Parallel-External Computation of the Cycle Structure of Invertible Cryptographic Functions

Andreas Beckmann
Martin-Luther-Universität Halle-Wittenberg
Institut für Informatik
06099 Halle (Saale), Germany
andreas.beckmann@informatik.uni-halle.de

Jörg Keller
FernUniversität in Hagen
LG Parallelität und VLSI
58084 Hagen, Germany
joerg.keller@fernuni-hagen.de

Abstract

We present an algorithm to compute the cycle structure of large directed graphs where each node has exactly one outgoing edge. Such graphs appear as state diagrams of finite state machines such as pseudo-random number generators in cryptography. The size of the graphs necessitates that the adjacency list is kept on hard disks. Our algorithm uses multiple processing units, so that a parallel storage system has to be employed to store the graph. We present experimental results for randomly chosen graphs, and for the graph of the A5/1 generator used in GSM mobile phones.

Keywords*: parallel storage systems, parallel graph algorithms, cryptography*

1. Introduction

Generators for pseudo-random numbers and stream ciphers can be modeled as deterministic finite state machines, that receive no more input once they are initialized to a certain state by an init vector or key value. Their state transition graphs are simple: each node has exactly one outgoing edge, each weakly connected component consists of one cycle and a number of trees, directed towards their roots, the roots being on the cycle. The edges are given via the state transition function. The analysis of such graphs may reveal weaknesses, such as short periods with non-negligible seed probability, or predictable sequences. By their very nature, such investigations cannot be done analytically but must be done experimentally. Yet, the sheer size of the state space (typically 2^{64} and beyond) prevents a construction of the graph in memory. Previous algorithms (see e.g. [5]) have attempted to explore those graphs partly by following paths through the graphs, although with limited success.

Our first contribution is a parallel-external algorithm that reduces such a graph, given on a parallel storage system, until it fits into the main memory of a parallel computer. We also give algorithms to compute the cycle structure for the reduced graph in memory. For graphs of a size too large to store the edges on hard disks, we investigate graphs with a certain property: one can identify a small subset of the nodes such that each cycle contains at least one node of the subset. A prominent example of this kind of graph is the graph of the A5/1 stream cipher generator, used in GSM mobile phones to encrypt data between the mobile phone and the base station. Thus, to explore the cycle structure, we can restrict ourselves to construct the graph of the subset nodes in a preprocessing phase. This graph is still large, but should fit into a parallel storage system with a capacity of less than 1 terabyte. We explore this graph by the parallel-external algorithm.

We apply our algorithms to randomly chosen graphs, and to the graph of the A5/1 stream cipher generator. We thus have derived the first algorithm able to completely explore the A5/1 cycle structure. Still, the construction of the graph of subset nodes takes about 6 months on a 32-processor cluster, and only one third of the work is done yet. Our preliminary results indicate that the weakly connected components of the A5/1 graph are shallow, in contrast to randomly chosen graphs, so that our reduction algorithm needs fewer rounds than expected.

The remainder of the paper is organized as follows. Section 2 introduces the relevant notation, background information, and the multi-step approach to reduce and explore the graphs. In Section 3 we develop our algorithms, and in Section 4 we apply them to the A5/1 stream cipher generator. In Section 5 we present our experimental results. Section 6 concludes.

2. Notation and Background Information

Let R be a large but finite set of size n, and let $f : R \to R$ be an arbitrary function. We consider f given as an oracle: we give x to the oracle, and get $f(x)$ in return, in constant time. In practice, f is either realized by a piece of code that is unknown to us, or as an array holding $f(0)$ to $f(n-1)$. If there is also an oracle to return $f^{-1}(y) = \{x \mid f(x) = y\}$ when given y, we call f *efficiently invertible*. Note that this is always the case if f is given as an array. An example function is the state transition function of the A5/1 key stream generator in mobile phones.

In order to be able to talk about the structure of a function, we define the graph induced by f as $G_{R,f} = (V = R, E = \{(x, f(x)) \mid x \in R\})$. If f is given as an array, this array then forms the adjacency list of $G_{R,f}$. The induced graph is a directed graph that consists of one or more weakly connected components (wCC). Each wCC consists of one cycle, and one or more trees, which are directed towards their roots. The roots are those nodes on the cycle with an indegree of at least 2.

We are interested in the number and lengths of the cycles, and also in the sizes of the wCCs. The cycles form the periods of the generator whose state transition function induces the graph, and the size of a wCC relative to the graph size gives the probability that the generator is initialized to use a certain period. Furthermore, Flajolet and Odlyzko [3] have analyzed average case properties of graphs induced by randomly chosen functions, which they call *random mappings*. If the particular graph deviates notably from those average values, this may hint towards (cryptographic) weaknesses of the generator behind.

There is a clear difference between the two possibilities how f is realized: if f is given by an unknown piece of code, we may turn it into an array representation (as long as that fits into memory or on disk), while the opposite is not possible. If f is given as code, we are able to formulate sequential and parallel algorithms [5, 6] with memory consumption much smaller than n, that compute the cycle and wCC structures of G_f, even if n is so large that an array representation would not fit into memory or on disk. However, those algorithms tend to be not very efficient. If f is given as an array, we need at least enough memory to hold the array. If n is too large to store it in main memory, we need to store it on hard disks. Obviously, one cannot handle the case where f is given as an array but does not fit onto disks. Yet, it is questionable how such a case should arise.

As our application has a very large n, it demands a multi-step approach:

Step 1: We start with a code-representation of f. By restricting ourselves to graphs with an additional property yet to be defined, we are able reduce the graph G_f to a smaller one that is given as an array-representation, and that fits onto hard disks.

Step 2: We employ an external-parallel algorithm to further compact the graph until it fits into the main memory. We do this by repeatedly removing the leaves of the graph.

Step 3: We explore the cycle structure of the remaining graph internally.

Should it happen that the graph is reduced to a collection of cycles before it fits into the main memory, we are able to employ an efficient parallel-external algorithm for so-called permutations [2]. Yet the probability for this to happen is small, as we can expect that the cycle lengths sum up to $\Theta(\sqrt{n})$ nodes. As one can normally expect that hard disk size n is less than the square of main memory size m, m will be larger than $\sqrt{n}$.

Note that during the reductions, the cycle length information is maintained exactly, and the component size information is maintained approximately. Note further that the additional property is only exploited in the first step. The second and third step algorithms work for arbitrary graphs.

We first state some simple bounds about the complexity of computing the graph structure of $G_{R,f}$, if enough memory is available (step 3).

Lemma 1 *With a memory consumption of $O(n)$ words, the structure of $G_{R,f}$ can be computed in time $O(n)$. If f is efficiently invertible, n bits suffice.*

Proof: We allocate an array a of n words, which we initialize to zero. We start at a node x with $a[x] = 0$, and follow the unique path from x through the graph $G_{R,f}$, by repeatedly executing $x = f(x)$. At each node x we visit, we set $a[x] = 1$. We stop when we reach a node x with $a[x] \neq 0$. This is exactly the case when we have gone round the cycle of the weakly connected component where x resides. We compute the cycle length and the cycle leader[1] by going round the cycle once more, and thus also know how far the node x is from its tree root. Thus we have discovered a new wCC. Now we pick another node x that is still unvisited, and follow the path with setting $a[\,] = 2$, until we reach a node u that is already visited. If $a[u] = 2$, then we have again found a new wCC. If $a[u] < 2$, we have reached a known wCC, and go again along the path, this time setting $a[\,] = a[u]$. This is repeated until all nodes are visited. Then a contains for each node, the index of its wCC.

If f is efficiently invertible, the array a only consists of bits. We follow the first path. When we have detected the cycle, we determine all tree roots on the cycle, which are the nodes x on the cycle with $|f^{-1}(x)| \geq 2$. Then, for each tree

[1] The node on the cycle with lowest number, which uniquely identifies the cycle and the wCC.

root, we completely explore the tree with the help of f^{-1} in linear time, and mark the tree nodes as visited. Note that we have to treat the root node x separately, because the set of its predecessors $f^{-1}(x)$ also contains a predecessor on the cycle, which must be ignored in the tree exploration. Thus, the complete wCC is visited. We search for another node (in another wCC) that is unvisited, and repeat this scheme, until all nodes are visited.

In both schemes, each edge is only traversed a constant number of times, and thus the time to execute the algorithms is $O(n)$. ∎

There also exist parallel algorithms to compute the wCCs of such graphs [4], so that step 3 of our multi-step approach is already solved.

3. Parallel-External Algorithms

3.1. Algorithm for Step 2

We now consider the case where f is given as an array that will not fit into main memory. It will be stored as a file on the hard disk as an adjacency list, i.e. as a number of pairs $(x, f(x))$. From the relation between disk sizes and main memory sizes m, we safely deduce that $n < m^2$.

We consider the file with the adjacency list to be sorted according to the first entries of the tuples. If not, we apply an external parallel sorting algorithm. Our goal is to continually reduce the size of this graph by cutting off the leaves, until the list is short enough to fit into the main memory, so that we can resort to one of the algorithms shown before.

If the graph has the properties of a random graph [3] we can expect that $p_0 = n/e$ graph nodes are leaves, and we can expect to achieve a reduction by a factor of $(1-1/e)$ in the first step. Let us denote by R_i the subset of nodes containing those nodes with a maximum distance of at least i from some graph leaf. Thus, R_0 is the set R of all nodes, R_1 contains all nodes except leaves, and so on. There is some i_{max} such that $R_i = R_{i_{max}}$ for all $i \geq i_{max}$, those sets contain the nodes on the cycles. Obviously,

$$R = R_0 \supseteq R_1 \supseteq \cdots \supseteq R_{i_{max}} = R_{i_{max}+1} = \cdots$$

and $R_{i+1} \setminus R_i$ is the set of nodes being removed in round i of the algorithm. For random graphs, we expect $i_{max} = O(\sqrt{n})$, as this is the expected depth of the largest tree, with a constant between 1 and 2. Also, the probability of a node not being in R_{i+1} is

$$p_{i+1} = e^{-1+p_i}$$

with $p_0 = 0$ [3, Thm. 2], so that the reduction factor $(1-p_{i+1})/(1-p_i)$ gradually increases towards 1, until only the cycles remain. Yet for small i, i.e. in the first rounds of the reduction, we can expect a reasonable reduction: for the first 5 reductions, it will be at most 0.86, for the next 5 reductions, it will be at most 0.92. Figure 1 depicts the sizes of R_i in relation to n, logarithmically scaled, so that it approaches 0.5 if $|R_i|$ approaches $\sqrt{n}$. The steps are measured relatively to $\sqrt{n}$ as we expect the number of steps i_{max} to be less than $2\sqrt{n}$.

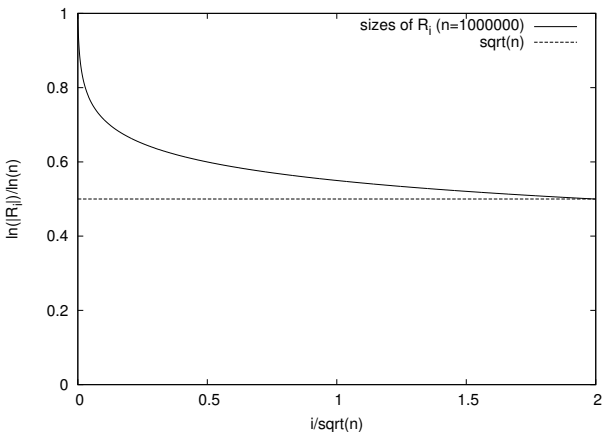

Figure 1. Analytical sizes of sets R_i, for $n = 10^6$.

If the graph is non-random, then two things may happen: either the components are more shallow than in a random graph, i.e. the trees are not as deep, and we can peel off many more leaves in each round. It may also happen, if a tree consists of a list of nodes, that only one leaf is cut off in each round, and we would need $n - m$ rounds to reduce to m nodes. In that case, one should think of something else.

When we remove a leaf x, we store this information in the leaf's successor node $f(x)$. Thus, we provide each node with information about the size and the depth of the subtree of ancestors already reduced. Initially, this information is zero, so it can be generated during the first step.

In order to realize a leaf-cutting step, we apply an external sort algorithm to the adjacency list to sort it according to the second tuple entries into a second file, i.e. we receive a list $(y, f^{-1}(y))$, sorted according to y. This list can be read from the disk block by block. Because of the sorting, we find all values of y for which no entry is present. Those are the y with no preimage, i.e. the leaves in the graph. We also read the adjacency list block by block and write it back to the file, after we have removed the edges starting from leaves. The targets of those edges, i.e. the successors of the leaves, are stored in a third file, together with their subtree information. This file is externally sorted afterwards. Then this file and the adjacency list are read again, and kind of merged: for each entry in the third file, the subtree information of the same node is updated in the adjacency list.

External sort of n items that reside on a disk having page

size b items with a processing unit having a memory that can hold m items can be achieved in time $O(n \log n)$ for internal processing and $O(n/b \log n)$ page accesses. External sort can be efficiently parallelized if each disk can be reached from each processing unit, i.e. if a parallel storage system is available. Thus we obtain the following Lemma.

Lemma 2 *Each reduction step needs $O(n \log n)$ internal processing time and $O(n/b \log n)$ accesses to disk pages of size b.*

3.2. Algorithm for Step 1

We now present an algorithm for the case of a code representation of function f, and for very large n. This algorithm works only for a subset of possible functions f: we must be able to specify a function-specific node property $\mathcal{P}$ such that each cycle contains at least one node with this property. We should also be able to efficiently test whether a graph node has property $\mathcal{P}$ and to generate graph nodes with this property. Let C be the subset of all nodes with property $\mathcal{P}$, the set of the *candidates*. It is always possible to specify such a property: if we choose a property that is true for every node x, e.g. $0 \le x \le n-1$, then each cycle obviously contains such a node. More exactly, it consists only of those nodes, and $C = \{0, \ldots, n-1\}$.

In order to be advantageous, the number of nodes with property $\mathcal{P}$ should be notably smaller than n. Typically, a considerable fraction of the candidates will still reside in trees of $G_{R,f}$ and not on cycles. As the candidates are nodes of the graph $G_{R,f}$, they have the property that from each candidate u there is a unique path of length $l(u)$ to another candidate $v = g(u)$. Thus, the set $C \subseteq R$ of the candidates together with the function $g : C \to C$ induces a graph $G_{C,g}$ where the edges have lengths, i.e. the edge $(u, g(u))$ has length $l(u)$.

A nice property of the graph $G_{C,g}$ is

Lemma 3 *The wCCs and cycles in $G_{C,g}$ correspond one-to-one to the cycles in $G_{R,f}$, and corresponding cycles have identical lengths.*

Thus, if we can compute the candidate graph efficiently, i.e. faster than solving the cycle structure problem on $G_{R,f}$ directly, we are able to reduce the problem of computing the cycle lengths for f to a smaller problem of size $n' = |C| < n$, which in turn can be solved by the algorithm of the previous subsection. The difference between the original and the reduced problem is that f is represented as a piece of code while g is represented as an adjacency list. Also, this method cannot be applied recursively. If we were able to formulate a second candidate set C' with property $\mathcal{P}'$ in $G_{C,g}$, we could and should have applied property $\mathcal{P}'$ directly on the original graph $G_{R,f}$.

If the candidate nodes have the additional property of being about evenly distributed among all nodes, then the sizes of the wCCs in $G_{C,g}$ relative to each other give an approximation of the ratio of the sizes of the corresponding wCCs in $G_{R,f}$. A similar approximation is obtained for the tree depths in $G_{R,f}$.

In order to compute the candidate graph, we first generate all candidates in $O(n')$ time. How this can be done depends on the application. An example is given in the next section. Then, we start at each candidate u and follow the path until we reach $v = g(u)$. If the maximum indegree of a candidate u in $G_{C,g}$ is k, then each edge in $G_{R,f}$ is traversed at most k times, and thus we need at most kn time. Most of the time however, the paths will be largely disjunct, so that the computation takes time $O(n)$. The algorithm is given in the following pseudo-code:

```
foreach u in C
  v := u;
  l := 0;
  repeat v := f(v); l := l+1 until v in C
  add (u, v, l) to G_C,g
```

The computation of the candidate graph can be easily parallelized: as f is realized with a piece of code, each processing unit in a multiprocessor or cluster computer can evaluate f. Each processing unit is assigned a number of candidates $u_1, \ldots$ for which it is to follow the paths from u_i to $v_i = g(u_i)$. If the assignment is done in blocks of fixed size, load balancing can be achieved in an easy way. One processing unit works as a master, and as soon as a processing unit has worked its block, it asks the master for a new block. Each processing unit writes the generated edges of the candidate graph to a file. When all processing units are done, the files can be merged into one.

Lemma 4 *If the candidate graph has a size of n', then it can be computed with $O(n)$ time of internal processing, and $O(n'/b)$ accesses to disk pages of size b. Both can be optimally parallelized.*

4. Application to A5/1

The A5/1 algorithm is a stream generator used to encrypt speech data between a GSM mobile phone and a base station. It has a 64 bit state which is distributed over three linear feedback shift registers (LFSR), as given in Figure 2 [1]. Each LFSR i realizes a cycle of length $2^{l_i} - 1$, containing all states except zero. Here, l_i is the length of LFSR i. We assume that the registers are never initialized to zero. The clock taps (C1, C2, C3) of the registers are used to compute a majority on their values, and only those registers agreeing with the majority are clocked. This means that in each clock cycle, at least two of the registers are clocked, and

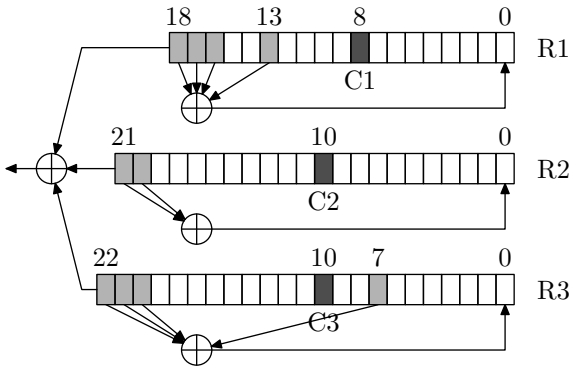

Figure 2. A5/1 stream cipher generator.

each register is clocked in 3 out of 4 cases. The A5/1 can be efficiently inverted.

When considering the cycle structure of A5/1, we can make the following assumption: each cycle must contain every non-zero state of R3 at least once. This is clear, as the register R3 can remain without clock for at most 19 cycles, because otherwise the register R1 would have to contain only 1s, which would lead to a feedback of zero in order to avoid a cycle on $1 \ldots 1$, or only zeroes, which is excluded. Hence the cycle length is a multiple of $2^{23} - 1$ R3 clocks. As R3 is clocked in 3 out of 4 cases on average, this will normally take a multiple of $(4/3) \cdot (2^{23} - 1)$ clock cycles.

We consider an arbitrary non-zero value of R3, e.g. $1 \ldots 1$, and define the property $\mathcal{P}$ of a state u as follows: when in state u, the register R3 must have this value, the state u has a predecessor, and the predecessor state has a different value of R3. Each cycle thus contains at least one state with property $\mathcal{P}$, it is easy to generate all states with this property, and it is easy to check whether a state has this property. The number of states with that particular value of R3 is $(2^{22} - 1) \cdot (2^{19} - 1) < 2^{41}$, one fourth of them being leaves, and one fourth of them having a predecessor with the same value in R3, so that the number of candidates is less than 2^{40}. As the distances between candidates is $(4/3) \cdot (2^{23} - 1)$, the effort to compute the candidate graph is less than 2^{63}, and can be reduced by parallelization.

Additionally, we can use the facts that A5/1 can be efficiently inverted, that many of the candidate nodes are leaves, and that the wCCs of A5/1 tend to be shallow. We can find the leaves by starting at candidate nodes and explore the A5/1 graph in inverse direction. If the candidate node is the root of a tree of non-candidates, then we know it is a leaf. In order to improve runtime, we do only a depth-restricted exploration, and remove only those leaves that can be found fast. The determination of the threshold depth requires some care. For the A5/1, a threshold of 8,800 was found to be sufficient. All leaves in the candidate graph thus revealed can be removed if we are mainly interested in the number of wCCs and the cycle lengths. Our experiments (see next section) revealed that about 99.6% of the candidate nodes encountered were found to be leaves in the candidate graph of the A5/1.

The effort can be further reduced by taking into account that there are implementations that allow to perform several clock cycles in one computation. This is possible as the clock and feedback taps are not lower than bit 7, and thus feedback does not influence clocking or feedback for at least 8 clock cycles.

With both improvements, our computational effort is about 2^{55} evaluations of the state transition function which is feasible on a large parallel machine or with the help of special purpose hardware.

Now we have reduced the problem to an external problem of size 2^{40}. As a hard disk today contains 250 gigabyte, i.e. about 2^{38} bytes, the problem can be stored on 128 hard disks, assuming that we need 16 byte to store an edge, and that we need to store two files. Taking into account that we can remove many leaves when generating the candidate graph of the A5/1, the problem reduces to about a size of 2^{32} and is likely to fit onto a single hard disk.

As we will be able to store 2^{29} edges in a 4 gigabyte main memory, we will need a reduction factor of 8 to reduce the external problem to an internal problem. If we achieve a reduction in the A5/1 candidate graph which is at least comparable to that of random graphs (the experiments suggest it will be better), then we need 10 rounds of the algorithm for step 2. This means that we have found the first algorithm to explore the cycle structure of the A5/1 algorithm completely. The computation will take about 6 months on a 32-processor cluster. We expect the results to improve our knowledge about the security of and possible attacks against this algorithm.

5. Experimental Results

We have implemented the algorithms for all three steps to be run on cluster computers with the Linux operating system. Here we report on results for steps 1 and 2.

First, we ran the algorithm for step 2 on random graphs. We chose graphs of several sizes: $n = 10^5$, $3 \cdot 10^5$, 10^6, $3 \cdot 10^6$, 10^7, $5 \cdot 10^7$. Figure 3 depicts how the graph size shrinks with the rounds of the algorithm, for 10 graphs with $n = 10^5$ and $n = 10^7$, respectively. The scaling is identical to the one in Figure 1, as this scaling allows to compare results for several n in one figure. Although the results vary, which is to be expected as the variance in the behavior of random graphs is quite large, they match the analytical result quite well. This reduction would at least be expected for the A5/1 as well.

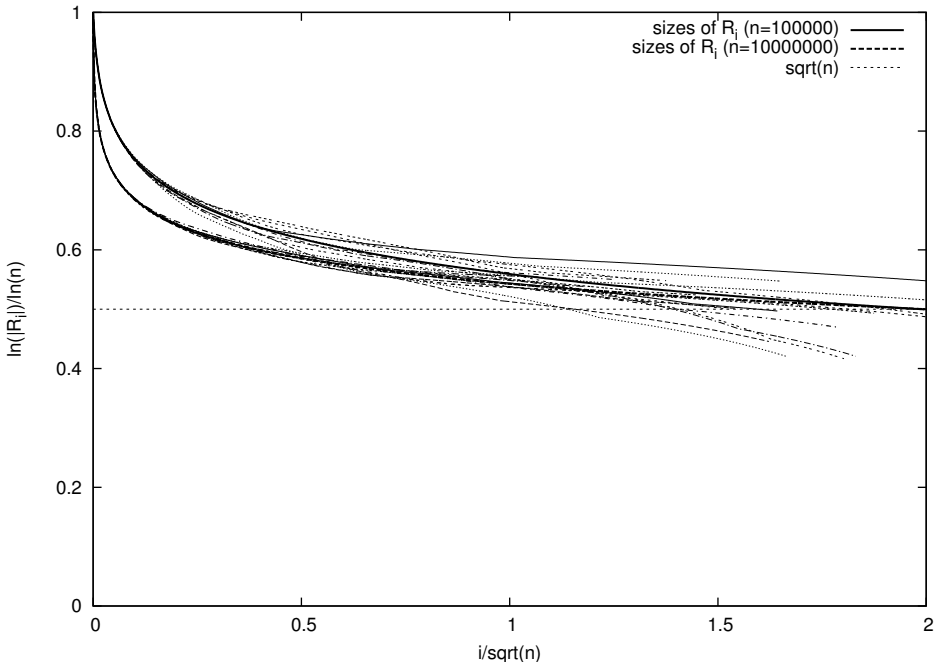

Figure 3. Experimental and analytical sizes of sets R_i of random graphs.

Next, we constructed reduced versions of the A5/1 generator, with 2^{32}, 2^{36}, and 2^{40} states respectively. For each of those, we computed the candidate graphs, and reduced those to their cycles. These reduced A5/1 generators are expected to have structural properties similar to the graph of the A5/1 generator. Studying these allows evaluation of different optimizations and discovery of new properties that hopefully apply to the full A5/1, too.

Figure 4 depicts the reduction of this graph, which is much faster than the corresponding analytical curve from Figure 1, which is also drawn. The lower curves depict the influence of the initial removal of candidate graph leaves during the construction of that graph, with two different thresholds. They show that there is an initial advantage, which is very valuable if only a few reduction steps are to be performed! One would need to zoom into the figure to quantify the advantage. Figure 5 depicts optimized curves for all three reduced versions. We see that the curves closely match, and that sizes below $\sqrt{n}$ are reached much faster than for random graphs.

Figure 6 depicts the results for the A5/1 graph as far it is computed at the present moment. We have tested about $3.63 \cdot 10^{11}$ candidates, of which most have been identified by inverse exploration to be leaves, so that a graph of about $1.33 \cdot 10^9$ nodes has remained and is reduced. We see that already for small values of i, its reduction is strong, so that it can be reduced much faster than a random graph.

As to parallelization, we feel from our preliminary experiments that the algorithm can be perfectly parallelized. Currently, the overall runtime is dominated by step 1 which spends most of its time without any I/O, only using the first level caches. We predict a runtime of about 6 months on a 32-processor cluster. So far we have constructed about one third of the candidate graph. Already, we have discovered more than 100,000 wCCs, which is much more than the approximately 100 wCCs that one would expect for a random graph of this size.

6. Conclusion

We have presented several parallel algorithms to explore the cycle structure of graphs induced by functions, differing by which graph sizes they can handle with either main memory or disk space, and by which function representation they can work with. We have applied those algorithms to the task of exploring the cycle structure of the state transition graph of the A5/1 stream cipher generator. Thus, we have presented the first algorithm to completely explore the cycle structure of the A5/1. Our experiments, as far as they have progressed, indicate that the A5/1 graph strongly deviates from a random graph. Future developments will try to develop better algorithms for step 2, that do not only cut off leaves, so that a certain rate of reduction is guaranteed. Also, we want to complete the analysis of the A5/1 graph.

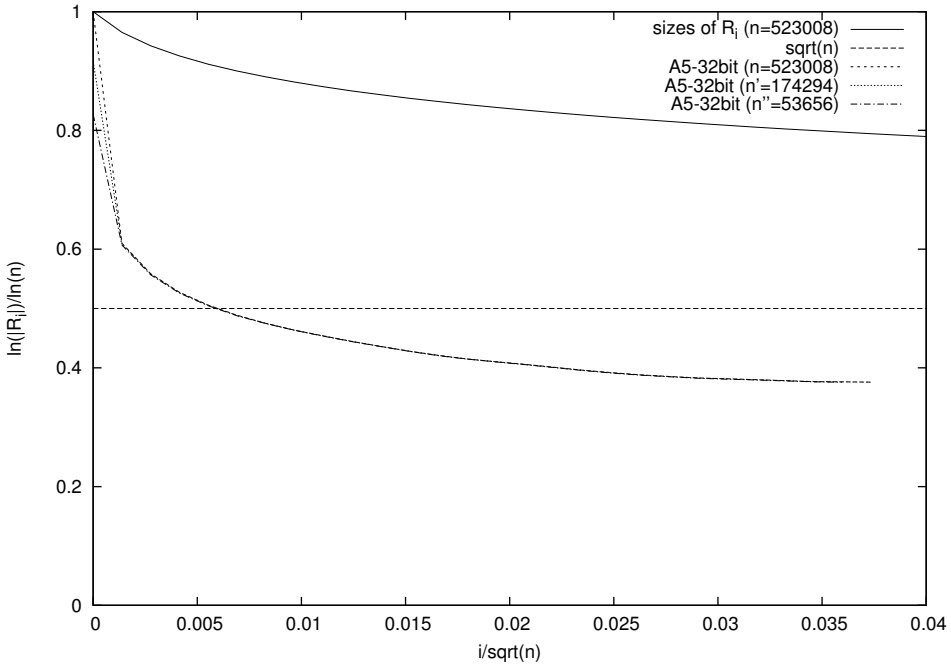

Figure 4. Candidate graphs of the reduced A5/1 generator with 2^{32} states.

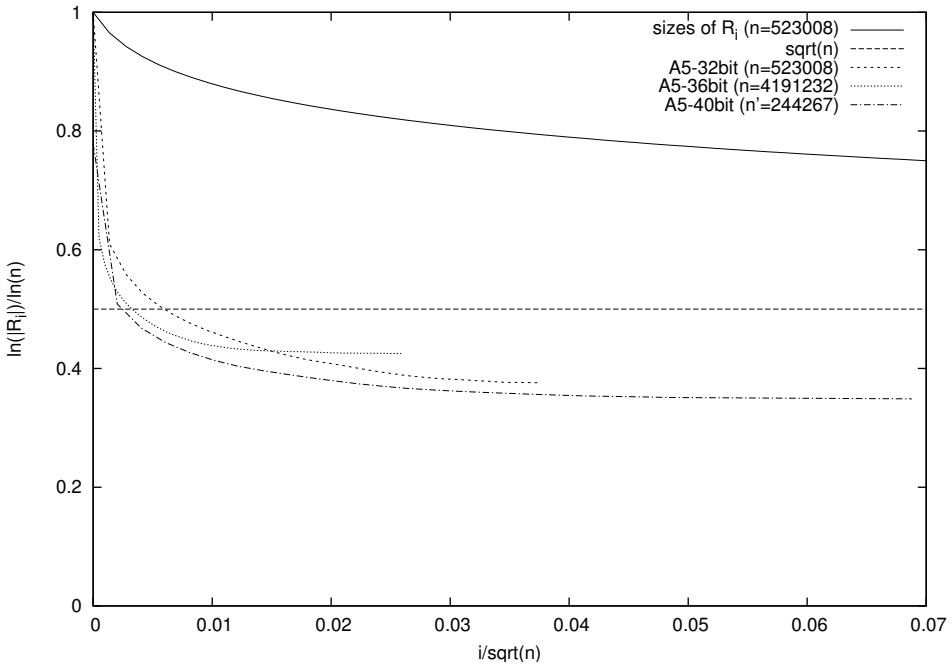

Figure 5. Optimized reductions for the candidate graphs of three reduced A5/1 generators.

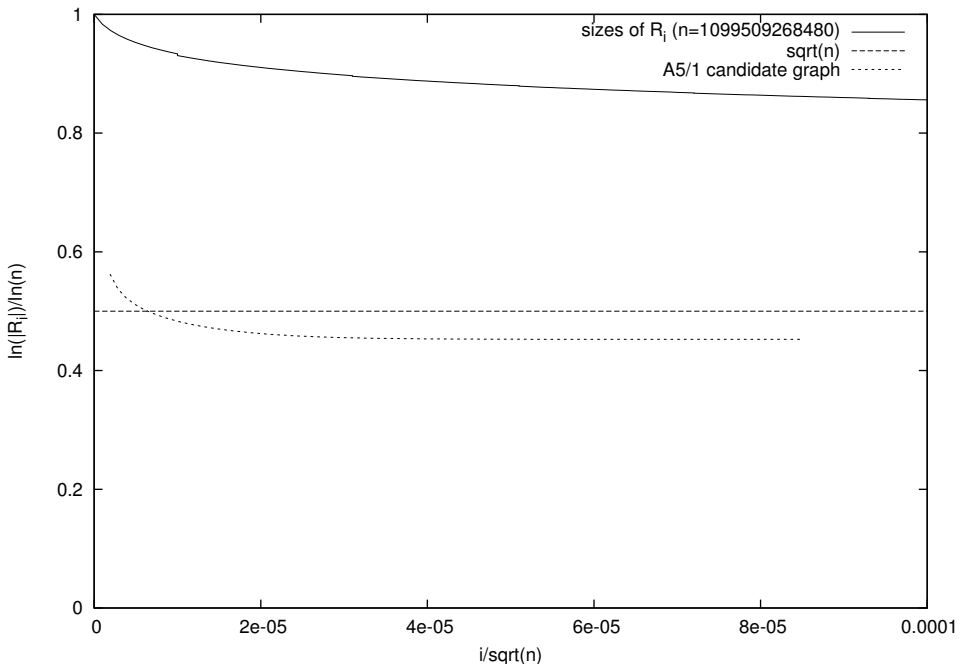

Figure 6. Experimental results for parts of the A5/1 candidate graph.

References

[1] A. Birykov, A. Shamir, and D. Wagner. Real time cryptanalysis of a5/1 on a pc. In *Presented at the Fast Software Encryption Workshop*, Apr. 2000. Available at http://cryptome.org/a51-bsw.htm.

[2] L. Boursas and J. Keller. Implementation and evaluation of a parallel-external algorithm for cycle structure computation on a pc-cluster. In U. Brinkschulte, J. Becker, D. Fey, K.-E. Großpietsch, C. Hochberger, E. Maehle, and T. A. Runkler, editors, *ARCS Workshops*, volume 41 of *LNI*, pages 348–357. GI, 2004.

[3] P. Flajolet and A. M. Odlyzko. Random mapping statistics. In *Proc. EUROCRYPT'89*, pages 329–354. Springer LNCS, 1990.

[4] Y. Han and R. A. Wagner. An efficient and fast parallel-connected component algorithm. *Journal of the ACM*, 37(3):626–642, 1990.

[5] J. Heichler, J. Keller, and J. F. Sibeyn. Parallel storage allocation for intermediate results during exploration of random mappings. In *Proc. 20. Workshop Parallel-Algorithmen, -Rechnerstrukturen und -Systemsoftware (PARS '05)*, pages 126–134, 2005.

[6] J. Keller. Parallel exploration of the structure of random functions. In *Proc. 6th Workshop on Parallel Systems and Algorithms (PASA 2002)*, pages 233–236. VDE Verlag, 2002.