View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Elsevier - Publisher Connector

ELSEVIER

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Theoretical Computer Science 328 (2004) 113–133

Theoretical Computer Science

www.elsevier.com/locate/tcs

Weak minimization of DFA—an algorithm and applications

B. Ravikumar[a,*], G. Eisman[b]

[a]Department of Computer Science, Sonoma State University, 1801 East Cotati Av., Rohnert Park, CA 94128, USA

[b]Department of Computer Science, San Francisco State University, San Francisco, CA 94132, USA

Abstract

DFA minimization is an important problem in algorithm design and is based on the notion of DFA equivalence: Two DFA's are equivalent if and only if they accept the same set of strings. In this paper, we propose a new notion of DFA equivalence (that we call weak-equivalence): We say that two DFA's are weakly equivalent if they both accept the same number of strings of length k for every k. The motivation for this problem is as follows. A large number of counting problems can be solved by encoding the combinatorial objects we want to count as strings over a finite alphabet. If the collection of encoded strings is accepted by a DFA, then standard algorithms from computational linear algebra can be used to solve the counting problem efficiently. When applying this approach to large-scale applications, the bottleneck is the space complexity since the computation involves a matrix of order $k \times k$ if k is the size of the underlying DFA M. This leads to the natural question: Is there a smaller DFA that is weakly equivalent to M? We present an algorithm of time complexity $O(k^3)$ to find a compact DFA weakly equivalent to a given DFA. We illustrate, in the case of a tiling problem, that our algorithm reduces a (strongly minimal) DFA by a factor close to 1/2.
© 2004 Elsevier B.V. All rights reserved.

Keywords: DFA minimization; Matrix power formula; Tiling automaton

1. Introduction

In this paper, we explore an application of finite automata to counting problems. To count the number of combinatorial structures of a certain size n, we map each structure to a string

* Corresponding author.
E-mail address: ravi@cs.sonoma.edu (B. Ravikumar).

0304-3975/$ - see front matter © 2004 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2004.07.009

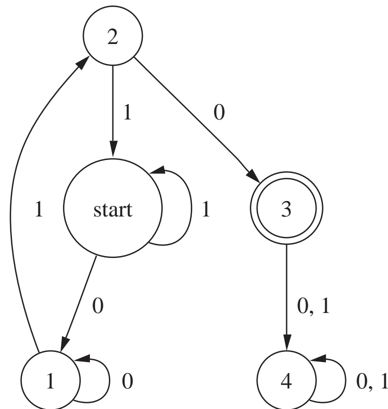

Fig. 1. DFA for $L = \{x|x$ ends with a unique occurrence of 010 as a suffix$\}$.

s from a carefully chosen alphabet *S* so that there is a 1-1 mapping between each object we are counting and a subset of strings over *S*. When establishing this 1-1 mapping, we can ignore the size parameter *n*, taking care only that it is length-preserving. If the set of such encoded strings is regular, then we have arrived at an efficient algorithm to solve our counting problem. This is a consequence of the fact that there is an efficient algorithm for counting the number of strings of length *n* accepted by a DFA.

We begin with an illustration of this approach to solve the following counting problem drawn from Liu's book [12]: "Find the number of *n*-bit binary sequences that have the pattern 010 occurring for the first time at the *n*th digit". For this problem, the encoding is trivial. We simply use the string itself as the encoding. The set of strings that have a unique occurrence of 010 as suffix is regular and can be accepted by the DFA shown in Fig. 1.

Let *M* be a DFA with *k* states. The transition matrix *A* of the DFA is a $k \times k$ matrix where a_{ij} is the number of transitions from state *i* to state *j*. It is well-known and is easy to show by induction on the length that the number of strings of length *n* that start in state *i* and end in state *j* is given by $[A^n]_{ij}$. Thus, the number of strings of length *n* accepted by DFA, *M* is given by xA^ny', where *x* is the start vector (of order $1 \times k$) such that $x_j = 1$ if and only if *j* is the start state, and y' is the transpose of *y* where *y* is the accepting vector (of order $1 \times k$) where $y_j = 1$ if and only if *j* is an accepting state. For the problem above, after we remove the *dead state* (since it does not contribute to the number of strings accepted) is easy to verify that

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

$x = (1\ 0\ 0\ 0)$ and

$$y^{\mathrm{T}} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}.$$

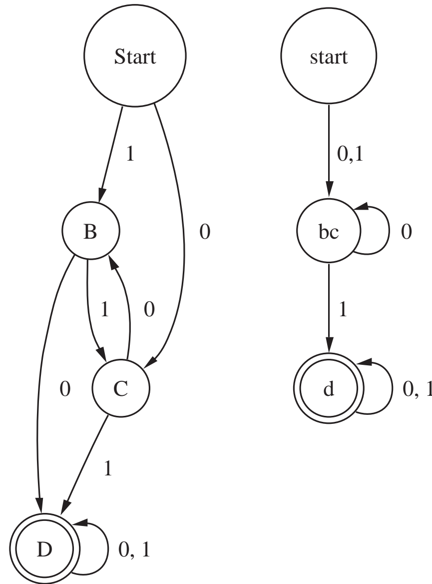

Fig. 2. A minimal, but not weakly minimal M and M' weakly equivalent to M.

The above technique to solve counting problems is sometimes called the transfer matrix method [26]. A DFA based approach can be very effective in creating the transfer matrix. For example, the closure properties of regular languages and other nice structural features can be used to design the DFA systematically. What motivated this work is the fact that the theory of finite automata can help in reducing the size of the transfer matrix as well. Optimizations such as removing unreachable states, identifying symmetries, minimization of the DFA (in the classical sense) are some of the ways to reduce the size of the transfer matrix. However, there is room for further improvement as the following example shows.

Example. Consider the DFA M shown in Fig. 2. It is a minimal DFA, but it can be seen that M' is a smaller DFA that is weakly equivalent to M.

Since we are only interested in using the DFA for counting the number of strings of given length, we need not limit the candidates (in the minimization) only to those DFA's that accept the same language. We can change the language so long as we do not change the number of strings accepted for each length.

This leads to the notion of weak equivalence of DFA's: We say that two DFA's are weakly equivalent if they both accept the same number of strings of length k for every k. The main goal of this paper is to describe an algorithm that finds a DFA M' with fewer states that is weakly equivalent to a given DFA M. In this preliminary version, we will focus on the algorithm for weak minimization but describe the applications only briefly, namely counting domino tilings and self-avoiding walks.

The rest of the paper is organized as follows. In Section 2, we describe an implementation that takes as input an integer k and automatically creates a DFA that accepts encodings of valid tilings of the $k \times n$ checkerboard (for all n). We also describe an implementation for automatically synthesizing a DFA for self-avoiding walks in the lattice grids of a specified width. In Section 3, we consider the time complexity of solving the counting problem after the transfer matrix has been created and present a theoretical lower bound on the minimum size of weakly equivalent DFAs. In Section 4, we present the main result of the paper—an algorithm for weak minimization and prove its correctness. We also estimate its time complexity, along with some implementation details. In Section 5, we state some open problems and directions for future work.

2. DFA design for counting problems

In this section, we are concerned with the design of DFA for two counting problems—tiling and self-avoiding walks. Both problems are parameterized by k, the number of rows in a rectangular grid. These and other counting problems require that software tools be developed to create DFA automatically from given specifications. We are currently in the process of creating a software engineering framework for such a general purpose tool. In a companion paper, we describe the details of such an implementation. However, in this section, we describe a more focussed and more narrowly defined effort to create programs tailor-made for the two specific applications. In Section 2.1, we will describe an algorithm to design DFA that accepts encoded tilings, and in Section 2.2, we describe a similar algorithm for self-avoiding walks on a rectangular grid.

2.1. DFA design for tiling problem

We are interested in counting the number of ways to use 2×1 tiles to completely tile a $k \times n$ checker-board with some removed squares. This problem is identical to counting the number of perfect-matchings of a subgraph G of the grid graph of order k by n. The connection between the two problems is as follows: We can create a graph from the checker-board by making each unremoved square a vertex, and connecting two vertices by an edge if the two squares share a common edge. It is clear that a 1×2 or a 2×1 tile corresponds to an edge, and thus a set of tiles corresponds to a matching. A valid tiling thus corresponds to a perfect matching. Counting perfect-matchings in a bipartite graph is known to be #P-complete [25]. Even when the instance is restricted to a subgraph of a grid graph, the problem seems to be hard, as recently shown by Ogihara and Toda [15]. The tiling problem with the number of rows restricted to a constant has also been extensively studied, see, for example [11,21,19,16], among others.

The encoding of tiling can be illustrated with $k = 4$. We represent the 2×1 vertical tile by the string $[0\ \ 0]'$, and the 1×2 horizontal tile by $[1\ \ 1]$.

Our alphabet consists of the pattern of vertical and horizontal tiles that are used to cover each column of the grid. Thus the alphabet is encoded as $\Sigma = \{a_0, a_3, a_9, a_{12}, a_{15}\}$ where the index of each symbol when expressed in binary indicates coverage with vertical or horizontal tiles. For example, $a_{12} = [1\,1\,0\,0]'$, meaning that the top two cells are covered with horizontal

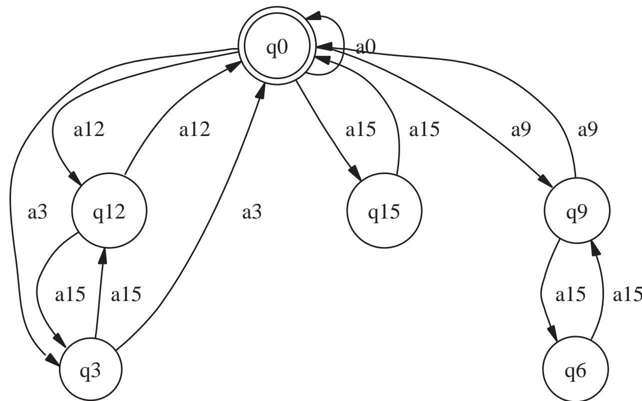

Fig. 3. A DFA for the $4 \times n$ tiling problem.

tiles and the bottom two with a vertical tile. The states of the DFA represent the binary pattern of horizontal tiles that begin in a column. Each 1 in the pattern dictates that its matching 1 occurs in the next column. There are 6 states possible: $Q = \{q_0, q_3, q_6, q_9, q_{12}, q_{15}\}$. The DFA M that accepts a valid tiling encoded using the above alphabet is shown in Fig. 3. (Here and throughout the paper dead states have been deleted since they do not contribute to the number of strings that are accepted.)

Thus, DFA's can be used to find the number of perfect-matchings in the $k \times n$ grid graph. But we are interested in a more general problem, namely to count the number of perfect-matchings in a *subgraph* of the grid graph. We can handle this more general problem as follows: The input to this problem is integer k, followed by a sequence I of squares (denoted by the (row number, column number) pair) that have been removed from the board. First, we design a DFA over a k-row alphabet in which each entry is {0, 1, 2}. Note that I also has information about the number of columns r in the board. 0 and 1 have the same meaning as above, but 2 now represents a removed square. Our program creates another DFA $M(I, r)$ that accepts all the strings of length r that have 2 in row i and column j if and only if the square (i,j) is in I. (There is no other restriction placed on the accepted strings.) Clearly, $M(I, r)$ has size at most $O(r * 2k)$. The number of perfect matchings in the grid graph associated with I (as the removed squares) is the number of strings of length r in the language $L(M(I, r)) \cap L(M(k))$. We use the standard pair construction to obtain a DFA that accepts this language and use the transfer matrix to count the number of perfect-matchings. It is clear from the above discussion that significant optimizations should be done to speed up the computation since the DFA's generated automatically through a series of mechanical steps tend to be rather large.

2.2. *DFA design for self-avoiding walks in a $k \times n$ grid*

Design of DFA to accept the encodings of self-avoiding walks in a $k \times n$ rectangular grid is more complex than the tiling problem described above. We describe our encoding scheme using $k = 2$ (namely paths in $2 \times n$ grid) as strings over a finite alphabet. We will

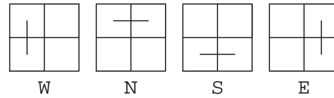

Fig. 4. Symbols to encode simple paths in a $2 \times n$ grid graph.

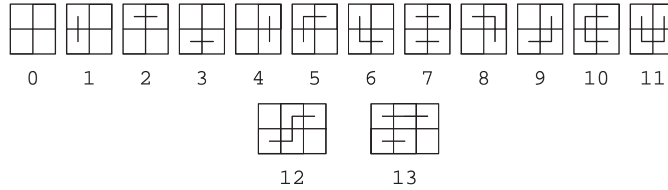

Fig. 5. States for DFA for $2 \times n$ self-avoiding walks.

again traverse the grid from left to right. At each step we will add a path segment connecting adjacent cells on the grid. There are four possible segments to use at each step as illustrated in Fig. 4.

In order to avoid duplicate strings that represent the same path, we will enforce the rule that inputs are prioritized in the order shown. For example, WN is the same path as NW, but only the former will be allowed. The states of the DFA will by the 2×2 and sometimes 2×3 cells containing the right most end of the path. There are 14 states as shown in Fig. 5.

All states are final except for states 7 and 13 where the path consists of two pieces that are not yet connected by a vertical edge. The transition table and matrix are shown below. (Note that the ordering we have placed on inputs sometimes restricts transitions between states.) The transition table and transition matrix are shown below.

$$\delta = \begin{bmatrix} - & W & N & S & e \\ 0 & 1 & 2 & 3 & 4 \\ 1 & - & 5 & 6 & - \\ 2 & - & 2 & 7 & 8 \\ 3 & - & 13 & 3 & 9 \\ 4 & - & - & - & - \\ 5 & - & 2 & 10 & 8 \\ 6 & - & 13 & 3 & 11 \\ 7 & - & 13 & - & 4 \\ 8 & - & - & 6 & - \\ 9 & - & 12 & - & - \\ 10 & - & 5 & 6 & - \\ 11 & - & 12 & - & - \\ 12 & - & 2 & - & 8 \\ 13 & - & - & 7 & - \end{bmatrix},$$

$$A = \begin{bmatrix} 0&1&1&1&1&0&0&0&0&0&0&0&0&0 \\ 0&0&0&0&0&1&1&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&1&1&0&0&0&0&0 \\ 0&0&0&1&0&0&0&0&0&1&0&0&0&1 \\ 0&0&0&0&0&0&0&0&0&0&0&0&0&0 \\ 0&0&1&0&0&0&0&0&0&0&1&0&0&0 \\ 0&0&0&1&0&0&0&0&0&0&0&1&0&1 \\ 0&0&0&0&1&0&0&0&0&0&0&0&0&1 \\ 0&0&0&0&0&0&1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&1&0 \\ 0&0&0&0&0&1&1&0&0&0&0&0&0&0 \\ 0&0&0&0&0&0&0&0&0&0&0&0&1&0 \\ 0&0&1&0&0&0&0&0&1&0&0&0&0&0 \\ 0&0&0&0&0&0&0&1&0&0&0&0&0&0 \end{bmatrix}.$$

Counting self-avoiding walks is an important problem in statistical mechanics and has been extensively studied [14]. The exact number of such walks is known for k up to 51 and has been accomplished through extensive computational effort. The DFA based approach is not likely to work for such large values of k since the number of states in the DFA will be very large. But asymptotic bounds on the number of self-avoiding walks can be obtained. Of special interest is the *connective constant*, μ defined as $\lim_{n\to\infty}(f(n))^{1/n}$ where $f(n)$ is the number of self-avoiding walks of length n in two-dimensional lattice grid. Surpassing the earlier work based on Golden and Jackson's (inclusion–exclusion) technique, Ponitz and Tittman [18] have obtained the best known upper-bound for the connective constant, namely $\mu \leqslant 2.6939$. Their method is based on DFA based counting. Let L be the set of strings that encode self-avoiding walks. Ponitz and Tittman designed a DFA for a subset of the complement of L. By determining the largest eigenvalue of the associated transfer matrix, they were able to establish the new upper-bound. Weak minimization technique proposed in this paper may be useful in solving similar problems, especially if we are only interested in approximate counting results.

3. Complexity of computing the matrix power formula

Counting the number of strings of length n accepted by a DFA raises many interesting computational problems. A direct approach would focus on the most efficient way to compute the matrix expression $xA^n y'$ where A is a $k \times k$ matrix of integers, x and y are k-vectors. Let $T(k)$ be the number of arithmetic operations (additions and multiplication) sufficient to compute the product of two $k \times k$ matrices. Coppersmith and Winograd's algorithm [2] establishes that $T(k) = (\mathrm{O}k^a)$ where a is approximately 2.4. But more realistically, α should be considered $\log_2 7$ using Strassen's algorithm [9] which is known to give a good performance even for moderate size matrices. Thus the arithmetic complexity of computing $xA^n y'$ is at most min $\{\mathrm{O}(k^a \log n), \mathrm{O}(k^2 n)\}$ based on two ways to compute the product. To get the first bound, we could compute A^n using repeated squaring with the fast matrix multiplication algorithm (such as Strassen's algorithm) as the base case, then pre- and post-multiply by vectors x and y' respectively. To get the latter bound, we multiply the

matrices from left to right. There are $n + 1$ multiplications involved. Each of them (except the last one) requires $O(k^2)$ operations since it involves multiplying a k-vector by a $k \times k$ matrix. The final multiplication is a dot product that takes $O(k)$ steps. The total cost is clearly seen as $O(k^2 n)$. In the following, we will refer to the two algorithms as Type-2 and Type-3, respectively (This is the standard parlance used in computational linear algebra: matrix–vector product is called Type-2 operation while matrix–matrix product is called Type-3 operation.) For a fixed k, it is easy to see that there is an n beyond which the Type-3 algorithm is faster. Asymptotically, if $n = \Omega(k \log k)$ then the Type-3 algorithm is the faster one.

By combining the two algorithms, we can get the following upper-bound.

Theorem 1. *The number of integer multiplications required to compute* $xA^n y'$ (*where* x *is a* 1 *by* k *vector*, A *is a* $k \times k$ *matrix*, *and* c *is a* $k \times 1$ *vector—all with integer entries*) *is* $O(\min_{0 \leqslant r \leqslant \lfloor lg\ n \rfloor} \{rk^\alpha + k^2(n - 2^r)\})$.

This bound is obtained as follows: First compute A^{2^r} by repeated squaring r times. This involves $O(rk^\alpha)$ multiplications. The product $xA^{2^r}A^{n-2^r}y'$ can be computed by left to right matrix multiplication with at most $k^2(n - 2^r)$ integer operations. Since we can choose r, the bound follows.

We have implemented the above algorithm where the matrix entries can be unlimited precision integers. We have been able to solve instances for which DFA's have several thousands of states and n (the length of the string) can also number in the thousands. In fact, it is not hard to see that the size of the alphabet has less impact on the performance of the algorithm than the number of states. The effect of large alphabets is that the entries in A would be large. However, with well-designed unlimited precision arithmetic package, this effect is not as significant as the increase in the size of the DFA.

Since the space complexity is even more critical in the above algorithms than the time complexity, it is worth examining it in some detail. We will examine the two algorithms described above, namely the Type-2 algorithm and Type-3 algorithm. We will assume that the matrix A is small enough (or equivalently, our main memory is large enough) so that it completely fits into the main memory. It is known [9] that Strassen's algorithm can be implemented using $O(k^2)$ additional memory. By implementing repeated squaring bottom-up (non-recursively), we can keep the total additional storage for Type-3 algorithm down to $O(k^2)$. But this bound (with a non-trivial multiplicative constant hidden in the O notation) adds significant overhead to the memory requirement. In contrast, the Type-2 algorithm requires only $O(k)$ additional storage since we only need to store a single vector of size $O(k)$. The memory requirement for the hybrid algorithm is also $O(k^2)$, essentially the same as that for the Type-3 algorithm. Analysis of space complexity becomes more complex when k is too large for the main memory to hold the entire matrix A. In this case, we have to redesign the algorithm using standard techniques like block transfer. The time-space trade-off for large matrix multiplication has been extensively studied and a trade-off for the problem of computing $xA^n y'$ can be modeled on such studies. This problem is beyond the scope of the present paper although it is an important one in view of the fact that many transfer matrices arising in real applications are quite large. (It should be noted that only

recently a detailed understanding of cache efficient algorithms for matrix computations has started to emerge [6].)

A different approach to computing $xA^n y'$ is based on inverting a Toeplitz matrix. This approach is particularly useful in the setting in which the counting problem will be invoked multiple times so that it is worth investing time for pre-processing so that the post-processing for specific instances of n is done fast. The basic idea behind this method is as follows: if $f(n) = xA^n y'$ where A is a $k \times k$ matrix, then $f(n)$ satisfies a linear recurrence equation with constant coefficients [13]. We can compute $f(n)$ for any n by determining the coefficients in the linear recurrence formula. This can be accomplished efficiently by inverting a Toeplitz matrix. The details are as follows: Let $f(n)$ satisfy the linear recurrence equation $f(n) = \Sigma_{j=1}^{k} c_{k-j} f(n-j)$. First we compute $a_i = f(i)$ for $i = 0, 1, \ldots, 2k-1$ using the algorithm described in Theorem 1. Then we determine the vector $c = [c_0 c_1 \ldots c_{k-1}]'$ as: $c = B^{-1}a$ where $a = [a_k a_{k+1} \ldots a_{2k-1}]'$ and B is given by

$$B = \begin{bmatrix} a_0 & a_1 & \ldots & a_{k-1} \\ a_1 & a_2 & \ldots & a_k \\ & & \vdots & \\ a_{k-1} & a_k & \ldots & a_{k-2} \end{bmatrix}.$$

(The algorithm for inversion should be modified to return the largest non-singular matrix that coincides with a left-upper submatrix of B in case B is singular. Any standard algorithm for inversion including Trench's algorithm referred to below can be so modified. We can then use this submatrix instead of B.)

The preprocessing stage can thus be summarized as follows:

(1) Use the algorithm of Theorem 1 to compute $f(i)$ for $i = 0, 1, \ldots, 2k-1$.
(2) Invert the matrix B.
(3) Compute the vector c.

The complexity of the preprocessing step is as follows: The complexity of step 1 is $O(k \min_{0 \leqslant r \leqslant \lfloor lg\ n \rfloor} \{rk^{\alpha} + k^2(n-2^r)\})$. The complexity of Steps 2 and 3 (using Trench's algorithm [23] for inverting a Toeplitz matrix) is $O(k^2)$. Since the exponent of k in the expression for Step 1 is at least 3, replacing Trench's algorithm by Gaussian elimination will not change the asymptotic complexity of the preprocessing stage. But it makes a significant difference in practice since Step 2 is the most time consuming step. Thus Trench's algorithm makes a significant difference in the actual performance of the algorithm.

The obvious algorithm for the postprocessing step involves computing iteratively $f(j)\ j = k+1, \ldots, n$ using the linear recurrence equation. It requires storing the last k computed values of $f(r)$. This algorithm has time complexity $O(kn)$ for computing $f(n)$ for each n. However, faster algorithms are possible. If n is very large, we can do better using Fiduccia's algorithm [4]. The arithmetic complexity of Fiduccia's algorithm is $O(k \log k \log n)$. This algorithm is based on $O(\log n)$ iterations of polynomial multiplication involving two polynomials of degree k. Using Fast Fourier Transform [1], polynomial multiplication over the field of integers can be performed in $O(k \log k)$ steps. The resulting algorithm has time complexity $O(k \log k \log n)$.

By combining the pre- and post-processing steps into a single algorithm, we obtain the following.

Theorem 2. *The number of integer multiplications required to compute xA^ny' (where x is a 1 by k, A is a $k \times k$ matrix, and c is a $k \times 1$ vector—all with integer entries) is at most* $O(k \min_{0 \leqslant r \leqslant \lfloor lg\ n \rfloor} \{2rk^{\alpha} + k^2(n - 2^r)\} + \min\{kn, k \log k \log n\})$.

Before we conclude this section, two additional observations are in order. First, it is more realistic to use *bit* complexity model for this problem since the numbers involved can be rather large. It is not difficult to convert the above upper-bounds to corresponding upper-bounds in the bit complexity model: We have to multiply the arithmetic complexity expression by $M(s)$ where s is the size of the largest number involved and $M(t)$ is the bit complexity of multiplying two t-bit integers. s can be seen to be bounded above by $O(k \log \sigma)$ where σ is the largest row sum of A. The easiest way to see this is as follows: The largest row sum represents an upper-bound on the size of the alphabet over which the automaton is defined. Since $f(n)$ is the number of strings of length n accepted by the automaton, it cannot exceed σ^n. All the intermediate numbers generated obey this bound as well.

Finally, we would like to address the question of whether the algorithms described above are polynomial time algorithms. If k is fixed, and n is the input to the algorithm (which means the input size is $\lg n$), it is easy to see that the arithmetic complexity of the above algorithm can be made a polynomial in $\log n$ by choosing Type-3 algorithm in Step 1 above and by using Fiduccia's algorithm. But the algorithm is still not a polynomial time algorithm since the bit complexity involves n, an exponential term. In fact, the (bit) complexity of this problem is inherently exponential since the output size is exponential in input size. So, a more interesting question is whether there is an ε-approximation algorithm of polynomial time complexity. Here the meaning of ε-approximation is as follows: A is an ε-approximation for B if $|A - B| \leqslant \varepsilon |B|$. First note that for a fixed ε, the leading $\lceil \lg (\frac{1}{\varepsilon}) \rceil$ bits of $f(n)$ together with the binary representation of length of $f(n)$ is an *epsilon*-approximation for $f(n)$. This floating-point representation is of size linear in n. This representation shows that it is possible that such a polynomial time approximation algorithm exists. For a very special case of this problem, such an algorithm has been recently presented by Hirvensalo and Karhumaki [7].

3.1. Theoretical bounds on minimization

The motivation of this study is to explore methods for minimizing the size of a DFA so that the computation of the expression xA^ny' is optimized. Classical linear algebra can assist in determining a theoretical lower bound for the size of A. If A is to represent a DFA, then we are somewhat restricted in that its entries must be non-negative integers. But here we consider the slightly more general case where the entries in the vectors x, y and A belong to the set of non-negative rational numbers. Such matrices can be viewed as representing a "weighted" FA, a model that has been extensively studied [20].

In determining a lower bound, we will make use of the following.

Theorem 3 (*Perron–Frobenius*). *Let A be an $n \times n$ matrix with non-negative entries, then A has a real eigenvalue $\lambda_A \geqslant 0$, which dominates all eigenvalues of A. That is, if λ is any eigenvalue of A, then $|\lambda| \leqslant \lambda_A$. Moreover, at least one right eigenvector and one left*

eigenvector associated with λ_A is semi-positive, and to each eigenvalue λ of A different from λ_A, there corresponds an eigenvector $x \neq 0$ which has at least one negative component.

Proof. See [3]. □

We will refer to λ_A as the *Frobenius root*. The matrix A may have more than one eigenvalue which in absolute value is equal to λ_A, i.e., which lies on the boundary of the circle of radius λ_A about the origin in the complex plane, but if we create the matrix $A' = A + I_n$ where I_n is the $n \times n$ identity matrix, then A' will have a unique maximum eigenvalue, $1+\lambda_A$.

The matrix, A', has an interpretation in terms of DFAs. If A is the transition matrix for a DFA, D, we may introduce a new symbol, x, not belonging to the input alphabet and specify that for each state in the DFA, x determines a transition from the state to itself (a self-loop), then the transition matrix for the new DFA, D', will be A'. We will refer to the new DFA so constructed as the augmented DFA. We have the following lemma.

Lemma 4. *Let D_1 and D_2 be two DFAs which are weakly equivalent. Then the augmentation of D_1 is weakly equivalent to the augmentation of D_2.*

Proof. Let L_1 be the language accepted by D_1 and L_2 by D_2 over input alphabets Σ_1 and Σ_2 respectively. Then for every $k \geqslant 0$ there exists a 1:1 correspondence $\pi_k : L_1 \cap \Sigma_1^k \to L_2 \cap \Sigma_2^k$. Let u_1 be any word accepted by the augmentation of D_1, and let v_1 be obtained by removing all occurrences of x from u_1. Clearly v_1 is accepted by D_1. If $|v_1| = k$, then we may use π_k to obtain a word v_2 accepted by D_2, and by replacing the occurrences of x in v_2 precisely where they occurred in u_1 we obtain a word u_2 accepted by the augmentation of D_2. Since each of these operations is invertible, we have obtained the 1:1 correspondence necessary for weak equivalence. □

The advantage of considering A' over A may be seen in the following. Let n be the size of A'. The *n-dimensional* complex vector space C^n may be decomposed in the traditional way into a direct sum of cyclic subspaces determined by the eigenvalues of A'. The vectors x and y may then each be expressed as linear sums of basis vectors chosen from the cyclic spaces. If there are non-zero components in these sums for the subspace determined by the Frobenius root, then if we iterate multiplication by A', the dominance of this eigenvalue will overtake all other components and the resulting vector will converge to a direction in the eigenspace of $1 + \lambda_A$.

Let D be a DFA that accepts a language L, A its transition matrix, and let L_m be the words in L of length m. We refer to the sequence $\{L_m | \, i \geqslant 0\}$ as the acceptance sets of D. When it exists, we define the *asymptotic acceptance ratio* of D as

$$\rho_D = \lim_{m \to \infty} \frac{|L_{m+1}|}{|L_m|}.$$

If we let x and y be the initial and final vectors, then L_m can be computed by

$$|Lm| = xA^m y'$$

and so

$$\rho'_D = \lim_{m\to\infty} \frac{xA'^{m+1}y'}{xA'^m y'}.$$

We may assume that we have removed all unreachable and non-terminating states from D' and that the resulting DFA has n states. Since the vector space has dimension n and the vectors representing each state of D' form the standard basis and are thus independent, then without using the dominant eigenvector, v_A, it is impossible to span all the state vectors. Thus, at least one state vector contains a non-zero component in v_A. Call one such state q_A, and consider the automaton with q_A as the initial state. Let u_A be the vector associated with q_A. Iterating A' on a vector in each cyclic subspace results in a vector that eventually approaches the eigenvector of that subspace. Thus, iterating A' on u_A will result in a vector approaching v_A, and each non-zero component will grow at a rate asymptotic to $(1 + \lambda_A)^n$. Let q_B be a state corresponding to a non-zero component of v_A. Since q_B terminates, then the set of strings that determine a path that start at q_A, go through q_B, and reach a final state will also grow at the same rate as word length increases. Since some strings beginning at q_0 will determine paths that pass through q_A, then L'_n will also grow at this rate as well, and we have shown that

$$\rho_{D'} = 1 + \lambda_A.$$

Moreover, since for any DFA, E, which is weakly equivalent to D, E' is weakly equivalent to D', and so $\rho_{E'} = \rho_{D'}$. This yields the following theorem.

Theorem 5. *Let D be a DFA with transition matrix A, and assume that λ_A is the root of an irreducible polynomial over the rationals of degree d. Then any DFA weakly equivalent to D must have at least d states.*

Proof. Suppose E is any DFA weakly equivalent to D with transition matrix B. As we have seen above, $\rho_{E'} = (1 + \lambda_A)$, and since λ_A is the root of the characteristic polynomial of B, then B has size at least d. This completes the proof. □

In general, the above theorem provides us with only a crude lower bound for the size of the minimum DFA. In most cases, eigenvectors associated with other eigenvalues also contribute non-zero components to the vectors x and y, and a more detailed analysis of these contributions would be required to determine the actual limit. However, as we shall see in the results below, the lower bound does illustrate the effectiveness of our minimization procedures at least in the cases where matrix size is reasonably small.

4. Weak minimization algorithm

From the discussion above, it is clear that reducing the number of states in the DFA is crucial for the success of a DFA based approach for counting problems. The standard optimizations we should first attempt are: (1) remove useless states and (2) perform DFA

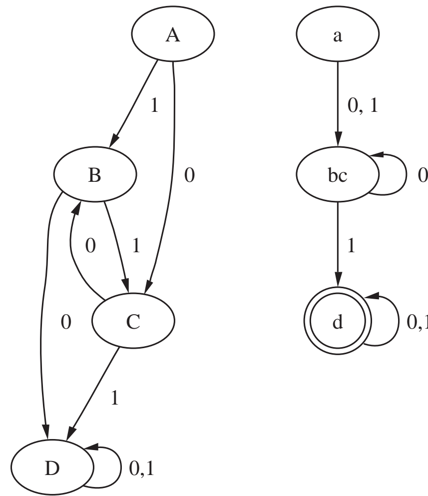

Fig. 6. M(on the left) is strongly minimal, but not weakly minimal as is M' (on the right).

minimization (in the classical sense). The following example illustrates how we can optimize beyond standard DFA minimization.

Example. Consider the DFA M shown in Fig. 6. It is a minimal DFA, but it can be shown below that M' is a smaller DFA that is weakly equivalent to M.

If we switch the transitions on input symbols 0 and 1 from state B (an operation that preserves weak equivalence) in M, the resulting DFA is not minimal and by (classical) minimization of this DFA, we get M'.

The above idea leads to the following concepts and problems.

Weak equivalence problem
Input: Two DFAs M_1 and M_2. *Output*: *yes* if M_1 and M_2 are weakly equivalent, *no* else.

Weak Minimization Problem
Input: A DFA M. *Output:* A DFA M' with as few states as possible such that M and M' are weakly equivalent.

4.1. Algorithm for weak equivalence

An algorithm for weak equivalence follows directly from the algorithm for the equivalence of probabilistic automata due to Tzeng [24]. The reason is as follows: Two probabilistic automata are equivalent if their acceptance probabilities are the same for every string. Tzeng presents a polynomial time algorithm for this problem. We can directly translate this algorithm into a weak equivalence testing algorithm as follows: Let M_1 and M_2 be two DFA's whose weak equivalence we want to test. We can convert each of them to a probabilistic automaton over a unary alphabet by assigning a transition probability $\frac{1}{\sigma}$ for

each transition in the given DFA where σ is the size of the input alphabet. Then, it is easy to see that M_1 and M_2 are weakly equivalent if and only if the corresponding probabilistic automata are equivalent. Tzeng shows an upper bound of $O(n^4)$ for his algorithm where $n = |M_1| + |M_2|$. The same bound thus holds for weak equivalence as well. We present a faster algorithm below based on the following lemma. For a DFA, M, and a state, q, we define $f_M(q, n)$ as the number of words of length n that determine paths from q to a final state. In the case, where q is the initial state we simplify this to $f_M(n)$.

Lemma 6. *Let $M_1 = \langle Q_1, \Sigma_1, \delta_1, s_1, F_1\rangle$ and $M_2 = \langle Q_2, \Sigma_2, \delta_2, s_2, F_2\rangle$ be two DFAs. If $f_{M_1}(n) = f_{M_2}(n)$ for all $n = 0, 1, 2, \ldots, 2(|Q_1| + |Q_2| - 1)$, then M_1 and M_2 are weakly equivalent.*

Proof. Can be found in [17,22]. □

This lemma leads to the following algorithm for weak equivalence testing: Given two DFAs M_1 and M_2 as above, we compute for each $j = 1, 2, \ldots, (|Q_1| + |Q_2| - 1)$, the number of strings of length j using the Type-3 algorithm presented in the last section. If they agree for every computed j, then they are weakly equivalent, else they are not. It is easy to see that this algorithm has complexity $O(n^3 \log n)$. It is an interesting problem to find a faster algorithm for weak equivalence.

4.2. *Algorithm for weak minimization*

The basic idea behind this algorithm is as follows. Recall the Myhill–Nerode theorem [8] that states that a DFA M is minimal if and only if (a) M has no unreachable states (from starting state) and (b) for any two states p and q, there is some string x such that exactly one of $\delta(p, x)$, $\delta(q, x)$ is in F. A simple (strong) minimization algorithm is: Create an initial partition of vertices into $S_1 = F$ and $S_2 = Q - F$. If there is a pair of states $p, q \in S_i$, and an input symbol $a \in \Sigma$ such that $\delta(p, x)$ and $\delta(q, x)$ are in different S_j's, split S_i such that all states that go to the same S_k on input a are in the same partition of S_i. This process of splitting continues until no split is possible. At this stage, each group contains states that are equivalent to each other and hence they can be merged to form the minimal DFA.

We use a similar idea. After k steps, two states belong to the same class in the partition if and only if the number of strings of length $i = 0, 1, 2, \ldots, k$ that are accepted by the DFA starting from each of the states is the same. In the next step, for each state in an equivalence class, we compute the number of strings of length $k+1$ that determine a path from that state to a final state. The partition is then refined by subdividing the class into those subsets in which the number of strings remain the same. Define two states p and q as weakly equivalent if for all k, the number of accepting strings of length k starting at p is the same as the number of accepting strings of length k starting at q.

Lemma 7. *Let M be a DFA and let $P = \{p_1, p_2, \ldots, p_r\}$ be r states that are weakly equivalent. Then, there exists a DFA, M', that is weakly equivalent to M obtained by merging states in P. Moreover, if q_1 and q_2 are two states not in P that are weakly equivalent in M, then they remain weakly equivalent in M'.*

Proof. Merging is done as follows. First, label each arc in M with a unique label. (Note that this does not change weak equivalence.) Choose one member of P (say, p_1) as the representative of the set. The outgoing arcs from p_1 remain the same in M'. For each p_i, $i = 2, \ldots, r$, redirect all incoming arcs to p_i that originate outside of P so that they now go into p_1 instead. In addition, redirect all incoming arcs to p_i that originate at p_1 back to p_1 itself. If q_0 is in P, then the representative of P becomes the initial state. If P is contained in F, then the representative of P is a final state.

We now demonstrate a 1:1 correspondence between paths in M and paths in M' that reach a final state. First consider a path in M from initial state, q_0, to a final state. We will prove by induction that there is a 1:1 correspondence between terminating paths in M that pass through P k times, and terminating paths in M' that pass through p_1 k times.

For $k = 0$, we note that if the path in M does not pass through a member of P, then it remains unchanged in M'.

For $k = 1$, the path can be segmented into two pieces: the segment leading from q_0 to some p_i belonging to P and the path from p_i to a final state. In the case that q_0 is p_i, i.e. the initial segment is the empty path, then the second segment, the path from q_0 to final state corresponds to the path from initial state to final state in M'.

Now suppose the path from q_0 to p_i is not empty. Suppose the last arc on this path is $\langle q_j, a_j, p_i \rangle$. Since none of the states in the initial segment of the path are in P then the path up to the last arc remains unchanged in M', and the last arc in m has been replaced in M' by $\langle q_j, a_j, p_1 \rangle$. Since the labels on arcs in M are unique, the replacement determines a unique path to p_1 in M'. Now assume that the second segment of the path has length m. (Note that m could be 0 if p_i is final.) Since p_i and p_1 are weakly equivalent, then there is a 1:1 correspondence between paths of length m from p_i to a final state and paths of length m from p_1 to a final state in M. In M' we may replace the second segment with its corresponding path in M from p_1. This yields a 1:1 correspondence for $k = 1$.

For $k > 1$, we may proceed inductively. We divide paths in M passing through P $k + 1$ times into three segments: a path through P k times ending in p_i, a path from p_i to p_j with no intermediate visits to P, and a terminating path from p_j. To find its corresponding path in M', we first remove the middle segment of the path (the piece from p_i to p_j). Since the first segment ends in p_i and the last begins in p_j and p_i and p_j are equivalent, we replace the final segment with its corresponding path beginning at p_i and adjoin this to the initial segment to obtain a path that passes through P k times. Inductively we find the path in M' corresponding to it and then reinsert the path from p_i to p_j changing p_i and p_j to p_1 in M'.

Conversely, the paths in M' can be mapped back into paths in M by examining the labels on the arc to each visit to p_1. Since these labels are unique, then the map back to M can be determined uniquely.

The last statement in the lemma may also be proved inductively by the number of visits to P. This completes the proof. $\square$

The idea behind the algorithm is to compute in an incremental manner the number of accepting strings of length k starting from each state and maintain an equivalence class that includes all the states that have the same count for all $j = 0, 1, \ldots, k$. In other words, two states are k-weakly equivalent if $f_M(p, j) = f_M(q, j)$ for all $j \leqslant k$. We refine the

partition during the next iteration based on $(k+1)$-weak equivalence, just as in the strong-minimization algorithm. The algorithm terminates after $2|Q|-1$ iterations. Now we can merge the elements of an equivalence class into a single state. A more formal description of the algorithm follows.

Algorithm WeakMinimizeDFA
Input: DFA $M = < Q, A, q_0, F >$
 //Assume that M is strongly minimal DFA with useless states removed.
 //A is the transition matrix for M
1. $NF = Q - F$ // non-final states
Partition = $\{F, NF\}$
len = 0 // string length
$v_P = < 1, 0 >$
// v_P represents the number of paths from each member of the partition to
// final state. v_P will grow in length as the Partition is refined.
p = size(Partition)
2. while ($len \leqslant 2 * |Q| - 1$)
 { Refinement = {} // Initially Refinement is empty
 $refine_v = <>$ // Initially $refine_v$ is an empty vector
 for every P_i in Partition
 {//create vector s_i of size $= |P_i|$
 for every state q in P_i
 //create vector v_q of size p where $v_q[k]$ is the number of arcs from q to P_k.
 $v_q[k]$ = sum of the elements of matrix A in the row corresponding to q
 and the columns corresponding to elements of P_k
 s_i[q] = v_q . v_P // s_i[q] is the number of paths of length $len + 1$ from q to F
 if (not all values of s_i are the same)
 {refine P_i by grouping elements with like values in s_i;
 Adjoin refinement of P_i to Refinement
 Adjoin corresponding values of s_i to $refine_v$
 }
 else
 { Adjoin P_i to Refinement
 Adjoin unique value of s_i to $refine_v$
 }
 Partition = Refinement
 p = size of Partition
 $v_P = refine_v$
 len = len + 1
 }
3. //Construct M' as follows.
For each $P_i = \{p_1, p_2, \ldots, p_r\}$ in Partition, choose one member (say, p_1)
as the representative of the set.
//The outgoing arcs from p_1 remain the same in M'.
 For each p_j, $i = 2 \ldots r$

remove the row of A corresponding to p_j from A // This removes p_j from the DFA
add the column of A corresponding to p_j to the column corresponding to p_1.
//This redirects all incoming arcs to p_j that originate outside of P_i
// so that they now go into p_1 instead.
//The arcs that originate at p_1 and terminate at p_j are now self-loops back to p_1
remove the column of A corresponding to p_j from A

If q_0 is in P_i, then the representative of P_i becomes the initial state.
If the representative of P_i belongs to F, then it remains a final state.

4.3. Correctness of the algorithm and its time complexity

We claim that M' is weakly equivalent to M. This can be shown as a direct consequence of the above lemma since we are computing for all states p, the set of strings of length k accepted starting from p. If the two vectors are identical after k iterations, it follows from the above theorem that the two states are weakly equivalent. Thus, we can combine them into a single state as done in Step 3. It is clear that M' has no more states than M and in practice, the algorithm reduces the number of states significantly. However, as shown below, the algorithm does not always produce a weakly minimal DFA.

Example. Minimal DFA for $2 \times n$ self-avoiding walk.

In an earlier section, we described a DFA for the $2 \times n$ self-avoiding walk problem. It had 14 states. The weak minimization algorithm reduces the size of the DFA to 9 states. The characteristic polynomial for the transition matrix is $p(x) = x^2(x-1)^3(x+1)^2(x^2-x-1)$ with dominant eigenvalue $= \frac{(1+\sqrt{5})}{2}$. As it turns out, eigenvectors for the non-dominant eigenvalues also contribute to the computation of the number of strings of length n that are accepted. A full analysis of eigenvectors allows for the construction of the optimally minimal DFA. If we consider only strings of length 2 or more (the nilpotent portion of the matrix has dimension 2 because of the multiplicity of the 0 eigenvalue), then we find that a matrix of size 6, shown below, with initial vector $x = [1\,0\,0\,0\,1\,1]$ and $y = [2\,6\,0\,-3\,-1\,0]$ is a transfer matrix of a DFA that is weakly equivalent to the above DFA.

$$A = \begin{bmatrix} 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 \end{bmatrix}.$$

A Transition matrix of minimal size for the $2 \times n$ self-avoiding walk.

Time complexity. Let k be the number of states in the DFA M. Step 1 requires $O(k)$ steps to partition the states into final and non-final subsets. The algorithm then performs $2k$-1 iterations of Step 2. In each iteration, we consider each state of M once, as we construct the s_i vectors. Each term in the vector is computed by forming the dot product of the vector v_q with the vector v where each vector has size, p, equal to the size of the partition. The dot product requires p multiplications and $p-1$ additions, i.e. $2p-1$ arithmetic steps. Since

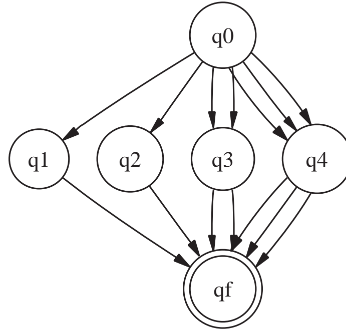

Fig. 7. States q_1 and q_2 are right weak equivalent. Their merger contains two incoming arcs and is thus becomes left weak equivalent to q_3. Merging these on the left creates a single state with three outgoing arcs which is now right equivalent to q_4. The last reduction produces a three-state DFA with 5 edges initiating at q_0 and 3 terminating at q_f.

p is bound by k, each iteration requires $\mathrm{O}(k^2)$ steps. Grouping the elements of each vector s_i by like value is $\mathrm{O}(k^2)$ in total. Thus, Step 2 is $\mathrm{O}(k^3)$. Finally, because Step 3 involves adding columns of transition matrix together (as we merge states) and eliminating rows, the cost of Step 3 is easily seen to be $\mathrm{O}(k^2)$. Thus we have.

Theorem 8. *The time complexity of constructing a reduced weakly equivalent DFA to a given DFA with k using the algorithm presented above is* $\mathrm{O}(k^3)$.

4.4. Left weak equivalence

The definition of weak equivalence concerns the number of paths from each state to final states. Symmetrically, one could as easily consider the number of paths from the initial state to any given state and define two states as *left* weakly equivalent if this number is the same for all string lengths. (To distinguish this new definition from our preceding one, we will refer to the former as *right* weak equivalence.) Except in one case, the algorithms and proofs carry over merely by considering the transpose of the transition matrix. The one exception is that in the case of left equivalence, it is not always possible to merge equivalence classes consisting of final states. A DFA and its reverse DFA is symmetric in every regard except for the uniqueness of the initial state in the former and the multiplicity in the latter. However, if we exclude classes of final states, we may utilize left weak equivalence to reduce the size of the DFA.

In fact, after performing a reduction based on right weak equivalence it is sometimes the case that the resulting DFA now contains left weak equivalent states that can allow a further reduction. Moreover, it is a simple matter to demonstrate that given any positive integer k there are DFAs for which a sequence of k alternating right minimizations and left minimizations may be performed with a reduction at each step. Fig. 7 illustrates one such DFA in which right, left, and right minimizations reduce the DFA.

A feed-forward network such as the one in Fig. 7 also illustrates that a sequence of alternating right and left reductions does not always yield the absolute minimal DFA. For

Table 1
Minimal DFA vs. Weak reduced DFA

k = Column width	Strong-minimal DFA	Weak-reduced DFA
5	20	10
6	20	14
7	70	32
8	70	43
9	252	114
10	252	142
11	924	418
12	924	494
13	3432	1646
14	3432	1780
15	12870	6272
16	12870	6563

example, consider a DFA with four states q_0, q_1, q_2, and q_3 where q_0 is the initial state and q_3 is the final state and in which there are 9 arcs from q_0 to q_1 and 1 arc from q_1 to q_3 and 2 arcs from q_0 to q_2 and 3 arcs from q_2 to q_3. Minimization, on the left or right, does not reduce this DFA though its behavior is identical to that in Fig. 7.

4.5. *Implementation results*

We have implemented the above algorithm and have tested it on several examples including the tiling DFA's described in Section 2.1. In most of the examples, we found moderate to significant reduction in the number of states when we applied the algorithm on strong-minimized DFA's.

The following table (Table 1) shows the size of the strongly minimized DFA from the DFA generated the by program of Section 2.1 and the size of the weak-reduced DFA (based on the algorithm presented above) for various values of k.

The above results indicate that the reduction in the number of states for this family of DFA's by applying the weak minimization algorithm is nearly by a factor of 1/2. In addition in the case $k = 5$, a reduction using left weak minimization further reduces the number of states to 8, the optimum size as determined by examining eigenvalues. (Left minimization did not further reduce the DFA for $k > 5$.)

4.6. *Extension of the algorithm to unambiguous NFA's*

Recall that the original goal of this paper is to show that many counting problems can be solved in a unified manner using a DFA model. It is easy to see that this approach works even if the strings that we want to count can be accepted by a NFA so long as it is unambiguous. The matrix power formula $xA^n y'$ for the number of strings of length n also holds for unambiguous NFAs. This fact was implicitly shown by Stearns and Hunt [22]. Our weak-minimization algorithm works for unambiguous NFA as well. Although converting a DFA to a minimal equivalent unambiguous NFA is known to be NP-complete

[10], the weak minimization may be an effective way to reduce the number of states of an unambiguous NFA when the application involves counting, and not membership.

5. Summary of contributions and directions for further work

We have accomplished the following goals in this paper: (1) We showed that a number of counting problems can be solved in a unified manner using a DFA based approach. As examples, we showed that the DFA based approach can be used to count the number of simple paths in a grid graph and the number of ways to tile a lattice grid using dominoes. (2) The problem of evaluating a matrix power formula has lead to the development of a hybrid algorithm based on a number of optimizations including the use of Trench's algorithm for inverting a Toeplitz matrix and Fiduccia's algorithm for solving a linear recurrence formula. (3) Further optimization issues (with the aim of reducing k, the size of the transfer matrix) led us to propose new notions of weak equivalence and weak minimization of DFA's. (4) Finally, we designed and implemented an efficient algorithm for the weak minimization problem.

This study has raised several interesting practical and theoretical problems. Here is a short list of them: (1) Determine for which classes of automata the algorithm presented in Section 4 always finds a weakly minimal DFA. (2) Develop a software design framework that converts a DFA specification into a DFA realization. Implicit representations and other compact representations can be pursued in counting applications to surmount the storage requirements of transfer matrix. (3) It is obvious that there are non-regular languages that have the same counting function as regular languages. A systematic identification of such languages will extend the scope of counting problems that can be solved using the transfer matrix approach.

Acknowledgements

Some of the programs described above were implemented by Xiaoming Lu, a graduate student at Sonoma State University.

References

[1] A. Aho, J. Hopcroft, J. Ullman, Design and Analysis of Computer Algorithms, Addison-Wesley, Reading, MA, 1973.
[2] D. Coppersmith, S. Winograd, Matrix multiplication via arithmetic progressions, J. Symbolic Comput. 9 (3) (1990) 251–280.
[3] G. Debreu, I.N. Herstein, Nonnegative square matrices, J. Econometric Soc. 21 (4) (1953) 597–606.
[4] C. Fiduccia, An efficient formula for linear recurrences, SIAM J. Comput. 14 (1) (1985) 106–112.
[6] P.J. Hanlon, D. Chung, S. Chatterjee, D. Genius, A. Lebeck, E. Parker, The combinatorics of cache misses during matrix multiplication, J. Comput. System Sci. 63 (1) (2001) 80–126.
[7] M. Hirvensalo, J. Karhumaki, Computing partial information out of intractable one—the first digit of 2^n at base 3 as an example, W. Rytter (Ed.), Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, Vol. 2420, Springer, Berlin, 2003.

[8] J. Hopcroft, J. Ullman, Introduction to Automata, Languages and Theory of Computation, Addison-Wesley, Reading, MA, 1979.
[9] S. Huss-Lederman, E. Jacobson, A. Tsao, T. Turnbull, J. Johnson, Implementation of Strassen's algorithm for matrix multiplication, ACM/IEEE Conf. on Supercomputing, 1996.
[10] T. Jiang, B. Ravikumar, Minimal NFA problems are hard, SIAM J. Comput. 22 (6) (1993) 1117–1141.
[11] D. Klarner, J. Pollack, Domino tilings with rectangles of fixed width, Discrete Math. 32 (1980) 45–52.
[12] C.L. Liu, Introduction to Combinatorial Mathematics, McGraw-Hill, New York, 1968.
[13] L. Lovasz, C. Kenyon, Lecture Notes in Algorithmic Discrete Mathematics, Princeton University, Technical Report, TR 251-90, 1990.
[14] N. Madras, G. Slade, The Self-Avoiding Walk, Birkhauser, Boston, MA, 1993.
[15] M. Ogihara, S. Toda, The complexity of computing the number of self-avoiding walks, J. Sgall et al. (Ed.), Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, Vol. 2136, Springer, Berlin, 2001, pp. 585–597.
[16] L. Pachter, Combinatorial approaches and conjectures for 2-divisibility problems concerning domino tilings of polyominoes, Electron. J. Combin. 4 (1997) R29.
[17] A. Paz, Introduction to Probabilistic Automata, Academic Press, New York, 1971.
[18] A. Ponitz, P. Tittmann, Improved upper bounds for self-avoiding walks in Zd, Electron. J. Combin. 7 (2000) R21.
[19] J. Propp, A reciprocity theorem for domino tilings, Electron. J. Combin. 8 (2001) R18.
[20] A. Salomaa, M. Soittola, Automata-Theoretic Aspects of Formal Power Series, Springer, Berlin, 1978.
[21] R. Stanley, On dimer coverings of rectangles of fixed width, Discrete Appl. Math. 12 (1985) 81–87.
[22] R. Stearns, H. Hunt, On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata, SIAM J. Comput. 14 (1985) 598–611.
[23] W.F. Trench, An algorithm for the inversion of finite Toeplitz matrices, J. SIAM 12 (3) (1964) 715–722.
[24] W. Tzeng, The equivalence and learning of probabilistic automata, 30th Annu. Symp. on Foundations of Computer Science, 1989, pp. 268–273.
[25] L. Valiant, The complexity of enumeration and reliability problems, SIAM J. Comput. 8 (3) (1979) 410–421.
[26] H. Wilf, The problem of kings, Electron. J. Combin. 2 (1995) R3.