RANDOMIZED PARALLEL ALGORITHMS FOR THE HOMING SEQUENCE PROBLEM

B. Ravikumar and X. Xiong
Department of Computer Science and Statistics
University of Rhode Island
Kingston, RI 02881, U.S.A.
({ravi,xiong}@cs.uri.edu)

Abstract – *Homing sequences play an important role in the testing of finite state systems and have been recently used in learning algorithms due to Rivest and Schapire [17] and Freund et al. [4]. It is well-known that every minimal DFA has a homing sequence of length $O(n^2)$ which can be constructed sequentially in time $O(n^3)$. But no efficient parallel algorithm was known for this problem. In this work, we present two RNC algorithms of time complexity $O(log^2 n)$ for this problem. We show that one of our RNC algorithms produces a homing sequence of length $O(n\ log^2 n)$ for almost all DFA's with n states using a random model of Traktenbrot and Barzdin. We also discuss connections between the homing sequence problem and other problems in the field of hardware fault-testing and protocol verification.*

1. INTRODUCTION

Locating the current state in a finite-state system is a fundamental problem in map-learning and robotics. This problem has many variations. The version we consider in this paper is the following: The transition function of the system is known, but the current state is not known. The states of the system are not externally visible but each state produces a visible output. The problem is to find an input sequence x such that the output sequence on x uniquely determines the state reached after applying x. An input sequence that achieves this goal is called a *homing sequence.* More precisely, suppose δ is the transition function and λ is the output function of a *deterministic finite automaton (DFA).* x is a (preset) homing sequence if for any two states q, q', if $\lambda(q,x) = \lambda(q',x)$ then $\delta(q,x) = \delta(q',x)$. A homing sequence brings a DFA to a known state and hence it is usually applied to a DFA before testing it (e.g. in protocol verification or fault-detection). Some recent applications of homing sequence can be found in the following list of papers: [17], [14], [4], [1] and [11]. These applications span a wide range of topics: exploration of an unknown environment and map learning [17], [4], [1], hardware fault-detection [14], reverse-engineering of protocols [11] etc. In view of the wide range of applications, it is of interest to design efficient algorithms to construct a homing sequence. In this work, we present efficient *randomized parallel* algorithms for this problem.

Every minimal DFA with n states has a homing sequence of length $O(n^2)$ and there is a well-known sequential algorithm to find a sequence of length $O(n^2)$ in time $O(n^3)$. It is also known that there are minimal DFA's with n states in which the shortest homing sequence is of length $\Omega(n^2)$ and therefore the sequential algorithm referred to above is essentially optimal (in the worst-case). However, this algorithm seems inherently sequential and designing a parallel algorithm for this problem seems hard. The best deterministic parallel algorithm for this problem (with a polynomial bound on the number of processors) is presented in [15] and has time complexity $O(\sqrt{n}\ log^2 n)$. In this work, we present a randomized parallel algorithm of time complexity $O(log^2 n)$ using a polynomial number of processors.

The remainder of the paper is organized as follows. In section 2 we introduce the basic definitions and terminology. In section 3, we present a randomized parallel algorithm for the homing sequence problem and show that the problem is in RNC. In section 4, we present another parallel algorithm. Although this algorithm is not an RNC, it is very efficient on almost all input instances. In section 5, we discuss some problems related to the homing sequence problem and conclude with some open problems.

2. DEFINITIONS AND PRELIMINARIES

A finite automaton (DFA) or a finite state system M is a 5-tuple $M = < Q, I, O, \delta, \lambda >$ where Q is a finite set of states, I is a finite set of input symbols, O is a finite set of output symbols, $\delta : Q \times I \to Q$ is the transition function and $\lambda : Q \times I \to O$ is the output function. Note that we use the *Mealy machine* model but all our results hold for *Moore machines* as well. Following [17], we will use the convenient abbreviation $q < x >$ to denote $\lambda(q, x)$ and qx to denote $\delta(q, x)$ throughout. If $R \subset Q$ and $x \in \Sigma^*$, we define $\delta(R, x) = \{\delta(r, x) | r \in R\}$. As above, we abbreviate $\delta(R, x)$ by abbreviated Rx. In a similar way, we define $\lambda(R, x)$ and abbreviate it $R < x >$. A string x is said to be a distinguishing string for two states $p, q \in Q$ if $p < x > \neq q < x >$.

Let $M = < Q, I, O, \delta, \lambda >$ be a DFA. A (preset) homing sequence for machine M is a string x such that for any $p, q \in A$, if $p < x > = q < x >$ then $px = qx$.

Given below is an example of a DFA M and a homing sequence for M.

Response to the Sequence 010

Initial State	Response to 010	Final State
A	000	A
B	001	D
C	101	D
D	101	D

The table shows that 010 is a homing sequence for M. Note that although 010 produces the same outputs from states C and D, the definition of a homing sequence is not violated since the same state is reached in both cases. On the other hand, 01 is *not* a homing sequence for M since $A < 01 > = B < 01 >$ but $A01 \neq B01$.

The study of homing sequences was initiated by Moore in the classical work [12]. Early work by him and others showed the existence of a homing sequence of length $O(n^2)$ for any minimal DFA M with n states. A sequential algorithm (taken from [17]) to construct a homing sequence is presented in **procedure** HSEQ. In line 3, the algorithm requires a distinguishing string x for the states ph and qh. We assume that distinguishing strings for all pairs have been already computed and stored in a table. These strings can be obtained by modifying the DFA minimization algorithm [6] and the existence of such a string is guaranteed for all pairs of states by the minimality of M. In fact, such a modified minimization algorithm can find the shortest distinguishing string for all pairs of states in time $O(n^2)$. It is well-known (see e.g. [6]) that for any pair of inequivalent states in an n state DFA, a distinguishing string of length at most $n - 1$ exists. It is also not difficult to see that the number of iterations of the *while* loop is bounded by $n - 1$ since the quantity $|Q < h > |$ increases by at least 1 after each iteration and $|Q < h > |$ is upper-bounded by n. Thus the length of the output h is bounded by $(n - 1)^2$. It is possible to implement this algorithm so that its sequential time complexity is $O(n^3)$ using standard techniques. In general the bound $O(n^2)$ on the length cannot be improved since examples of DFA are known whose shortest homing sequence is of length $\Omega(n^2)$ [7].

```
procedure HSEQ;
begin
  $h \leftarrow \epsilon$;
  while $\exists\ p, q \in Q$ s.t. $p < h > = q < h >$ and $ph \neq qh$ do
  begin
    Let $x$ be a distinguishing string for states $ph$ and $qh$;
    $h \leftarrow hx$;
  end;
  output($h$);
end
```

In this paper, we present a parallel algorithm for the homing sequence problem and analyze it on the PRAM model. We assume that the readers are familiar with this model. For a thorough discussion of PRAM with a large number of examples, we refer the reader to [8]. In the next paragraph, we describe a randomized PRAM model very briefly and define the classes NC and RNC.

A PRAM is a collection of sequential Random Access Machines (RAM) which interact through a shared memory. Each processor has its program and local memory as well as access to shared memory. During each instruction cycle, each processor reads the data from its local or shared memory, performs an instruction and stores the result in its local memory or shared memory. Various modes by which concurrent read/write access are permitted give rise to different models such as EREW, CREW, common CRCW, priority CRCW etc. The details can be found in [8]. A

PRAM algorithm that solves a problem Π is said to have a processor bound $P(n)$ if the number of processors used by the algorithm to solve instances of size n of Π is bounded by $P(n)$. Its time complexity $T(n)$ is defined as the maximum time taken by any processor to solve any instance of size n. An NC algorithm is a PRAM algorithm in which the number of processors is bounded by a polynomial in the problem size and the time bound is $O(log^k n)$ on inputs of size n (with a constant k). A randomized parallel algorithm is a PRAM algorithm in which the instruction set includes an instruction like $X \leftarrow TOSS$. This instruction randomly assigns to X the value 0 or 1 with equal probability. A randomized parallel algorithm (just like a randomized sequential algorithm) may produce different results on the same input since the computation sequence is governed not only by the input but also by the random numbers produced. A randomized algorithm is said to solve a problem with *high probability* of correctness if on any input of size n, the probability of error is bounded by $1/p(n)$ where $p(n)$ is a polynomial in n. Note that this kind of correctness bound is stronger than a mere constant bound (such as error bounded by 0.0001) since the error bound tends to 0 as the problem size increases. But there is an even stronger requirement (ultra-high probability) in which the error bound is required to be c^{-n} for some $c > 1$. The results of this paper can be easily translated so that the resulting algorithms are correct with ultra-high probability. But in order to compare different algorithms, we should keep one model consistently and we choose the *high probability* model. Finally, we define an RNC algorithm for problem Π as a randomized parallel algorithm that uses a polynomial number of processors on all inputs, has a poly-logarithmic time complexity on all inputs and is correct with high probability. It is possible to add an additional checking step so that our algorithm is always correct (a *Las Vegas type* algorithm). In this case, the time complexity becomes a random variable and a poly-log time bound will hold with a high probability.

3. A RANDOMIZED PARALLEL ALGORITHM

In this section, we present an RNC algorithm for the homing sequence problem. The algorithm presented below is based on a similar algorithm for synchronization sequence due to Eppstein [3].

Algorithm 1:

Input: A DFA $M = < Q, I, O, \delta, \lambda >$ where $Q = \{1, 2, ..., n\}$, and an integer m. (m is related to the error tolerance of the algorithm. See the discussion below.)

Output: A string h over the alphabet I. (h will be a homing sequence of M with probability at least $1 - c^{-m/n^2}$ for some $c > 1$.

Step 1. Generate a sequence of pairs of integers (i_1, j_1), (i_2, j_2),..., (i_m, j_m) where each i_k and j_k is a random integer between 1 and n, for all k.

Step 2. For each (i, j) do in parallel

if (i, j) is a pair in the above sequence, find a distinguishing sequence $d_{i,j}$ for the states i and j.

Step 3. Output the string $d_{i_1,j_1} d_{i_2,j_2} d_{i_3,j_3} ... d_{i_m,j_m}$.

First we show the correctness of the algorithm (in a probabilistic sense).

Theorem 1. *The probability that h is a homing sequence in Algorithm 1 is at least $1 - c^{-m/n^2}$ for some $c > 1$.*

Proof. (sketch) For simplicity, we assume that the DFA M is a permutation machine, i.e. for every state $p \in Q$ and each input $a \in \Sigma$ there is exactly one state p' such that $p'a = p$. (With minor modifications, the proof can be extended to the general case.) We view the string h as being created by successively concatenating an $x_{i,j}$ in m stages and let $h_r = x_{i_1,j_1} x_{i_2,j_2} ... x_{i_r,j_r}$. Define an equivalence relation on Q (with respect to h_r) as follows: p is equivalent to q if $p < h_r > = q < h_r >$ and let Q_1, ..., Q_k be the partition of Q induced. Let $Q'_i = \{q h_r | q \in Q_i\}$. Since M is a permutation machine, $|Q_i| = |Q'_i|$. Let $|Q_i|$ be n_i so $n_1 + ... n_k = n$. Define a random variable $X(r)$ as number of distinct outputs at time r. Our goal is to estimate a lower bound on the probability that $X(r+1) > k$ given that $X(r) = k$. We do this by considering a Markov chain $P = [p_{ij}]$ (of order n) such that $p_{i,j}$ is an estimate of the probability that $X(r+1) = j$ given that $X(r) = i$. Clearly, $X(r+1) > k$ given that $X(r) = k$ if $q_{i_{r+1}}$ and $q_{j_{r+1}}$ both belong to Q'_i for some i. The probability that this happens is given by

$$\left(\frac{n_1^2 + n_2^2 + ... + n_k^2 - n}{2} \right) \left(\frac{2}{n(n-1)} \right)$$

In Cauchy-Schwartz inequality:

$$\left(\sum_{i=1}^{k} x_i y_i \right)^2 \leq \left(\sum_{i=1}^{k} x_i^2 \right) \left(\sum_{i=1}^{k} y_i^2 \right)$$

choosing $x_i = n_i$ and $y_i = 1$ we get the inequality

$$(n_1 + n_2 + ... + n_k)^2 \leq (n_1^2 + n_2^2 + ... + n_r^2) \cdot k$$

Thus the above probability is lower bounded by $\left(\frac{n^2-nk}{2k}\right)\frac{2}{n(n-1)} = (n-k)/k(n-1)$.

Define the Markov chain P as follows:

$$p_{i,j} = \begin{cases} \frac{n-i}{i(n-1)} & \text{if } j = i+1 \\ 1 - \frac{n-i}{i(n-1)} & \text{if } j = i \\ 0 & \text{otherwise} \end{cases}$$

From the foregoing discussion it is clear that the probability that h_m is a homing sequence is lower-bounded by $[P^m]_{1,n}$. In the remainder of the proof we will obtain a lower bound on $[P^m]_{1,n}$ using the well known connection to the eigenvalues of the matrix P.

The eigenvalues of P are 1, $n(n-2)/(n-1)^2$, ... , $n/2(n-1)$ and 0 (in decreasing order of magnitude). The second eigenvalue is $n(n-2)/(n-1)^2$. It is not difficult to show that $[P^m]_{1,n}$ is given by:

$$[P^m]_{1,n} \simeq 1 - d\left(\frac{n(n-2)}{(n-1)^2}\right)^m$$

for some $d \geq 0$. From this, we can show that

$$[P^m]_{1,n} > 1 - c^{-m/n^2}$$

for some $c > 1$. This completes the proof. •

Next we will show that Algorithm 1 can be implemented as an RNC algorithm.

Theorem 2. *Algorithm 1 can be implemented on a probabilistic PRAM with time bound $O(log^2 n)$ using $O(n^7)$ processors and the length of the output produced will be $O(n^3 log\ n)$ for all inputs of size n. The algorithm is correct with (polynomially) high probability.*

Proof. We will assume that each processor of the PRAM has a source of randomness with which it can produce a sequence of unbiased, independent random bits. We will also assume that the randomness sources of different processors are independent of others. (In practice, of course, we will use a pseudo-random generator as a substitute for the source and provide different seeds for each processor.) With this assumption, Step (1) can be easily implemented in O($log\ n$) time using $O(m)$ processors as shown below. (Later we will choose m to be $O(n^2 log\ n)$; clearly, m is within the number of processors available.) Processor k will generate two sequences of $log\ n$ bits and these sequences will define two integers (in binary) i_k and j_k between 1 and n.

Step (2) is implemented as follows. We will describe an algorithm that finds $d_{i,j}$ a shortest distinguishing string for all pairs of states (i,j). This is done using a modified matrix multiplication. Let M be the input DFA. Define an associated $n \times n$ matrix T whose entries are letters over $I \cup \{\phi\}$ (where ϕ is a special symbol not occurring in I) as follows. Assume that I is ordered. $T_{i,j}$ is the first letter $a \in I$ such that $ia = ja$ or $i < a > \neq j < a >$, if such a exists, else it is ϕ. We define matrix multiplication as follows: Let A, B be two $n \times n$ matrices. Define AB as the product matrix (also $n \times n$) as: $[AB]_{i,j}$ is the string $a_{i,t}.b_{t,j}$ where t is the smallest integer such that both $a_{i,t}$ and $b_{t,j}$ are not ϕ. (If no such t exists, then define $[AB]_{i,j}$ as ϕ.) It is easy to see that the smallest k for which $[T^k]_{i,j}$ is not ϕ is the length of the shortest distinguishing string for the state pair (i,j). Thus the shortest distinguishing strings for all pairs of states can be extracted from the matrices T, T^2, T^3, ..., T^{n-1}. These can be computed in $O(log^2 n)$ using $O(n^7)$ processors.

Step (3) is easy to implement within the available resources. To obtain the claimed error bound and the output length, set $m = n^2 log\ n$. Then by Theorem 1, the error probability will be at most $1 - 1/cn$ (for some constant c) and the length of the output will be $O(n^3 log\ n)$. This is because the pairwise distinguishing strings found in Step (2) are the shortest ones and thus their lengths are bounded by $(n-1)$ [9]. This completes the proof. •

The performance of an NC (or RNC) algorithm for an optimization problem can be measured by three criteria: (i) parallel time bound, (ii) processor bound and (iii) the quality of the solution found. The algorithm presented above has a good time bound, but is not satisfactory on the other measures. The processor bound $O(n^7)$ is too large to make the algorithm practical. But this is a common feature of any parallel algorithm that depends on some variant of the transitive-closure problem as a subroutine. The output length $O(n^3 log\ n)$ is much larger than $O(n^2)$ length guaranteed by the simple sequential algorithm. In section 4, we will improve on these aspects.

4. AN IMPROVED RNC ALGORITHM

In this section, we will show how to get improved performance results for Algorithm 1 (and its variations) that holds for *most of the instances.* In order to present such results, we first introduce an average-case model. The standard way of creating a random instance of a DFA is to choose both δ and λ functions randomly. A stronger model was introduced by Traktenbrot and Barzdin [18]. In their model, only the output function λ is chosen randomly. An adversary chooses the transition function subject to the only condition that all states are reachable from the start state. Let G be the graph thus chosen by the adversary. The probability distribution is defined over all DFA's M_G which can be derived from this automaton graph by randomly selecting the output function. Throughout this section, the performance bounds of the algorithms are based on this stronger model. Clearly any upper bound on this model implies a matching bound in the weaker model in which both δ and λ are chosen randomly.

Let $P_{n,\varepsilon}$ be any predicate on an n-state automaton which depends on n, a confidence parameter ε (where $0 \leq \varepsilon \leq 1$). We say that **uniformly almost all automata** have property $P_{n,\varepsilon}$ if the following holds: for all $\varepsilon > 0$, for all $n > 0$ and for **any** n-state underlying automaton graph G, if we randomly choose λ, then with probability at least $1-\varepsilon$, the predicate $P_{n,\varepsilon}$ holds.

Lemma 3. *For uniformly almost all automata with n states, every pair of inequivalent states has a distinguishing string of length at most $2\ lg(n^2/\varepsilon)$.*

This lemma is Theorem 2 in [4]. Our next result is that Algorithm 1 can be implemented as a PRAM algorithm with better processor bound and shorter output length *uniformly on almost all DFA's.*

Theorem 4. *Algorithm 1 can be implemented on a probabilistic PRAM so that almost uniformly on all input DFA's with n states, its time complexity is $O(log^2 n)$, number of processors used is $O(n^2)$ and the output length is $O(n^2 log^2 n)$. The algorithm will be correct with high probability.*

Proof. (sketch) The essential idea behind the proof is that Step (2) (which is the most expensive step) of Algorithm 1 can be implemented in a more economical way on almost all instances. The new algorithm is slower (in the worst-case), and uses fewer processors. But it has a good time complexity on almost all instances. The idea is to modify the algorithm in Figure 3.8 of [6] and parallelize it. First we describe the modification needed so that the algorithm will find distinguishing strings $d_{i,j}$. (The present algorithm is aimed at minimizing a DFA, not to find pairwise distinguishing strings.) When a state pair (i,j) is placed in the list for a state pair (i',j'), we also store the symbol on which the transition between (i,j) and (i',j') occurred. The original algorithm's recursive marking step (5) in which a state pair (i,j) is marked is replaced by identifying $d_{i,j}$ as follows. Suppose (i,j) occurs in the list (i',j'). Inductively assume that $d_{i',j'}$ has been determined already. Then $d_{i,j}$ is $ad_{i',j'}$ where a is the symbol stored for the pair (i,j). Correctness of the algorithm is easy to see: If there is a transition on a from (i,j) to (i',j') and if i' and j' are distinguished by x then i and j are distinguished by ax. This can be implemented in parallel by a pointer jumping using string concatenation as the basic operation. The algorithm is parallelized as follows. The **for** loop beginning in line (2) is implemented in parallel for all pairs (p,q) in $F \times F$ or $Q-F \times Q-F$ *and is repeated t times* for a chosen value of t. Note that this redundancy is needed because of the parallelism. We call the new loop introduced as the outer loop. We can show by induction on i that a distinguishing string for all states (p,q) that have a distinguishing string of length at most i would have been found after the i-th iteration of the outer loop. The algorithm is repeated for t steps until no new state pair is marked during the t-th iteration. By Lemma (3), for almost all DFA's on n states, t is $O(log\ n)$. The inner loop can be implemented in $O(log\ n)$ time using $O(n^2)$ processors using pointer jumping as indicated above. The rest of the details are easy to fill in. This completes the sketch of the proof. •

It should be noted that the length of the output claimed above is obtained by averaging over all possible random moves of the randomized PRAM algorithm as well as over the random choices of the λ function. The other resources claimed are averaged only over the random choices of λ function.

We can improve Theorem 4 even further. We can design a randomized PRAM algorithm which has the same bounds as in Theorem 4 *except that uniformly on almost all instances, the output produced will be of length $O(nlog^2 n)$.* In order to achieve this result, we need to modify Algorithm 1. We also need a new concept called *a local homing sequence* introduced in [4]. Let $M = < Q, I, O, \delta, \lambda >$ be a DFA where $Q = \{1, 2, ..., n\}$. A string x is said to be a *local homing sequence* for state i if for any state $j \in Q$, $i < x > = j < x >$ implies $ix = jx$. Local homing sequences play a central role in [4]. They are also important in the deterministic parallel algorithm for the homing sequence problem presented in [15]. We now present Algorithm 2.

Algorithm 2:
Input: A DFA $M = < Q, I, O, \delta, \lambda >$ where $Q = \{1, 2, ..., n\}$.
Output: A string h over the alphabet I. (h will be a homing sequence uniformly on almost all instances with a high probability.)
Step 1. Generate a sequence of integers $i_1, ..., i_m$ where $m = O(n\ log\ n)$.
Step 2. For all i do in parallel
    if i occurs in the above list, obtain **a local homing sequence** L_i for state i.
Step 3. Output $L_{i_1} ... L_{i_m}$.

Note the two main differences between Algorithms 1 and 2. In Algorithm 1, a larger number (namely $n^2\ log\ n$) of pairwise distinguishing sequences were concatenated. In the latter, fewer (i.e. $n\ log\ n$) local homing sequences (which possess stronger properties than pairwise distinguishing sequences) are concatenated. The proof of the next result is left for the final version.

Theorem 5. *Algorithm 2 can be implemented on a probabilistic PRAM so that uniformly almost on all input DFA's with n states, its time complexity is O($log^2 n$), the number of processors used is O(n^2) and the length of the output produced is O($n\ log^2 n$). The algorithm will be correct with high probability.*

5. RELATED PROBLEMS

Some sequences closely related to a home sequence are the synchronizing sequence, the distinguishing sequence and the checking sequence defined as follows: A string x is a synchronizing sequence for a DFA M if $\delta(q, x) = \delta(p, x)$ for any two states p, q of M. A string x is a distinguishing string for M if for any pair of states p, q, $p < x > = q < x >$ implies $p = q$. A string x is a checking sequence for a DFA M if for any other DFA M' with fewer than or equal number of states the following holds: If p is a state in M and q is a state in M', then $p < x > \neq q < x >$. A minimal DFA need not have any of these sequences. But if a synchronizing sequence exists, there must be one of length bounded by $O(n^3)$. With some connectivity assumptions, a checking sequence of length O(n^3) can be shown to exist. A polynomial time algorithm (which is similar to the algorithm **procedure** HSEQ) to find a synchronizing sequence (if it exists) has been presented in [3]. The problem of designing a deterministic NC algorithm for synchronizing sequence was left open by Eppstein [3]. In the case of a distinguishing sequence, even if it exists, its length can be exponential in n. Further, even the decision problem of determining if a given DFA has a distinguishing sequence is PSPACE-complete [10]. There is a simple NC^1 reduction from the homing sequence problem to the synchronizing sequence problem. Our parallel algorithms for the homing sequence problem readily translate into parallel algorithm for the distinguishing sequence problem in the special case of *permutation* DFA's in which for every state and every input symbol there is exactly one incoming arc into that state labeled by the symbol. A randomized polynomial time algorithm for finding a checking sequence was presented in [10]. This algorithm can be easily modified into an RNC algorithm.

There are three analogous problems of computing the adaptive counterpart of homing, synchronizing and distinguishing sequences. See [9] for a definition of the adaptive versions of these sequences. Since a preset sequence is also an adaptive sequence, the upper bounds automatically hold in each case. In the case of distinguishing sequences, Lee and Yannakakis [10] present a polynomial time algorithm to determine if an adaptive distinguishing sequence exists and if it does, to find one. This also implies a polynomial bound on its length (which is defined as the height of the tree). At this point, we do not even know how to solve this decision problem efficiently in parallel; it may well be P-complete. Finally, we note that a deterministic NC algorithm is open even for the local homing sequence problem. This collection of problems thus indicates the importance and usefulness of randomness as a tool for parallel algorithm design.

6. CONCLUSIONS

In this paper, we presented randomized parallel (RNC) algorithms for finding a homing sequence in a given DFA. In a companion paper [16], we present the implementation of different sequential and parallel algorithms for this problem. Our preliminary tests indicate that it is hard to achieve a good speed up using parallel processors in the worst-case, but it is achievable when the instance is selected randomly according to the Traktenbrot-Barzdin model. If both λ and δ functions are selected at random, then the parallel program performs really well; it can even handle DFA's with millions of states. Although these programs are not directly based on the algorithms presented in this paper, they use randomization and some ideas from this work.

Our work on the homing sequence is part of a larger project in which we plan to develop good algorithms and programs (both sequential and parallel) for a number of testing problems. This includes, besides homing sequences, the synchronizing, distinguishing and checking sequences - their preset as well as adaptive versions. We are interested in the theoretical problem of (provably) efficient algorithm design as well as practical implementations which can solve instances arising from real applications. As far the homing sequence problem, it is not clear how to make further progress on the theoretical front. The only remaining problems are the design of a deterministic NC algorithm and of an algorithm with guarantee on the length of the output (e.g., relative to the length of the shortest homing sequence). These problems appear to be difficult. But much remains to be done on the practical implementation.

References

[1] M. A. Bender and D. K. Slonim, "The power of team exploration: Two robots can learn unlabeled directed graphs", *35th Annual IEEE Symposium on Foundations of Computer Science*, pp. 75-85 (1994).

[2] S. Cho and D. Huynh,"The parallel complexity of finite-state automata problems", *Information and Computation*, Vol. 97, No. 1, pp. 1-22 (1992).

[3] D. Eppstein, "Reset sequences for monotonic automata", *SIAM Journal on Computing*, Vol. 19, No. 3, pp. 500-510, (1990).

[4] Y. Freund, M. Kearns, D. Ron, R. Rubinfeld, R. Schapire and L. Sellie, "Efficient Learning of typical automata from random walks", *Proc. of 25th Annual ACM Symposium on Theory of Computing*, pp. 315-324 (1993).

[5] G. Holzmann, *Design and validation of protocols*, Englewood Cliffs, NJ, Prentice-Hall (1990).

[6] J. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley Inc. Reading, MA (1979).

[7] T. Hibbard, "Least upper bounds on minimal terminal state experiments for two classes of sequential machines", *Journal of the ACM*, Vol. 8, pp. 601-612, (1961).

[8] J. JáJá, *An Introduction to Parallel Algorithms*, Addison-Wesley Inc. Reading, Mass. (1992).

[9] Z. Kohavi, *Switching and Finite Automata Theory*, McGraw-Hill Publishers Inc. Second Edition (1978).

[10] D. Lee and M. Yannakakis, "Testing finite-state machines: state identification and verification", *IEEE Transactions on Computers*, Vol. 43, No.3, (1994).

[11] D. Lee and M. Yannakakis, "Testing finite state machines: fault detection", *Journal of Computer and System Sciences* Vol. 50, No.2, (1995).

[12] E. F. Moore, "Gedanken-experiments on sequential machines", pp. 129-153, *Automata studies*, Ed: McCarthy and Shannon, Princeton University Press, Princeton, NJ (1956).

[13] C. Papadimitriou and M. Yannakakis, "On complexity as bounded rationality", *26th Annual ACM Symposium on Theory of Computing* pp. 726-733 (1994).

[14] I. Pomeranz and S. Reddy, "Application of homing sequences in the fault-detection of sequential circuits, *IEEE Transactions on Computers*, (1994).

[15] B. Ravikumar, "A deterministic parallel algorithm for the homing sequence problem" (unpublished manuscript).

[16] B. Ravikumar and X. Xiong, "Implementing sequential and parallel programs for the homing sequence problem" (unpublished manuscript).

[17] R. L. Rivest, and R. E. Schapire, "Inference of finite automata using homing sequences", *Information and Computation*, Vol. 103, pp. 299-347, (1993).

[18] B. A. Traktenbrot and Ya. M. Barzdin, *Finite Automata: Behavior and Synthesis*, North-Holland Publishing Company, Amsterdam (1973).