A Heuristic to Accelerate In-situ Permutation Algorithms

Jörg Keller[*]
FernUniversität Hagen, Germany

Abstract

In-situ permutation algorithms permute an array of n items without using a second array. Little space is needed when one proceeds permuting along a cycle. Thus, those algorithms first search for one element called the leader on each cycle of the permutation. Then they move the items cycle by cycle. A great fraction of the runtime is spent with testing whether elements are leaders. We present a simple-to-implement heuristic to accelerate the search for leaders and present performance gains for randomly chosen permutations on three algorithms.

Keywords: algorithms, performance evaluation, heuristics, permutations

1 Introduction

We consider the task of permuting an array of n items according to a permutation π which is given as an oracle. This means, when we specify x, we get back $\pi(x)$ from the oracle, but have no further knowledge about the permutation's structure. This task is trivial when we are allowed to use a second array, a case that is rather seldom for large n. Instead, we consider *in-situ* permutation of an array, that is, permutation without a second array. This problem has first been investigated by Knuth [2] and has been further studied by Fich et al. [1].

All known algorithms that solve this problem are *cycle leader* algorithms, that is, they try to find, for each cycle of the permutation, a unique element called the leader. Normally, the leader is the smallest element on the cycle. Then the algorithms permute each cycle separately, starting

[*]Address: FernUniversität, LG Technische Informatik II, 58084 Hagen, Germany. Phone/Fax +49–2331–987–376/308, Email `joerg.keller@fernuni-hagen.de`

```
void *array[n]; /* the array to be permuted */
extern int pi(int x); /* permutation as oracle */
extern int IsLeader(int i);

void PermuteCycle(int i)
{ int j=i; /* loop variable */

  do{ j=pi(j);  /* follow cycle */
      exchange(i,j); /* exchange a[i] and a[j] */
  } while (j!=i);
}

void main(void)
{ int i; /* loop variable */

  for(i=0;i<n;i++) if(IsLeader(i)) PermuteCycle(i);
}
```

Figure 1: The structure of cycle leader permutation algorithms

from the leader. The structure of the algorithms is given in Figure 1. The leaders are found by checking with a procedure `IsLeader()` all elements from 0 to $n-1$ whether they are leaders.

As the complete array must be permuted, the runtime of all calls to `PermuteCycle` adds up to $\Theta(n)$. However, all known algorithms needing only polylogarithmic space besides storing the array need time $\Omega(n \log n)$ [1], even in the average case. Hence, the majority of time is spent in the calls to `IsLeader`.

In Section 2, we sketch how the `IsLeader` check typically works and present a simple-to-implement heuristic to speed up this search. In Section 3, we present experimental results for randomly chosen permutations on three algorithms. The results indicate that the heuristic accelerates three algorithms considered in [1] by a constant factor which is between 30 and 50 percent on the average. In Section 4, we conclude.

```
extern int pi(int x); /* permutation as oracle */

int TestHypo(int j,int i)
{ return i<=j; }

int IsLeader(int i)
{ int j=i; /* loop variable */
  int flag=1; /* assume i is leader */

  do{ j=pi(j); /* follow cycle */
      if(flag=TestHypo(j,i)) break;
               /* until assumption turns out false */
  } while (j!=i); /* or cycle is completely traversed */
  return flag;
}
```

Figure 2: The structure of cycle leader checks in the algorithms used

2 Checking for Leaders

A check whether i is a leader normally proceeds along the cycle starting in i, testing i against $j = \pi^k(i)$ for $k = 1, 2, \ldots$ with procedure `TestHypo()`, until i is reached again or until the assumption that i is a leader turns out to be false, see Figure 2. The algorithms known from literature differ in their implementations of `TestHypo(j,i)`, which give different runtime and space requirements.

The simplest test is of the form $j \geq i$. If j is smaller than i, then i cannot be the smallest element on the cycle, i.e. i is not a leader. This test leads to a worst case runtime of $O(n^2)$, and to an average runtime of $O(n \log n)$, if each permutation is equally likely. This algorithm, which we will call A1, was presented and analyzed by Knuth [2], but is also mentioned by Fich et. al. [1] as a motivation for the next algorithm.

The algorithm presented by Fich et. al. in Figure 4, called A2, uses $b \leq n$ bits to store information on whether certain elements have already been visited. It proceeds in $\lceil n/b \rceil$ rounds. In round k, where $0 \leq k < \lceil n/b \rceil$, the bits store visit information on elements $k \cdot b, \ldots, (k+1) \cdot b - 1$. An element outside this range always counts as unvisited. The test in procedure `TestHypo()` is

$$j \geq i \text{ and } j \text{ is unvisited}$$

Algorithm A2 has a worst case runtime of $O(n^2/b)$; the average runtime remains $O(n \log n)$ for $b \leq n/\log n$.

The new algorithm by Fich et. al. [1, Fig. 7], called A3, constructs a hierarchy of local minima while proceeding along the cycle until it finds that either i is the global minimum, that is, the smallest element of the cycle, or that i is not a leader. It has a worst case runtime of $O(n \log n)$.

None of the algorithms take into account the following observation. After a cycle has been permuted, the length of the cycle is known. As soon as the lengths of the cycles permuted so far sum up to n, the array is completely permuted and the program can terminate. This leads to a modified algorithm given in Figure 3. We call this heuristic Fastbreak1.

The simple observation above can be extended to function `IsLeader`. If only r elements have not been visited yet by function `PermuteCycle`, then proceeding along a cycle is only useful for r steps. Otherwise, the cycle currently traversed must contain an element that has already been visited by function `PermuteCycle`, the cycle hence has already been permuted, and i cannot be a leader. The modified function `IsLeader` is shown in Figure 4. We call the extended heuristic Fastbreak2.

The heuristic presented will surely not change the asymptotic behaviour of the permutation algorithm used. Yet, there is a rationale behind it. If we assume that the permutation π is randomly chosen, then the expected length of the longest cycle (which will very likely be found first) is about $0.63 \cdot n$. Also, the expected number of cycles is small, i.e. $\ln n + O(1)$. For these and more results on random permutations, see [3, Chap. 6]. As the cycle leaders are the smallest elements on their cycles, the last leader will often be found early, and many iterations of the loop in function `main` will be in vain. Also, after the first cycles have been found, the search length in function `IsLeader` will be seriously restricted.

We try to motivate the average improvement by heuristic Fastbreak2 on algorithm A1. Without the heuristic, `IsLeader(i)` will perform n/i steps on the average, if encountering an element with index less than i is modelled by a binomial trial with success probability i/n.

From [3, Theorem 6.12] it follows that the j-th largest cycle, $1 \leq j \leq \Theta(\log n)$ has an average length of $c_j = \Theta(n/2^j)$. The leader of the j-th largest cycle then has an expected index of $i_j = n/c_j$.

When `IsLeader(i)` is called, all cycles j with leaders smaller than i have already been found. This is the case for $i_j = n/c_j < i$, i.e. for $j < \log i$ on the average. The variable `remaining` will then have the value

$$n - \sum_{j=1}^{\log i} c_j = \Theta(n/i)$$

```
void *array[n]; /* the array to be permuted */
extern int pi(int x); /* permutation as oracle */
extern int IsLeader(int i,int remaining);

int PermuteCycle(int i)
{ int j=i; /* loop variable */
  int length=0; /* cycle length */

  do{ j=pi(j); length++; /* follow cycle */
      exchange(i,j); /* exchange a[i] and a[j] */
  } while (j!=i);
  return length;
}

void main(void)
{ int i; /* loop variable */
  int remaining=n; /* number of elements to be permuted */

  for(i=0;i<n;i++)
   if(IsLeader(i,remaining)){
    remaining-=PermuteCycle(i);
    if(!remaining) break;
   }
}
```

Figure 3: The modified permutation algorithm in heuristics Fastbreak1 and Fastbreak2

```
extern int pi(int x); /* permutation as oracle */
extern int TestHypo(int j,int i);

int IsLeader(int i,int remaining)
{ int j=i; /* loop variable */
  int flag=1; /* assume i is leader */
  int steps=0; /* number of steps along cycle */

  do{ j=pi(j); steps++; /* follow cycle */
      if(!TestHypo(j,i) || (steps>remaining)){ flag=0; break; }
      /* until assumption turns out false or gone too far */
  }while(j!=i); /* or cycle is completely traversed */
  return flag;
}
```

Figure 4: The modified cycle leader check in heuristic Fastbreak2

Consequently, if the constant factor c in this term is less than 1, then the heuristic will reduce the runtime by a factor c.

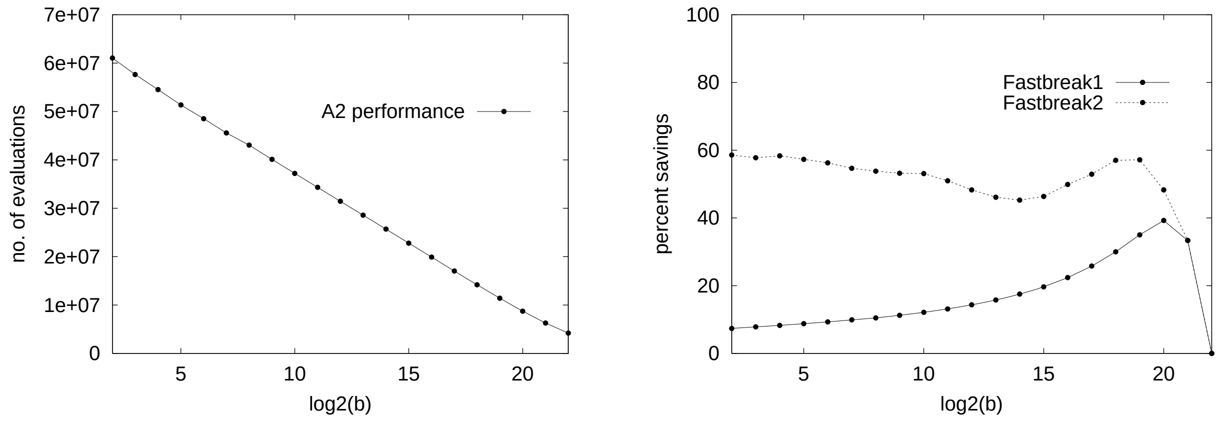
Note, that this is not a rigorous proof. As no element can be chosen twice, the success probability after k steps of `IsLeader(i)` is $i/(n-k)$. However, the value of variable `remaining` will have an influence only when it has become sufficiently small compared to n, i.e. when the large cycles have been found. In this case, $j \ll n$ and the average number of steps performed by `IsLeader(i)` will not differ much from n/i.

3 Experiments

We implemented the three algorithms in C on a SUN Sparc Ultra 60 with 2 processors at 300 MHz, 2 GByte of main memory, running the Solaris 8 operating system. We compiled the programs with the GNU C compiler using option `-O3`.

We started the algorithms with randomly chosen permutations for $n = 2^5, \ldots, 2^{26}$, with 50 permutations for each n. On our platform, we achieve about $2.7 \cdot 10^6$ evaluations of π per second per processor. Hence, the runtimes for larger values of n and the available time on the platform determined the number of permutations we could test for each n.

Figure 5a shows the influence of b on algorithm A2 for a permutation with $n = 2^{22}$ elements



(a) Runtime (b) Influence on heuristics

Figure 5: Influence of parameter b on algorithm A2

generated with seed 100. The number of evaluations linearly decreases with $\log_2 b$ from $0.69 \cdot n \log n$ for $b = 2$ to n for $b = n$. Figure 5b shows b's influence on the heuristics. Over a wide range, i.e. $b = 2, \ldots, n/4$, the savings due to heuristic Fastbreak1 were 7 to 35% increasing with b. The savings due to heuristic Fastbreak2 vary between 45 and 58%. We are not aware of recommendations and thus set b to $b = \sqrt{n}$. This choice is somehow arbitrarily, but at least avoids having b fixed over a wide range of values for n, while keeping b small compared to n, which might be necessary for very large values of n such as $n \geq 2^{48}$.

The permutations to be tested were stored as an array. Hence, the available main memory gave an upper bound on n. We created the permutations with the procedure given in [3, Program 6.1] for randomly permuting an array. Here, the array was initialized with `pi[i]=i`. We used `lrand48` as pseudo random number generator and used multiples of 100 as seeds.

To compare execution time, we used the number of calls to `pi` from `IsLeader`[1] instead of wall-clock time in order to abstract from unwanted influences on execution time such as cache behaviour, which might have significant impact depending on the permutation. We also felt that the overhead incurred by implementing the heuristic can be neglected, especially under the assumption that the evaluation of permutation π will often dominate the time spent in one iteration of the loops in `main` and `IsLeader`.

First, we found that the algorithms A1 to A3 without the heuristic perform on the average about $2/3 \cdot n \log n$, $1/3 \cdot n \log n$, and $1 \cdot n \log n$ evaluations of π, respectively.

[1]Thus, we only compare the times needed to find the leaders and do not count the time to permute cycles.

We recorded the savings due to both heuristics as the percentage of the runtime of the unmodified algorithm for each n, each algorithm and each permutation chosen. Figure 6 depicts the savings as scatter plots; the average savings are marked by a line.

We see from Figure 6 that the savings vary considerably, but that 10 to 15 percent of the runtime can be saved on the average by Fastbreak1 over all values of n on each algorithm. We also see that the savings slowly decrease with growing n. We also see that the variation gets even larger for Fastbreak2. On the average, the savings by Fastbreak2 are in the range of 30 to 50 percent and thus visibly larger than those by Fastbreak1. The average savings for algorithms A1 and A2 appear to be a constant percentage over all values of n. For algorithm A3 the savings decrease with growing n.

4 Conclusions and Open Problems

The savings from heuristic Fastbreak2 appear to be a constant factor of the runtime of the unmodified algorithms. The size of the constant factor depends on the algorithm, but not in an obvious way. Astonishingly, algorithm A3, which is the slowest among the unmodified algorithms, also has the least profit from the heuristics.

The overhead for heuristic Fastbreak2 clearly is larger than for Fastbreak1. Hence, whether the additional overhead for Fastbreak2 is worthwhile must be decided depending on the particular application. In any case, the savings due to Fastbreak1 are already large enough to include this heuristic in implementations of in-situ permutation algorithms.

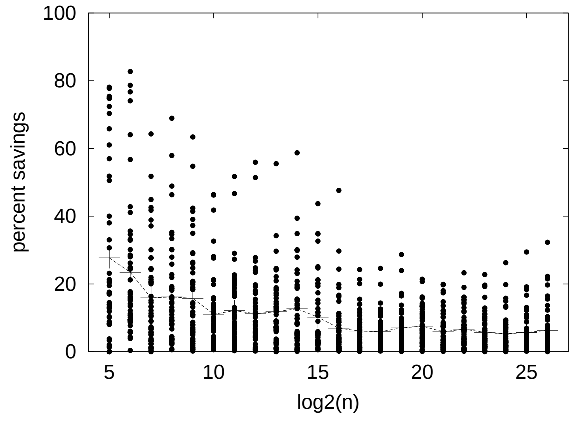
A formal average-case analysis of the algorithms' runtime when using the heuristics remains an open problem.

References

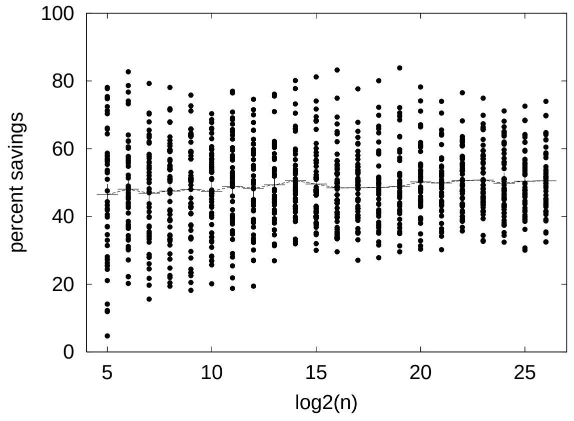
[1] Faith E. Fich, J. Ian Munro, and Patricio V. Poblete. Permuting in place. *SIAM Journal on Computing*, 24(2):266–278, April 1995.

[2] Donald E. Knuth. Mathematical analysis of algorithms. In *Proc. of IFIP Congress 1971*, Information Processing 71, pages 19–27. North-Holland Publ. Co., 1972.

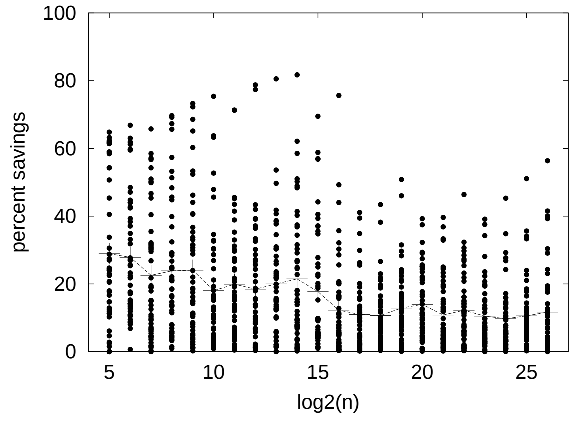
[3] Robert Sedgewick and Philippe Flajolet. *An Introduction to the Analysis of Algorithms*. Addison Wesley, Reading, Mass., 1996.



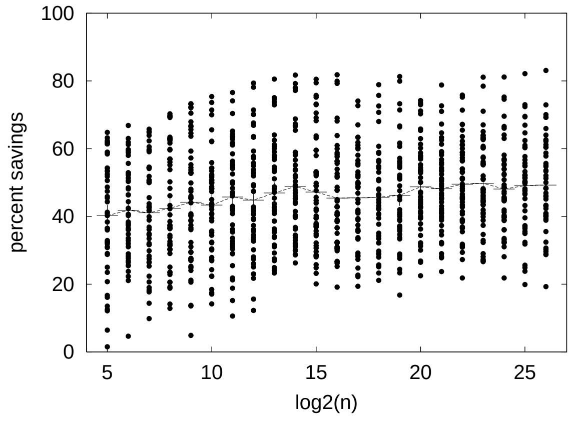
(a) A1 with Fastbreak1



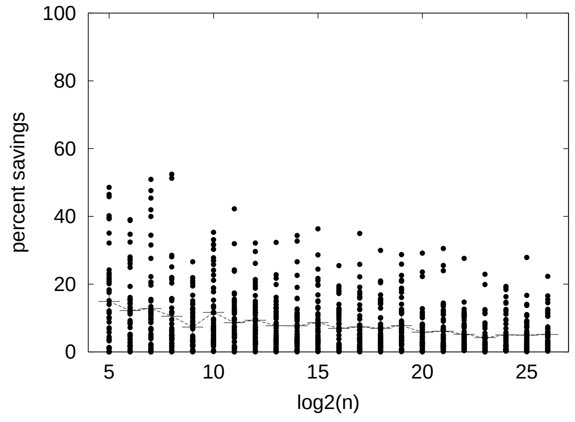
(d) A1 with Fastbreak2



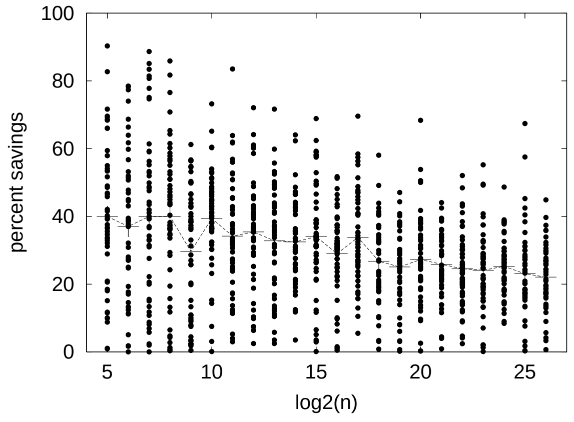
(b) A2 with Fastbreak1



(e) A2 with Fastbreak2



(c) A3 with Fastbreak1



(f) A3 with Fastbreak2

Figure 6: Savings due to the heuristics Fastbreak1 and Fastbreak2