–199, 1976.