Parallel Exploration of the Structure of Random Functions

Prof. Dr. Jörg Keller, FernUniversität, 58084 Hagen, Germany

Abstract

We present an algorithm to compute the size of all components and cycles of the graph associated with a random function on n elements, without explicitly constructing that graph. The algorithm is shown to have runtime $O(n^2)$ in the worst case and $O(n\sqrt{n})$ on the average. Its space requirements are much smaller than n, and it can be efficiently parallelized, making it suitable for large n. Its simple control structure also allows an efficient implementation on programmable hardware. We report on our first experiences with sequential and parallel implementations. The algorithm can be applied to compute the period (and sizes of unwanted additional cycles) of pseudo-random number generators and stream ciphers with a non-bijective state-transition function, such as the A5 algorithm used in the GSM standard.

1 Introduction

Pseudo-random number generators (PRNGs) are often used in cryptography, e.g. to generate session keys or to encrypt stream ciphers. PRNGs consist of a state, taken from a finite set of states S, and a state-transition function $f : S \to S$. Clearly, a PRNG also must provide an output function to derive the next generated number from the current state, but we will not need that function here.

Among other parameters, the period of the PRNG, i.e. the minimum number of state-transitions before a state occurs again, is an important criterion for the deployment decision of a particular PRNG. Often, the state-transition function contains additional, smaller cycles besides the main cycle that defines the period. In this case, there exists a set of unwanted states in which the PRNG should not start, because they lead to a much smaller period.

While the period of a simple PRNG can be analytically derived, this is often not possible for more sophisticated PRNGs. In this case, the period might be found experimentally. The challenge lies in the size $n = |S|$ of the state set S, which might be as large as $n = 2^{40}$.

The problem is especially important for PRNGs with a non-bijective state-transition function f, because there the probability of a short period is much higher than for a bijective function f [1]. An example of a non-bijective PRNG is the A5 stream cipher that encrypts packets transmitted between a mobile phone and its base station in the GSM network [5]. Also, while in bijective state-transition functions the lengths of the small cycles give the number of unwanted start states, this is not true for non-bijective state-transition functions, where each component of the associated graph consists of a cycle to which trees of states are attached. Hence, to determine the number of unwanted states one needs the sizes of the components. Note that the longest cycle does not necessarily represent the largest component.

In the sequel, a bijective function f will also be called permutation, a non-bijective function f will also be called mapping.

We consider the graph $G = (V, E)$ where $V = S$ and $E = \{(s, f(s)) \mid s \in S\}$. A bijective state-transition function partitions the states into cycles of successive states. Algorithms to compute the cycles' lengths can be derived from in-situ permutation algorithms, their runtime is $O(n^2)$ in the worst case, and $O(n \log n)$ in the average case. For a survey, see e.g. [2]. A key technique is to avoid explicit construction of G as n can be as large as 2^{64} in the targeted applications. Consequently, even parallel algorithms for the problem have been developed [3].

However, all these algorithms fail for non-bijective state-transition functions, which partition the states into components, each consisting of one cycle with one or several attached trees. Each tree is directed towards its root, which sits on the cycle. We present an algorithm to compute the structure of such a graph. The algorithm needs time $O(n^2)$ in the worst-case and $O(n\sqrt{n})$ on the average. The algorithm allows to give the size of the main cycle and the size of the component containing it, and to reveal additional, small components that contain the set of "unwanted" states, from which the PRNG should not start.

The algorithm can easily be parallelized because of its large number of independent tasks. Its simple control structure and small storage requirements also allow for an efficient implementation in hardware. We present first experimental results for sequential and parallel implementations.

The remainder of the paper is organized as follows. In Section 2, we describe the problem to be solved in detail and outline the basic algorithmic technique to be used. In Section 3, we present the complete algorithm and its analysis. In Section 4, we present some experiments done with an implementation. In Section 5, we conclude.

2 The Structure of Permutations and Mappings

Consider the graph G defined in the introduction. If function f is bijective, then it can be inverted and each node has a unique predecessor. Thus, G only consists of cycles. To find the structure of f, we need the length of each cycle. To report each cycle only once, we define the leader

x	$f(x)$
0	3
1	2
2	1
3	4
4	7
5	7
6	3
7	6
8	7

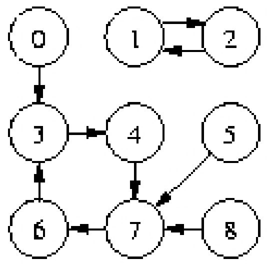

Figure 1 An example of a non-bijective function and its graph. We see two components: one consisting of a cycle of length 2, and one with 7 elements, a cycle of length 4, and two trees.

of the cycle to be the element with smallest index on it. Now one can start in an arbitrary element i and follow the cycle until either an element $j < i$ is met or i is reached again. In the former case, i is not the leader, in the latter case, i is the leader and the cycle length can be reported as the number of steps done. This simple algorithm has worst case runtime $O(n^2)$ and was shown by Knuth [4] to have expected runtime $O(n \log n)$ if each bijective function f is equally likely. Note that the graph G is not constructed, and hence memory consumption is very small, making the algorithm's code and data fit into the first level cache of the processor it runs on.

If function f is not bijective, then some nodes of G might have no predecessor and some nodes might have more than one predecessor. The graph G consists of one or more components, each consisting of one cycle, and one or more trees directed towards their roots. The tree roots are sitting on the cycles. The leaves of the trees are the nodes with indegree 0. The structure of the graph G consists of a list of all components. For each component, the number of nodes in this component, the length of the cycle, and the height of the largest tree in the component are reported. Each component can be uniquely identified by the leader of its cycle.

An example of a non-bijective function f and its associated graph G is given in **Figure 1**.

When we want to treat non-bijective functions, there are two main differences to Knuth's algorithm that we must take into account:

1. An element in a tree is never reached again. Instead, the algorithm must find out when a cycle has been reached.
2. For an element i in a tree, the method to abort the search when $j < i$ is reached makes no sense, because at least the cycle must be reached to identify the component of i.

Basically, we track the complete path from i to a cycle of G. In order to detect that we have reached a cycle, we proceed as follows. While following the path through G starting in i, we record from time to time an element x that we visit; we call x an *anchor.* After each step, we test whether we have reached x again. By doubling the time between successive recordings, we will finally detect that we are on a cycle. The number of recordings to be done is logarithmic in the length of the path. The number of steps taken so far is an upper bound on the length t_i of the tree path from i to the cycle.

When we have detected that we are on a cycle, then we run around the cycle once more, thereby obtaining the cycle length l_i and identifying the cycle leader cl_i. We know the number of steps t we have done in total so far, and we know the distances of the anchors and the cycle length. Hence we can compute the first anchor x that lies on the cycle and the last anchor x' on the tree path. They could be apart by $t \leq t_i/2$. By running from x around the cycle for $l_i - t$ steps, we reach a point x'' on the cycle from which we reach x again in t steps. Now we simultaneously follow the paths from x' and x''. After each step, we check whether the elements reached are identical. If this happens for the first time, we have reached the entry point into the cycle. Now we can compute the length t_i of the tree path.

The complete procedure described will be called `findcycle()` in the sequel. It is called with parameter i, and returns the tree path length t_i (which is zero if i is on the cycle), the cycle length l_i, and the cycle leader cl_i.

Lemma 1 *If the path through G starting in i follows a tree for t_i steps before reaching a cycle of length l_i, then at most $2.5t_i + 4l_i$ steps are taken by procedure* `findcycle()`.

Proof: The tree path from i until entering the cycle is run only once, taking t_i steps. If an anchor was recorded immediately prior to entering the cycle, then at most t_i steps could be done on the cycle before the first anchor sitting on the cycle is recorded. If i itself is on the cycle and the cycle length l_i is a power of two, then we will have been round the cycle once before we have a chance to reach the anchor in time. To reach the anchor again, we have to do a second round of the cycle. Determining the cycle leader necessitates running around the cycle one more time, hence the number of steps so far is at most $2t_i + 3l_i$.

To compute the length of the tree path, we first do $l_i - t$ steps on the cycle and then t steps on the cycle and on the tree path. Hence we do $l_i + t \leq l_i + t_i/2$ steps. In total, we have $2.5t_i + 4l_i$ steps. q.e.d.

3 The Complete Algorithm and its Analysis

The complete algorithm now consists of the following steps, which are performed for each element i, where $i = 0, 1, 2, \ldots, n-1$:

1. Execute procedure `findcycle()`, described in the previous section, on i and obtain cl_i, l_i, t_i.
2. If a component with cycle leader cl_i has already been found, then increase its size by one and re-

place its current tree height h by $\max\{t_i, h\}$.

3. If no component with leader cl_i has been found yet, then create a new one with identifying cycle leader cl_i, cycle length l_i, size 1, and tree height $h = t_i$.

Lemma 2 *The algorithm correctly computes the structure of G.*

Proof: As procedure `findcycle()` is executed for every element, it is surely executed for every leader, and thus every component is detected. Now, consider a particular component. The cycle length is identical no matter which element of this component first initiated the call to `findcycle()`. Hence, the cycle length of that component is correctly computed. As `findcycle()` is executed for each tree element of this component, the tree height of the component is the maximum of all heights observed. The size is incremented each time an element of the component is detected. Thus, the component's size is computed correctly as well. q.e.d.

Lemma 3 *If function f can be evaluated in constant time, then the algorithm has a worst-case runtime of $O(n^2)$ and an expected runtime of $O(n\sqrt{n})$ if each random mapping is equally likely.*

Proof: The runtime of procedure `findcycle()` is clearly $O(n)$, since every element on the path from i to the entry of the cycle is visited only once, every element on the cycle is visited at most twice, and there are no more than n distinct elements. For each visit, at most a constant number of operations plus one evaluation of function f are necessary, which take constant time.

As procedure `findcycle()` is called n times, the worst case runtime is $O(n^2)$. This bound is sharp. For $f(x) = x + 1 \mod n$, the runtime is $\Theta(n^2)$.

The exact runtime of procedure `findcycle()` is $O(t_i + l_i)$. The average tree height and average cycle length of a component are $O(\sqrt{n})$ if each mapping is equally likely [1]. Hence the average runtime of procedure `findcycle()` is $O(\sqrt{n})$. The time to search for the existence of a component is at most $O(\log n)$ with a suitable data structure. As the above steps are executed for n elements, the average runtime of the algorithm is $O(n\sqrt{n})$. q.e.d.

The algorithm can be parallelized for a multiprocessor by distributing the iterations of its outer loop over the available processors. As there is no communication necessary between different iterations, there is no difference between shared-memory and message-passing systems. Each processor maintains its “local” structure. In the end the local structures must be merged into one, which can be considered as a kind of post-processing which can be neglected here because the structures' sizes will be small and the processing time to merge them into one will be dominated by the algorithm itself.

The load should be well balanced because of the large number of iterations to be distributed, where each iteration has the same expected runtime. This is a sharp difference to algorithms for permutations. Those are difficult to parallelize, because the runtime of one iteration could be proportional to the length of the longest cycle, which has an expected value of approximately $0.624 \cdot n$ [6, p. 358].

n	av.	var.
1000	1.839181	1.327707
2000	1.803836	1.457447
4000	1.955553	1.880349
8000	1.863986	1.618747
16000	2.177782	2.542506
32000	1.819600	1.496151
64000	2.179798	2.560023
128000	1.976349	1.897604

Table 1 Average and variance of runtimes in relation to $n\sqrt{n}$

4 Experiments

We implemented the algorithm from the previous section in C on a PC with a Pentium II processor running at 333 MHz under the FreeBSD Unix operating system. We compiled the program with `gcc` version 2.7.2.3 using options `-O6 -fexpensive-optimizations -pedantic -fomit-frame-pointer -DINTEL_GCC`.

We tested the algorithm for $n = 1000, 2000, 4000, \ldots, 128000$. For each n, we generated 100 mappings on which we ran the algorithm. Each mapping was generated by allocating an array of size n and randomly assigning values between 0 and $n - 1$ to all array elements. As PRNG, we used lrand48 with seeds $0, \ldots, 99$.

We counted the number of evaluations t of function f as a measure of runtime, to avoid influences from caches and the like. For each experiment, we recorded $t/(n\sqrt{n})$. We computed the average and the variance over the 100 experiments done for each value of n. The results are depicted in **Table 1**.

We see that in the average, about $2n\sqrt{n}$ evaluations of function f are performed. The variance is quite large which can, at least partly, be attributed to the small number of samples. The fastest instance we saw needed $0.51 \cdot n\sqrt{n}$ evaluations, the most expensive instance $6.96 \cdot n\sqrt{n}$ evaluations, which means a factor of almost 14 in performance. The experiments support our average-case analysis; the constant factor is pretty small. Thus, the algorithm is usable in practice.

In addition, we measured the absolute runtime of the algorithm on our platform. Experiments with $n = 128,000$ took about 4 seconds on the chosen platform which means an execution rate of about $23 \cdot 10^6$ evaluations of f per second.

We also implemented a parallel variant of the algorithm on the Störtebecker cluster of the University of Lübeck. We used the PVM library to generate one process on each processor and distributed the iterations of the outer loop over the processes. In the end, we merged the processes' re-

sults, because one component could be detected by several processes.
For six processors, we achieved a speedup of 5.95 for a static mapping of loop iterations to processes, hinting an almost perfectly balanced load as expected. For the experiments we used a randomly chosen function with $n = 10^6$ to have enough iterations per processor.

5 Conclusions and Future Work

We have presented a new algorithm to compute the structure of random mappings, and given an analysis of its performance. We have indicated an application area for this algorithm. We have presented an implementation together with the results of preliminary tests on (pseudo-)randomly generated mappings. The experimental data support our claim that the algorithm is practical. Currently we are in the phase of obtaining more experimental data to be able to give answers on average cycle size and tree depths as well, both for generated mappings and mappings from applications. We hope to present these results in the final version of the paper.
In a further step, we will implement the algorithm on a multiprocessor to compute the structure of a stream cipher with a medium sized state space. For generators with large state spaces, such as A5/1 with 2^{64} states, the algorithm is not yet practical in its current form. However, we want to improve the algorithm with a two phase technique from [3] which could bring a further performance improvement.
Furthermore, with the simple control structure of the algorithm, it could be implemented in hardware, e.g. in FPGAs or simple ASICs. An average size ASIC could probably host about 16 instances of the parallel algorithm. Hence, a chip farm with 1,024 chips could be able to achieve 2^{39} evaluations per second. With it, we could tackle state spaces of 2^{40} within one day.

6 References

[1] W. G. Chambers. On Random Mappings and Random Permutations. In *Proc. Fast Software Encryption, 2nd International Workshop*, Leuven, Belgium, Dec. 1994. Lecture Notes in Computer Science 1008, pp. 22–28, Springer, 1995.

[2] F. E. Fich, J.I. Munro, P.V. Poblete. Permuting in Place. *SIAM Journal on Computing*, 24(2), pp. 266–278, 1995.

[3] J. Keller, J. F. Sibeyn. Beyond External Computing: Analysis of the Cycle Structure of Permutations. In *Proceedings Euro-Par 2001, European Conference on Parallel Computing*, August 28-31, 2001, Manchester, UK. Lecture Notes in Computer Science 2150, pp. 333–342, Springer, 2001.

[4] D. E. Knuth. Mathematical Analysis of Algorithms. In *Proc. of IFIP Congress 1971*, Information Processing 71, pp. 19–27, North-Holland Publ. Co., 1972.

[5] B. Schneier. *Applied Cryptography 2nd Edition.* John Wiley & Sons, New York, 1996.

[6] R. Sedgewick, Ph. Flajolet. *An Introduction to the Analysis of Algorithms.* Addison Wesley, Reading, Mass., 1996.