Derandomization, witnesses for Boolean matrix multiplication and construction of perfect hash functions[*]

Noga Alon [†] Moni Naor [‡]

To appear in Algorithmica, final version

Abstract

Small sample spaces with almost independent random variables are applied to design efficient sequential deterministic algorithms for two problems. The first algorithm, motivated by the attempt to design efficient algorithms for the All Pairs Shortest Path problem using fast matrix multiplication, solves the problem of computing *witnesses* for the Boolean product of two matrices. That is, if A and B are two n by n matrices, and $C = AB$ is their Boolean product, the algorithm finds for every entry $C_{ij} = 1$ a witness: an index k so that $A_{ik} = B_{kj} = 1$. Its running time exceeds that of computing the product of two n by n matrices with small integer entries by a polylogarithmic factor. The second algorithm is a nearly linear time deterministic procedure for constructing a perfect hash function for a given n-subset of $\{1, \ldots, m\}$.

[*]Part of this paper was presented at the IEEE 33rd Symp. on Foundations of Computer Science

[†]Incumbent of the Florence and Ted Baumritter Combinatorics and Computer Science Chair, Department of Mathematics, Raymond and Beverly Sackler Faculty of Exact Sciences, Tel Aviv University, Tel Aviv, Israel. Research supported in part by a USA Israeli BSF grant and by the Fund for Basic Research administered by the Israel Academy of Sciences. e-mail: noga@math.tau.ac.il.

[‡]Incumbent of the Morris and Rose Goldman Career Development Chair, Department of Applied Math and Computer Science, Weizmann Institute, Rehovot 76100, Israel. Supported by an Alon Fellowship and by a grant from the Israel Science Foundation administered by the Israeli Academy of Sciences. e-mail: naor@wisdom.weizmann.ac.il. Some of this work was done while the author was with the IBM Almaden Research Center.

1 Introduction

In this paper we show how to make two very efficient probabilistic algorithms deterministic at a relatively small degradation in their performance. The random choices made by a probabilistic algorithm define naturally a probability space where each choice corresponds to a random variable. In order to remove randomness from an algorithm one should come up with a way of finding deterministically a successful assignment to these choices.

One approach for achieving it, known as the *method of conditional probabilities*, is to search the probability space for a good choice by bisecting the probability space at every iteration by fixing an additional choice. The value of the next bit is chosen according to some estimator function that should approximate the probability of success given the choices fixed so far. The number of steps is therefore proportional to the number of choices made by the probabilistic algorithm and each step involves evaluating an estimator which usually takes time proportional to the input size. Thus, the cost of derandomization though polynomial, can considerably increase the complexity of the algorithm. This approach is taken in [29], [25], (cf., also [6].)

A different approach for finding a good point is to show that the random choices made need not be fully independent, i.e. even if some limited form of independence is obeyed, then the algorithm is successful. A smaller probability space where the random choices obey this limited independence is constructed. If this space is exhaustively searched, then a good point is found. The complexity is increased by a factor proportional to the size of the space. The size of the space is usually some polynomial in the input size. Thus again this approach suffers from considerable increase in time. This approach is taken in [17], [1], [15] using probability spaces that are k-wise independent, and in [5], [23] using small bias probability spaces and almost k-wise independence (see definition below in subsection 1.3).

Our goal in this work is to use these methods without incurring a significant penalty in the run time. We exploit the fact that very small (polylogarithmic) probability spaces exist if one is willing to live with very limited independence. This form of independence is usually too limited to be applicable directly for replacing the random choices in a probabilistic algorithm. Our tactic will be to divide the random choices into a small (logarithmic or polylogarithmic) number of sets of random variables with complete independence between the sets. However within each set we will require only very limited independence. The search algorithm finds a good assignment by fixing the sets one by one. At every iteration all the points of a small probability space corresponding to the current set of random variables is examined and the one that maximizes some estimator function of the probability of success is chosen. Since we have only a few sets of random variables and since each probability space is small the total work is increased by only a polylogarithmic factor.

We note that the two approaches described above had been combined in a different way previously in [18, 7, 22]. The random variables used there were k-wise independent resulting in a probability space of size $O(n^k)$. This probability space was then searched using an estimator function in $O(k \log n)$ steps. This method does not seem applicable for the problems considered here, since we could not come up with appropriate estimator functions that were efficiently computable.

In the following two subsections we describe the two algorithmic problems for which applying the above mentioned method yields efficient deterministic algorithms: the computation of Boolean matrix multiplication with witnesses and the (deterministic) construction of perfect hash functions. Although the two problems are not related, the algorithms we suggest for both are similar, and are based on the same approach outlined above. In subsection 1.3 we review the probability spaces that have the independence properties used for both applications.

1.1 Witnesses for matrix multiplication

Consider a Boolean matrix multiplication: $C = AB$, $C_{ij} = \bigvee_{k=1}^{n}(A_{ik} \wedge B_{kj})$. The n^3 time method that evaluates these expressions gives for every i, j for which $C_{ij} = 1$ all the k's for which $A_{ik} = B_{kj} = 1$. The subcubic methods on the other hand (see, e.g., [8]) consider A and B as matrices of integers and do not provide any of these k's. We call a k such that $A_{ik} = B_{kj} = 1$ a *witness* (for the fact that $C_{ij} = 1$). We want to compute in addition to the matrix C a matrix of witnesses. When there is more than one witness for a given i and j we are satisfied with one such witness.

We use $O(n^\omega)$ to denote the running time of some subcubic algorithm for Boolean matrix multiplication. Our algorithm for this problem can be derived from any such algorithm yielding a corresponding time bound as a function of w. The best asymptotic bound known at present is the one with the exponent $\omega < 2.376$ and is due to Coppersmith and Winograd [8].

For two functions $f(n)$ and $g(n)$ we let $g(n) = \tilde{O}(f(n))$ denote the statement that $g(n)$ is $O(f(n)(\log n)^{O(1)})$.

Several researchers (see, e.g., [28], [4]) observed that there is a simple randomized algorithm that computes witnesses in $\tilde{O}(n^\omega)$ time. In Section 2 we describe a **deterministic** algorithm for computing the witnesses in $\tilde{O}(n^\omega)$ time. It is essentially a derandomization of a modified version of the simple randomized algorithm using the approach outlined in the Introduction, i.e. the combination of small sample spaces and the method of conditional probabilities. A different, more complicated algorithm for this problem, whose running time is slightly inferior, i.e. not $\tilde{O}(n^\omega)$ (but is also $O(n^{\omega+o(1)})$), has been found by Galil and Margalit [20], [14].

The main motivation for studying the computation of witnesses for Boolean matrix multiplication is the observation of Galil and Margalit that this problem is crucial for the design of efficient algorithms for the All-Pairs-Shortest-Path problem for graphs with small integer weights which are based on fast matrix multiplication. Efficient algorithms for computing the *distances* in this way were initiated in [3] and improved (for some special cases) in [13], [28]. The attempt to extend this method for computing the shortest paths as well leads naturally to the above problem, which already found other (related) applications as well. See [4, 14, 20, 28] for more details.

1.2 Perfect hash functions

For a set $S \subset \{1, \ldots, m\}$ a *perfect hash function* is a mapping $h : \{1, \ldots, m\} \mapsto \{1, \ldots, n\}$ which is 1-1 on S. H is an *(m, n, k)-family of perfect hash functions* if $\forall S \subset \{1, \ldots, m\}$ of size k there is an $h \in H$ that is perfect for S. We will be interested mainly in the case $k = n$.

The requirements from a perfect hash function are

- Succinct representation - the mapping h can be described by a relatively small number of bits.
- Efficient evaluation - given a value $x \in \{1, \ldots, m\}$ and the description of h, there should be an efficient method of computing $h(x)$.
- An efficient construction - given S there should be an efficient way of finding $h \in H$ that is perfect for S

Perfect hash functions have been investigated extensively (see e.g. [9, 10, 11, 12, 16, 19, 21, 26, 27, 30]). It is known (and not too difficult to show, see [11], [16], [24]) that the minimum possible number of bits required to represent such a mapping is $\Theta(n + \log\log m)$ for all $m \geq 2n$.

Fredman, Komlós and Szemerédi [12] developed a method for constructing perfect hash functions. Given a set S, their method can supply a mapping with the required properties in almost linear expected *randomized* running time. Deterministically, however, they only describe a variant of their algorithm that works in worst-case running time $O(n^3 \log m)$. In Section 3 we describe a construction of perfect hash functions and a *deterministic* algorithm that for a given S in time $O(n \log m \log^4 n)$ finds a mapping with the above properties. Note that the size of the input is $\Theta(n \log m)$ and hence this algorithm is optimal, up to a polylogarithmic factor. In case m is polynomial in n, the representation of the mapping h constructed requires $O(n)$ bits. Given x computing $h(x)$ takes $O(1)$ operations. In the general case, the size of the mapping h requires $O(n + \log n \log\log m)$ bits. The time to evaluate $h(x)$ depends on the computational model (i.e. what operations can be performed in one step): it is either $O(\frac{\log m}{\log n})$ in a weak model or $O(1)$ in a strong one (see more about the model in Section 3).

1.3 Small bias probability spaces

Let Ω be a probability space with n random variables $x_1, x_2, \ldots x_n$. We say that Ω is a *c-wise ϵ-bias* probability space if for any nonempty subset S of $x_1, x_2, \ldots x_n$ of size at most c we have

$$|Prob[\bigoplus_{i \in S} x_i = 0] - Prob[\bigoplus_{i \in S} x_i = 1]| < \epsilon.$$

The property of a c-wise ϵ-bias probability space that we use is that for any subset S of $x_1, x_2, \ldots x_n$ of size $i \leq c$ the probability that the random variables of S attain a certain configuration deviates from $1/2^i$ by at most ϵ. Therefore c-wise ϵ-bias probability spaces are described as almost c-wise independent.

The known constructions of these probability spaces are of size polynomial in $c, 1/\epsilon$ and $\log n$ (often described by saying that the number of random bits required to sample from them is $O(\log 1/\epsilon + \log c + \log\log n)$). Therefore if $1/\epsilon$ is logarithmic in n and c is at most logarithmic in n the size of the probability space is still polylogarithmic in n. To be more precise, the construction of [23], as optimized in [2], yields a probability space of size $O(\frac{c \log n}{\epsilon^3})$ and the ones in [5] yield probability spaces of size $O(\frac{c^2 \log^2 n}{\epsilon^2})$.

2 Boolean matrix multiplication with witnesses

All the matrices in this section are n by n matrices, unless otherwise specified. If M is such a matrix, we let M_{ij} denote the entry in its ith row and jth column. Let A and B be two matrices with $\{0,1\}$ entries, and let C be their product over the integers. Our objective is to find witnesses for all the positive entries of C, i.e., for each entry $C_{ij} > 0$ of C we wish to find a k such that $A_{ik} = B_{kj} = 1$. This is clearly equivalent to the problem of finding witnesses in the Boolean case. As observed by several researchers there is a simple randomized algorithm that solves this problem in expected running time $\tilde{O}(n^{\omega})$. Here we consider *deterministic* algorithms for the problem. Our algorithm, described in the next two subsections, is a derandomized version of a modification of the simple randomized solution. Its running time is $\tilde{O}(n^{\omega})$. The analysis of the algorithm is presented in subsection 2.3, where it is also shown how to replace the probabilistic steps with deterministic ones.

2.1 Outline and intuition of the algorithm

The starting observation is that finding witnesses for entries which are 1 is easy: if E and F are two matrices with $\{0,1\}$ entries and $G = EF$ is the result of their multiplication over the integers, then one multiplication of matrices with entries of size at most n suffices for finding witnesses for all the entries of G which are precisely 1. Indeed, simply replace every 1-entry in the kth row of F by k (for all $1 \le k \le n$) to get a matrix F' and compute $G' = EF'$. Now observe that if $G_{ij} = 1$ and $G'_{ij} = k$ then k is a witness for G_{ij}.

The idea (of both the randomized and deterministic algorithms) is to dilute F gradually, thus making the entries of G go down to 0, however not before passing through 1. Therefore if $G_{ij} = \ell$ and every entry in F is made zero with probability roughly $1/\ell$, then G_{ij} becomes 1 with probability bounded away from zero. A way of achieving it simultaneously for all entries is to work in phases, where in each phase every '1' entry of F is zeroed with probability 1/2. At each phase also find witnesses for all entries of G that became 1. If $G_{ij} = \ell$ then for the (i,j) entry of EF after $\log \ell$ such phases there is a constant probability of turning G_{ij} into a 1 and thus enabling the discovery of a witness. By repeating this process $\log n$ times, there is a high probability of discovering all the witnesses.

The choices we must make in the execution of the algorithm is which entries of F to zero at what phase. In the randomized algorithm all choices are independent, and thus the size of the probability is exponential in $O(n \log^2 n)$. In order to remove the randomness from the witness finding algorithm we follow the paradigm outlined in the Introduction. Random choices corresponding to different phases remain independent, however, the choices made in a phase will be highly dependent. We must also find a good estimate of progress. The key point is that for our estimate of progress it is sufficient that the choices within a phase be made according to a c-wise ϵ-bias sample space for c and $1/\epsilon$ that are logarithmic in n. Our notion of progress is defined by two "contradicting" conditions: first the total sum of entries in G must go down significantly at every round (by at least a constant fraction). This implies that in $O(\log n)$ rounds we get that G vanishes. The second condition is that we do not lose too many entries of G, where by lose we mean that they go from a large value to 0 without passing through 1.

The second condition turns out to be too strong. We relax it by specifying some bound c (not coincidently, the same c as above) such that we would like every entry to pass through the range $\{1,\ldots,c\}$ before vanishing. We show how to find a witness in this case as well. The fraction of entries that disobey the second condition should be small enough to assure that at least a constant fraction of the entries do not skip the desired range. The set of good assignments to the choices of a phase is obtained by an exhaustive search among all the sample space for an assignment that progresses nicely. This is repeated for all phases and the whole process is repeated $O(\log n)$ times until all entries have a witness.

2.2 Detailed description of the algorithm

Define $c = \lceil \log\log n + 9 \rceil$ and $\alpha = \frac{8}{2^c}$. For two matrices E and F with $\{0,1\}$ entries define $G = E \wedge F$ by $G_{ij} = E_{ij} \wedge F_{ij}$.

Besides A,B and $C = AB$ our algorithm employs two sequences of $\{0,1\}$ matrices: $R^1, R^2 \ldots, R^{t+1}$ and $D^1, D^2, \ldots D^{t+1}$ where $t = \lceil 1 + 3\log_{4/3} n \rceil$. For matrices R and R' we say that R' is a *dilution* of R if for every $1 \le j,k \le n$ we have $R_{j,k} \ge R'_{j,k}$. The sequence $R^1, R^2 \ldots R^{t+1}$ is monotonically decreasing, i.e. for every $1 \le i \le t$ R^{i+1} is a dilution of R^i. We now describe the algorithm; the way to perform steps 4b and 4c will be described later. The definition of a good dilution is given below.

- While not all witnesses are known
  1. Let L be the set of all positive entries of C for which there are no known witnesses.
  2. Let R^1 be the all 1 matrix.
  3. Let $D^1 \leftarrow A \cdot (B \wedge R^1)$
  4. For $i = 1$ to $t = \lceil 1 + 3\log_{4/3} n \rceil$, Perform the following:
     (a) Let L' be the set of all non-zero entries of D^i in L which are at most c.
     (b) Find witnesses for all entries in L'.
     (c) $R^{i+1} \leftarrow$ *good* dilution of R^i (see definition of "good" below)
     (d) $D^{i+1} \leftarrow A \cdot (B \wedge R^{i+1})$ (The matrix multiplication is over the integers)

A matrix R^{i+1} is *good* with respect to R^i (in step 4c above) if the following two conditions hold:

a) The total sum of the entries of $D^{i+1} = A \cdot (B \wedge R^{i+1})$ is at most $3/4$ of the total of the entries of $D^i = A \cdot (B \wedge R^i)$. (Observe that this guarantees that after $1 + 3\log_{4/3} n$ iterations of good R^i's all the entries of D will vanish.)

b) The fraction of entries in L that are 0 in D^{i+1}, among those larger than c in D^i is at most α.

2.3 Analysis of the algorithm

We first analyse one iteration of a randomized version of the algorithm (Lemma 1), then analyse it when the sample space has a small bias (Lemma 2) and finally we show that this suffices for achieving a deterministic algorithm.

Lemma 1 *For any $1 \leq i \leq t$, suppose that $R^{i+1} \leftarrow R^i \wedge S$ in step 4c where S is a random $0, 1$ matrix, then the R^{i+1} is good with probability at least $1/6$.*

The lemma follows from the following three claims:

Claim 1 *The probability that the sum of entries of D^{i+1} is at most $3/4$ the sum of entries of D^i is at least $1/3$.*

To see this, observe that the expected sum of entries of D^{i+1} is $1/2$ the sum of entries of D^i, since for every $1 \leq j, k, l \leq n$ such that $A_{jk} = (B \wedge R^i)_{kl} = 1$ the probability that $(B \wedge R^{i+1})_{kl} = 1$ is exactly $1/2$. The claim then follows from Markov's Inequality. $\Box$

Claim 2 *The probability that a fixed entry of D^i which is at least c drops down to 0 in D^{i+1} is at most $1/2^c$.*

This is obvious. Observe that the claim holds even if we only assume that every c entries of S are independent. $\Box$

Claim 3 *The probability that more than a fraction α of the entries in L that had a value at least c in D^i drop to 0 in D^{i+1} is at most $\frac{1}{2^c}\frac{1}{\alpha} = \frac{1}{8}$.*

This follows from Claim 2 by Markov's Inequality. $\Box$

Since Claims 1 and 3 describe the event we are interested in and $1/3 - 1/8 > 1/6$ the lemma follows. $\Box$

Define $\epsilon = \frac{1}{2^{c+1}}$. The crucial point is to observe that the proof of the above lemma still holds, with almost no change, if the matrix S is not totally random but its entries are chosen from a c-wise ϵ-dependent distribution in the sense of [23], [5]. Recall that if m random variables whose range is $\{0, 1\}$ are *c-wise ϵ-dependent* then every subset of $j \leq c$ of them attains each of the possible 2^j configurations of 0 and 1 with probability that deviates from $1/2^j$ by at most ϵ.

Lemma 2 *If $R^{i+1} \leftarrow R^i \wedge S$ in step 4c where the entries of S are chosen as n^2 random variables that are c-wise ϵ-dependent, then R^{i+1} is good with probability at least $1/12 - 2\epsilon$.*

We note that in fact it is sufficient to choose only one column from a c-wise ϵ-dependent sample space and copy it n times. However, this changes the size of the sample space by a constant factor only. The proof of Lemma 2 is by the following modified three claims, whose proofs are analogous to those of the corresponding previous ones.

Claim 4 *The probability that the sum of entries of D^{i+1} is at most $3/4$ the sum of entries of D^i is at least $1/3 - 2\epsilon$.* $\Box$

Claim 5 *The probability that a fixed entry of D^i which is at least c drops down to 0 in D^{i+1} is at most $1/2^c + \epsilon$.* $\square$

Claim 6 *The probability that more than a fraction α of the entries in L that had a value least c in D^i drop to 0 in D^{i+1} is at most $(\frac{1}{2^c} + \epsilon)\frac{1}{\alpha} < \frac{2}{2^c}\frac{1}{\alpha} = 1/4$.* $\square$

Since Claims 4 and 6 describe the event we are interested in and $1/3 - 2\epsilon - 1/4 > 1/12 - 2\epsilon$ the lemma follows. $\square$

As shown in [23] and in [5] there are explicit probability spaces with n^2 random variables which are c-wise ϵ-dependent, whose size is

$$(\log(n) \cdot c \cdot \frac{1}{\epsilon})^{2+o(1)},$$

which is less than, e.g., $O((\log n)^5)$. Moreover, these spaces can be easily constructed in time negligible with respect to the total running time of our algorithm.

Suppose that in step 4c all the matrices S defined by such a probability space are searched, until a good one is found. Checking whether a matrix is good requires only matrix multiplication plus $O(n^2)$ operations. Therefore the inner loop (starting at step 4) takes polynomial in $\log n$ times matrix multiplication time.

Executing Step 4b: It is important to note that during the performance of step 4c, while considering all possible matrices S provided by our distribution, we can accomplish step 4b (of the next iteration) as well. To see this we need

Claim 7 *If $R^{i+1} \leftarrow R^i \wedge S$ in step 4c where the entries of S are chosen as n^2 random variables that are c-wise ϵ-dependent, then for each entry in L' there is a positive probability to be precisely 1 in D^{i+1}.*

This follows since if S is chosen uniformly at random, then the probability that an entry in L' is precisely 1 in D^{i+1} is at least $c/2^c$ and this event depends on at most c variables. $\square$

To apply the claim, recall the observation at the beginning of Section 2.1, that we can find witnesses for entries that are at most 1. If we replace each matrix multiplication in the search for a good S by two matrix multiplications as described in that observation, we complete steps 4b and 4c together.

Analysis of the outer loop: In every iteration of the inner loop 4 at most an α fraction of the entries of L are "thrown" (i.e. their witness will not be found in this iteration of the outer loop). Therefore at least $1 - (1 + 3\log_{4/3} n)\alpha$ fraction of the entries of D in L will *not* be thrown during the completion of these iterations. For those entries, which are at least $1/2$ of the entries in L, a witness is found. Therefore, only $O(\log n)$ iterations of the outer loop are required, implying the desired $\tilde{O}(n^\omega)$ total running time.

We have thus proved the following:

Theorem 1 *The witnesses for the Boolean multiplication of two n by n matrices can be found in deterministic $\tilde{O}(n^\omega)$ time.*

3 Efficient deterministic construction of perfect hash functions

In this Section we describe an efficient method of constructing perfect hash functions. Recall that for a set $S \subset \{1,\ldots,m\}$ a perfect hash function is a mapping of $\{1,\ldots,m\}$ onto $\{1,\ldots,n\}$ which is 1-1 on S.

We are given a set of n elements out of $\{1,\ldots,m\}$ and the goal is to build a perfect hash function from $\{1,\ldots,m\}$ to a range which is $O(n)$ with the properties listed in Section 1.2 (Given a function that maps to a range of size $O(n)$, another function which maps to a range of size n can be constructed using the technique in [12] or in [10].)

Outline of the FKS scheme: Our scheme has the same structure as the one of Fredman, Komlós and Szemerédi [12] which we now review: The FKS scheme consists of two levels. The first level function, denoted by h, maps the elements of $\{1,\ldots,m\}$ into a range of size $O(n)$; all the elements that were sent to the same location i are further hashed using a second level hash function h_i. The second level hash function h_i should be 1-1 on the subset that was hashed to location i by h. For every i in the range of h we allocate as much space as the range of h_i which we denote by r_i. The perfect hash function is now defined as follows: if $x \in \{1,\ldots,m\}$ is mapped to i by h, then the scheme maps x to $h_i(x) + \sum\limits_{1\le j<i} r_j$. The size of the range is therefore $\sum_i r_i$.

Let $s_i(h) = |\{x | x \in S \textit{ and } h(x) = i\}|$, i.e $s_i = s_i(h)$ denotes the number of elements mapped to i. The property we require h to satisfy is that $\sum_{i=1}^{n} \binom{s_i(h)}{2}$ should be $O(n)$. The size of the range of h_i will be $O(\binom{s_i}{2})$. The functions suggested by [12] for both levels were of the form $(k \cdot x \bmod p) \bmod r$ where p is an appropriate prime, r is n for the first level and s_i^2 for the second level.

Overview of the new scheme:

The scheme consists of more than one level of hashing. The first level is a hash function which is used to partition the elements according to their hash value, where S_i is the set of all elements with hash value i. Then, each S_i is going to be mapped to a separate final region, say R_i, where the size of R_i depends on $|S_i|$. This first hash function is of the same form $h(x) = Ax$ where A is a $\log(n) \times \log(m)$ matrix over $GF[2]$ and x is treated as a vector of length $\log m$ over $GF[2]$. The mapping of S_i into R_i consists of either one more level of hashing or two more levels, depending on how large S_i is. If S_i is sufficiently small, then there is only one more hash function that maps S_i into R_i, and it is of the same form as [12]. If S_i is large enough, then there are two more levels of hashing to map S_i into R_i, where the bottom level is the same form as [12] but the upper level (which we will refer to as the intermediate) is of the form $h(x) = Ax$ where A is a matrix over $GF[2]$ of appropriate dimensions.

Our main difficulty is coming up with the first level hash function h. Given the proper h we can allocate relatively long time for finding the second and third level functions: even if constructing a good (i.e. 1-1) h_i takes time proportional to $O(s_i^2)$, then the total amount of work in the second step would still be linear. In fact, finding a perfect hash function of the form $(k \cdot x \bmod p) \bmod r$ requires time proportional to $s_i^3 \log m$. For the sake of completeness we outline in Section 3.2 how to achieve this.

Finding the top level hash function h and the intermediate level h_i's is done using the approach outline in the Introduction to the paper. Instead of choosing h from a collection at random we select it by considering it a concatenation of one-bit functions and fixing each one in turn. We must show that the one-bit functions can be chosen from a very small collection (defined by a small bias sample space) and that there is a good estimator of progress (which will be the number of pairs that must be separated).

Model: Since we are interested in fast on-line evaluation of the constructed perfect hash function we must specify the computational power of the evaluator. The weakest model we consider only assumes that the evaluator can access in one operation the memory that stores the description of the perfect hash function and retrieve a word of width at most $\log n$. It needs only very simple arithmetic, basically addition. Note that in this weak model we can perform a lot of computation in $O(1)$ time using pre-stored tables of of size $O(n)$ as we can see in the following example.

Consider the function $f_r(x) = r \cdot x$ where r and x are in $\{0,1\}^{\log m}$ and f_r computes their inner product over $GF[2]$. Partition r into $k = \frac{\log m}{\log n - \log\log m}$ parts $r_1, r_2, \dots r_k$. For each r_j arrange a table of size $n/\log m$ such that for $0 \leq y < n/\log m$ entry y in the table contains the inner product of y and r_j where y and r_j are considered as vectors of length $\log n - \log\log m$ over $GF[2]$. Given these tables, evaluating $f_{r_j}(x)$ requires k operations: access the tables at entries x_1, x_2, x_k and Xor the results. If m is polynomial in n than this is $O(1)$ operations and in general takes $O(\log m/\log n)$ time. This example is important to us, since the top level hash function is of the form $h(x) = Ax$ where the multiplication is over $GF[2]$

A stronger model is to assume that any operation on words of size $O(\log m)$ takes constant time. Thus we count only accesses to the memory (which may be interesting sometimes).

3.1 First level hash function

To find the first level hash function we solve the following problem for all t in the range $\{1, \dots \log n\}$: given a set $S \subset \{1, \dots, m\}$ of size n, construct a function $h : \{1, \dots, m\} \mapsto \{1, \dots 2^t\}$ such that if $s_i(h) = |\{x | x \in S \textit{ and } h(x) = i\}|$ then $\sum_{i=1}^{2^t} \binom{s_i(h)}{2} \leq q = e^2\binom{n}{2}/2^t$. Note that q is only a constant factor larger than the expected value of $\sum_i \binom{s_i}{2}$ in case h is chosen at random from all functions $\{1, \dots, m\} \mapsto \{1, \dots 2^t\}$.

We find h in a step by step manner, forming it as a concatenation of one bit functions $f_1, f_2, \dots, f_t$ where $f_j : \{1, \dots, m\} \mapsto \{0,1\}$. After we have decided on $f_1, f_2, \dots f_j$ we determine f_{j+1} using an estimator function which we try to reduce. The estimator which we use is

$$P_j(f_1, f_2, \dots f_j) = \sum_{i \in \{0,1\}^j} \binom{s_i(f_1, f_2, \dots f_j)}{2}$$

where $s_i(f_1, f_2, \dots f_j) = |S_i(f_1, \dots, f_j)|$ and $S_i(f_1, \dots, f_j) = \{x | x \in S \textit{ and } f_1 f_2 \dots f_j(x) = i\}$ is the set of all elements that were mapped by the first j functions to i. The motivation for choosing this estimator is the observation that $P_j(f_1, \dots, f_j)/2^{t-j}$ is the conditional expectation of the number of pairs mapped to the same point by h given the chosen $f_1 \dots f_j$ and assuming the rest of the bit functions will be chosen randomly.

P_0 is $\binom{n}{2}$ and if the $j+1$ function is random, then

$$E[P_{j+1}(f_1, f_2, \dots f_{j+1}) | f_1, f_2, \dots f_j] = \frac{1}{2} P_j(f_1, f_2, \dots f_j),$$

since the expected contribution from each pair of elements that have not been separated so far is $1/2$.

What should we do in order to reduce the randomness and yet get that the expectation of $P_{j+1}(f_1, f_2, \dots f_{j+1})$ is at most half of $P_j(f_1, f_2, \dots f_j)$? It is enough to have that for any two distinct $x, y \in \{1, \dots, m\}$ the probability that $f_{j+1}(x) = f_{j+1}(y)$ is at most $1/2$. This is true if f_j is selected by choosing a random vector $r \in \{0,1\}^{\log m}$ and then setting $f_j(x)$ to be the inner product modulo 2 of x and r (x is treated as a vector in $\{0,1\}^{\log m}$). Searching among all such vectors for a "good" one requires m tests, far more that we are willing to spend. Instead, we use a small collection of vectors in $\{0,1\}^{\log m}$ which (almost) preserves this property.

Suppose that f_j is chosen from some collection F. If the probability that $f_j(x)$ and $f_j(y)$ are equal is at most $1/2 + \epsilon$ then $E[P_{j+1}(f_1, f_2, \dots f_{j+1})]$ is at most $(1/2+\epsilon)P_j(f_1, f_2, \dots f_j)$ and we know that there must be some function $f \in F$ such that if we set f_{j+1} to be f, then we get that $P_{j+1}(f_1, f_2, \dots f_{j+1}) \leq (1/2+\epsilon)P_j(f_1, f_2, \dots f_j)$. If we let F be the set of points of a sample space with any pairwise ϵ-dependent distribution on m variables, then a randomly chosen f from F satisfies the above requirement. (An equivalent description of the properties of F is to say that we let F be the set of functions corresponding to computing inner products with the columns of the generating matrix of a linear error correcting code over $GF[2]$ of dimension $\log m$, length $|F|$ and distance at least $(\frac{1}{2} - \epsilon)|F|$. This is true since the requirement here is only almost pairwise independence.)

As mentioned in subsection 1.3, there are explicit collections F as above of size $|F| \leq O(\log(m)/\epsilon^3)$ (using the construction of [2]), and somewhat simpler constructions of size $O(\log^2(m)/\epsilon^2)$ (given in [5]). Searching all of F is therefore possible in polylogarithmic time. By maintaining the sets $S_i(f_1, f_2, \dots f_j)$ we can check in $O(n)$ time (when m is polynomial in n) whether a given $f \in F$ is good (i.e., achieves $P_{j+1}(f_1, f_2, \dots f_{j+1})$ that does not exceed $(1/2+\epsilon)P_j(f_1, f_2, \dots f_j)$): we simply go over these sets and examine to what size subsets these are split by introducing f as f_{j+1}.

The procedure is therefore:

- set $\epsilon = 1/t$ and find a collection F of m random variables that are pairwise ϵ-bias with $|F| \leq O(\log(m)/\epsilon^3)$.
- For $j = 1$ to t
  1. For all $f \in F$ compute $P(f) = P_j(f_1, f_2, \dots f_{j-1}, f)$
  2. Choose f_j as the f with the smallest $P(f)$
- Set $h = f_1, f_2, \dots f_t$.

We know that the choice at step 2 implies that

$$P_j(f_1, f_2, \dots f_j) \leq (1/2+\epsilon) \cdot P_{j-1}(f_1, f_2, \dots f_{j-1}).$$

Therefore

$$\begin{aligned} P(h) &= P(f_1, f_2, \ldots, f_t) \leq (1/2+\epsilon)^t \cdot P_0 \\ &= (1/2+\epsilon)^t \cdot \binom{n}{2} \\ &= (1+2\epsilon)^t \cdot 2^{-t} \cdot \binom{n}{2} \\ &= (1+2/t)^t \cdot 2^{-t} \cdot \binom{n}{2} \\ &\leq e^2 \binom{n}{2} / 2^t \end{aligned}$$

The total amount of work is $O(t \cdot |F| \cdot n)$. (The time for constructing the sample space F is negligible compared with the the time to find h). Since in our case we can choose $t = \lceil \log n \rceil$ and $|F| = O(t^3 \log m) = O(\log m \log^3 n)$ this gives a total running time of $O(n \log m \log^4 n)$. For these parameters we have that $\sum_{i=1}^{n} s_i^2(h) \leq O(n)$.

It is worth noting that by precomputing and storing linear sized tables (as illustrated in the model description) we can make the computation of h to be constant time under the strictest definition (i.e. the weaker model), as long as m is polynomial in n. For general m the time is $O(\log m / \log n)$.

3.2 Resolving collisions of $S_i(h)$

We now turn to the question of resolving the collisions of $S_i(h)$. Suppose that we attempt to resolve the collisions of $S_i(h)$ using a second level a lá Fredman, Komlós and Szemerédi [12] as briefly explained below. First observe that for any set S of k elements in $\{1, \ldots, m\}$ there is a prime $p \leq k^2 \log m$ so that for any two distinct $x, y \in S$, $x \pmod{p} \neq y \pmod{p}$. Indeed, this follows from the fact that every prime that does not satisfy the above property divides the product $\Pi_{x,y \in S, x<y}(y-x)$, which is smaller than $m^{k^2/2}$ and the fact that the product of all primes up to x is $e^{(1+o(1))x}$. Given h, for any i we can find a prime $p_i \leq s_i^2(h) \log m$ such that all the elements of $S_i(h)$ are different mod p_i. Searching for this p_i does not take more time than testing all the primes smaller than $s_i^2(h) \log m$ where each test takes time $s_i(h)$. Therefore the total time is at most $s_i^2(h) \cdot \log m \cdot s_i(h) = s_i^3(h) \log m$. Given p_i, we need to find $k_i \leq p_i$ such that the function $h_i(x) = (k_i \cdot x \bmod p_i) \bmod s_i^2(h)$ is perfect on $S_i(h)$ (we are assured of its existence, since $k \cdot (x-y) \bmod p_i$ is uniformly distributed if k is chosen at random from $\{0, \ldots, p_i - 1\}$ and $x \neq y$). Again, this does not take more time than $p_i \cdot s_i(h) \leq s_i^3(h) \log m$. Therefore the total amount of work is

$$\sum_i s_i^3(h) \log(m) \leq \log(m) (\sum_i s_i^2(h))^{1.5} \tag{1}$$

which is at most $O(n^{1.5} \log m)$ since $\sum_i s_i^2(h) \leq O(n)$.

As for the length of the representation, for every i we need $O(\log s_i + \log\log m)$ bits, making it $O(n \log n + n \log\log m)$ bits altogether. However Schmidt and Siegel [27] have found a way to amortize the cost. They choose the functions so that representing them requires $O(n + \log\log m)$ bits only. We briefly describe this method: for each $S_i(h)$ at least half the primes $p \leq s_i^2(h)\log m$ are good (i.e 1-1 on $S_i(h)$) and given p_i at least half the k_i are good. Therefore, many $S_i(h)$ can have the same p and k. We construct a collection of functions of the form $(kx \bmod p) \bmod s^2$ in the following way: we partition the $S_i(h)$'s into (at most) $1/2\log n$ sets such that the jth set has all the i for which $2^j \leq s_i(h) \leq 2^{j+1}-1$. For each $1 \leq j \leq 1/2\log n$ find a p and a k that is good for $1/4$ of the S_i' of the jth set, then one that is good for $1/4$ of the remaining members and so on. The size of the collection is $O(\log^2 n)$ and for every $S_i(h)$ at least one function in the collection is perfect for $S_i(h)$. Furthermore, because of the way the collection was constructed, we can encode for every i which hash function should be used using only $O(n)$ bits by giving the more popular functions shorter codes using, say, Huffman coding. The additional time this coding requires is at most $O(\sum_i s_i^2(h)\log m)$ which is $O(n\log m)$.

The resulting total time, $O(n^{1.5}\log m)$, is larger than we are aiming for. However, notice that the small $s_i(h)$'s do not contribute much to the excess. We set (somewhat arbitrarily) a threshold at $\log n$ and call those i's such that $s_i(h) \leq \log n$ *small* and the remaining i's *large*. For any small i we choose h_i as described in the preceding paragraph. The total amount of work this takes is

$$O(\sum_{s_i \leq \log n} s_i^3 \log m) \leq O(\sum s_i^2)\log m \log n = O(n\log n\log m).$$

The large i's are those that contribute the most to (1). For them we employ a two level scheme, but note that since $\sum s_i^2$ is $O(n)$ our problem is easier than the original problem. Instead of mapping S_i to a range of size $O(s_i)$, we can map it to an exclusive range of size s_i^2 without violating the condition that the total range size be $O(n)$. Thus we have a relaxed version of the original problem, since the range can be much larger than the set for which we are resolving the collisions.

Note also that there are at most $O(n/\log^2 n)$ large i's, since

$$O(n) \geq \sum s_i^2 \geq \sum_{\text{large } i} s_i^2 \geq \sum_{\text{large } i} \log^2 n = (\#\text{ of large } i)\log^2 n$$

This means that we have some flexibility in the size of the representation as well (which is significant, since there is a $\log\log n$ lower bound on the size of the description of h_i).

We now describe in detail how to deal with the large i's. First, h_i is chosen by a similar method to which h was chosen. I.e., h_i is composed in a step by step manner as the concatenation of the one-bit functions $g_1, g_2, \ldots$

The estimator we use is, as in the construction of h,

$$P_k^i(g_1, g_2, \ldots g_k) = \sum_{j\in\{0,1\}^k} \binom{s_{ij}(g_1, g_2, \ldots g_k)}{2}$$

where $s_{ij}(g_1, g_2, \ldots g_k) = |S_{ij}(g_1, \ldots, g_k)|$ and

$$S_{ij}(g_1, \ldots, g_k) = \{x | x \in S_i(h) \textit{ and } g_1 g_2 \ldots g_k(x) = j\}.$$

The procedure is now

- Set $\delta = (2^{1.25/2}-1)/2$ and $t_i = 2\log s_i(h)$. (The choice of δ guarantees that $(1+2\delta)^{2\log s_i}$ will be at most $s_i^{1.25}$.) Find a sample space G of m random variables that are pairwise δ-bias with $|G| \leq O(\log(m)/\delta^3)$ (here we should use the construction in [2]).
- For $k = 1$ to t_i
  1. For all $g \in G$ compute $P^i(g) = P^i_k(g^i_1, g^i_2, \ldots g^i_{k-1}, g)$
  2. Choose g^i_k as the g with the smallest $P^i(g)$
- Set $h_i = g^i_1, g^i_2, \ldots g^i_{t_i}$.

As before, we are assured that at the end of the procedure

$$\begin{aligned}
\sum_j \binom{s_{ij}(h_i)}{2} &= P^i(h_i) \\
&= P^i(g^i_1, g^i_2, \ldots, g^i_{t_i}) \leq (1/2+\delta)^{t_i} \cdot P^i_0 \\
&= (1/2+\delta)^{t_i} \cdot \binom{s_i(h)}{2} \\
&= (1+2\delta)^{t_i} \cdot 2^{-t_i} \cdot \binom{s_i(h)}{2} \\
&= (1+2\delta)^{2\log s_i(h)} \cdot 2^{-2\log s_i(h)} \cdot \binom{s_i(h)}{2} \\
&\leq s_i^{1.25}/2
\end{aligned}$$

The amount of work spent constructing h_i is $O(s_i \log m \log n)$ and therefore the amount of work spent constructing all the h_i's is $O(n \log n \log m)$.

The third level hash functions h_{ij} are chosen by the method described above for the functions h_i corresponding to small s_i. The total amount of work is

$$O(\sum_j s^3_{ij} \log m) \leq O(\log m (\sum_j s^2_{ij})^{1.5}) \leq O((s_i^{1.25})^{1.5} \log m) \leq O(s_i^2(h) \log m).$$

Since $\sum s_i^2(h) \leq O(n)$ we conclude that the total amount of work for the computation of all the third level functions on the large $s_i(h)$ is at most $O(n \log m)$.

Note that since δ is fixed the size of G is $O(\log m)$. The description of h_i is simply a subset of the members of G. Therefore, as with the first level hash function, we have that by precomputing and storing linear sized tables (the same tables for all the h_i's!) we can make the computation of h to be constant time under the strictest definition, as long as m is polynomial in n.

The length of the description of the function (including all the levels) is the sum of:

1. The number of bits needed to describe h, which is $\log n \log m$ (it can be compressed to $\log n \log |F|$)

2. The number of bits to describe the second and third level hash functions: for the small i's representing h_i requires $O(\log\log m + \log s_i(h))$ bits. However, using the amortization, all the h_i's can be represented by $O(n + \log\log m)$ bits. For the large i's for which $s_i(h) > \log n$, first recall that there can be at most $n/\log^2 n$ of them. Also, the description of each h_i is simply that of a subset of G which can be described by $O(\log m)$ bits. It follows that for the second level for these h_i's at most $O(n/\log^2 n \cdot \log m)$ bits suffice. For a fixed i, all the third level functions h_{ij} can easily be represented by $O(s_i^2(\log s_i + \log\log m)$ bits. However this can be reduced to $O(s_i^2 + \log\log m)$ via amortization. Since $\sum_i s_i^2$ is $O(n)$, all the third level hash functions together require at most $O(n/\log^2 n \cdot \log\log m + n)$ bits.

3. Some bookkeeping information, e.g. $\sum_{1\leq j<i} r_j$ for all $1 \leq i \leq n$. This can be done using $O(n)$ bits (see e.g. [10]).

4. Tables to allow fast computation of h and the intermediate h_i's - $O(n)$ bits.

For $m = n^{O(1)}$ the total number of bits is therefore linear in n and we obtain the following:

Theorem 2 *For any polynomial $q(n)$ there is a deterministic algorithm that constructs a perfect hash function for a given subset $S \subset \{1, \ldots m\}$ of cardinality n where $m = q(n)$. The perfect hash function can be found in time $\tilde{O}(n)$, its description requires $O(n)$ bits of representation and the value of the hash function on any $x \in \{1, \ldots, m\}$ can be computed in constant time.*

If m is superpolynomial in n then we can still get something: we first find a 1-1 mapping of S into $\{1, \ldots, n^3\}$ using the same method to construct the first level hash function but with ϵ a fixed constant (as we do for the intermediate level). This takes time $O(n \log m \log n)$. We proceed from there on as if the range is n^3, getting an additional $\tilde{O}(n)$ factor. The only problem in computing the hash value of a given $i \in \{1, \ldots, m\}$ quickly is the time to compute the initial hash function. Depending on the exact computational model (i.e. what we assume that can be done in $O(1)$ time) it may take $O(\log m/\log n)$ time or $O(1)$ to compute it.

Theorem 3 *For any m there is a deterministic algorithm that constructs a perfect hash function for a given subset $S \subset \{1, \ldots m\}$ of cardinality n. The perfect hash function can be found in time $\tilde{O}(n \log m)$, its description requires $O(n + \log n \cdot \log\log m)$ bits and the value of the hash function on any $x \in \{1, \ldots, m\}$ can be computed in $O(\log m/\log n)$ time (or constant time if we assume the stronger model).*

Remark: We do not know whether the scheme described above can be made implicit, i.e. not requiring any additional memory to represent the function (in the sense of [9]). The methods described in [10] and [9] for making the FKS scheme implicit are applicable here as well. The problem however seems to be in finding a deterministic way of doing the encoding in nearly linear time. Consider for instance the following problem: given $n/4$ values $v_1, v_2, \ldots v_{n/4}$ in $\{0, \ldots n-1\}$ find $n/4$ *disjoint* pairs $(x_1, y_1), (x_2, y_2), \ldots (x_{n/4}, y_{n/4})$ such that $v_i = x_i - y_i \bmod n$. The deterministic algorithm for this problem seems to take $O(n^2)$ time.

Remark: One can construct a similar alternative algorithm, which uses only two levels, by using a 3-wise ϵ-dependent distribution in the first level to conclude that by the end of this level the inequality $\sum \binom{s_i}{3} = O(n)$ holds. We omit the details.

Acknowledgments

We thank the anonymous referees and Mike Luby for their many useful remarks and suggestions.

References

[1] N. Alon, L. Babai and A. Itai, *A fast and simple randomized parallel algorithm for the maximal independent set problem*, Journal of Algorithms 7 (1986), pp. 567–583.

[2] N. Alon, J. Bruck, J. Naor, M. Naor and R. Roth, *Construction of asymptotically good, low-rate error-correcting codes through pseudo-random graphs*, IEEE Transactions on Information Theory, 38 (1992), 509–516.

[3] N. Alon, Z. Galil and O. Margalit, *On the exponent of the All Pairs Shortest Path problem*, Proc. 32th IEEE annual Symposium on Foundations of Computer Science (1991), 569–575. Also: J. Computer and System Sciences, to appear.

[4] N. Alon, Z. Galil, O. Margalit and M. Naor, *Witnesses for Boolean matrix multiplication and for shortest paths*, Proc. 33^{rd} IEEE Annual Symposium on Foundations of Computer Science (1992), 417–426.

[5] N. Alon, O. Goldreich, J. Hastad and R. Peralta , *Simple constructions of almost k-wise independent random variables*, Proc. 31^{st} IEEE Symposium on Foundations of Computer Science (1990), 544–553. Also: Random Structures and Algorithms 3 (1992), 289-304.

[6] N. Alon and J. Spencer, **The probabilistic method**, John Wiley and Sons Inc., New York, 1991.

[7] B. Berger and J. Rompel, *Simulating* $(\log^c n)$*-wise independence in NC*, J. of the ACM 38 (1991), pp. 1026–1046.

[8] D. Coppersmith and S. Winograd, *Matrix multiplication via arithmetic progressions*, Journal of Symbolic Computation 9(1990), 251–280.

[9] A. Fiat and M. Naor, *Implicit* $O(1)$ *probe search*, SIAM J. Comp. 22 (1993), pp. 1–10.

[10] A. Fiat, M. Naor, J. P. Schmidt and A. Siegel, *Non-oblivious hashing*, J. of the ACM 31 (1992), pp. 764–782.

[11] M.L. Fredman and J. Komlós. On the size of separating systems and families of perfect hash functions. *SIAM Journal on Algebraic and Discrete Methods*, 5(1):61–68, 1984.

[12] M. L. Fredman, J. Komlós and E. Szemerédi, *Storing a Sparse Table with $O(1)$ Worst Case Access Time*, J. of the ACM 31 (1984), 538–544.

[13] Z. Galil and O. Margalit, *A faster algorithm for the All Pairs Shortest Path problem for undirected graphs*, Manuscript, August 1991.

[14] Z. Galil and O. Margalit, *Witnesses for Boolean matrix multiplication and for transitive closure*, J. of Complexity 9 (1993), 201–221.

[15] R. M. Karp and A. Wigderson, *A Fast Parallel Algorithm for the Maximal Independent Set Problem*, J. of the ACM 32 (1985), 762–773.

[16] J. Körner, Fredman-Komlos bounds and information theory, *SIAM. J. Alg. Disc. Meth.*, 7:560–570, 1986.

[17] M. Luby, *A simple parallel algorithm for the maximal independent set problem*, SIAM J. Comp. 15 (1986), pp. 1036–1053.

[18] M. Luby, *Removing randomness in parallel computation without a processor penalty*, Journal of Computer Systems and Science 47 (1993), pp. 250–286.

[19] H. Mairson, *The effect of table expansion on the program complexity of perfect hash functions*, BIT 32, 1992, 430–440.

[20] O. Margalit, PhD. Dissertation, Tel-Aviv Univ. 1993.

[21] K. Mehlhorn, **Data Structure and Algorithms 1: Sorting and Searching**, Springer-Verlag, Berlin, 1984.

[22] R. Motwani, J. Naor and M. Naor, *The probabilistic method yields deterministic parallel algorithms*, Proceedings of the 30th IEEE Symposium on Foundations of Computer Science, (1989), pp. 8–13.

[23] J. Naor and M. Naor, *Small-bias probability spaces: efficient constructions and applications*, SIAM J. on Computing 22, 1993, pp. 838–856.

[24] A. Nilli, *Perfect hashing and probability*, Combinatorics, Probability and Computing, to appear.

[25] P. Raghavan, *Probabilistic construction of deterministic algorithms: approximating packing integer programs*, Journal of Computer Systems and Science 37 (1988), pp. 130–143.

[26] Sager, *A Polynomial time generator for minimal perfect hash functions* CACM 28, 1985.

[27] J. Schmidt and A. Siegel, *The Spatial Complexity of oblivious k-probe hash functions*, SIAM J. Comp. 19 (1990), pp. 775–786.

[28] R. Seidel, *On the All-Pairs-Shortest-Path Problem*, Proc. 24th annual ACM Symposium on Theory of Computing(1992), 745–749.

[29] J. Spencer, **Ten lectures on the probabilistic method**, SIAM (Philadelphia), 1987.

[30] R.E. Tarjan and A.C. Yao, *Storing a Sparse Table*, Communications of the ACM 22, 1979, pp. 606–611.