Local dependency dynamic programming in the presence of memory faults

Saverio Caminiti, Irene Finocchi, and Emanuele G. Fusco

Department of Computer Science, *Sapienza* University of Rome
Via Salaria 113, 00198 Rome, Italy
`{caminiti,finocchi,fusco}@di.uniroma1.it`

Abstract

We investigate the design of dynamic programming algorithms in unreliable memories, i.e., in the presence of faults that may arbitrarily corrupt memory locations during the algorithm execution. As a main result, we devise a general resilient framework that can be applied to all local dependency dynamic programming problems, where updates to entries in the auxiliary table are determined by the contents of neighboring cells. Consider, as an example, the computation of the edit distance between two strings of length n and m. We prove that, for any arbitrarily small constant $\varepsilon \in (0, 1]$ and $n \geq m$, this problem can be solved correctly with high probability in $O\left(nm + \alpha\delta^{1+\varepsilon}\right)$ worst-case time and $O(nm + n\delta)$ space, when up to δ memory faults can be inserted by an adversary with unbounded computational power and $\alpha \leq \delta$ is the actual number of faults occurring during the computation. We also show that an optimal edit sequence can be constructed in additional time $O\left(n\delta + \alpha\delta^{1+\varepsilon}\right)$. It follows that our resilient algorithms match the running time and space usage of the standard non-resilient implementations while tolerating almost linearly-many faults.

1998 ACM Subject Classification B.8 [Performance and reliability]; F.2 [Analysis of algorithms and problem complexity]; I.2.8 [Dynamic programming].

Keywords and phrases Unreliable memories, fault-tolerant algorithms, local dependency dynamic programming, edit distance.

Digital Object Identifier 10.4230/LIPIcs.xxx.yyy.p

1 Introduction

Dynamic random access memories (DRAM) are susceptible to errors, where the logical state of one or multiple bits is read differently from how it was last written. Such errors may be due either to hardware problems or to transient electronic noises [14]. A recent large-scale study of DRAM memory errors reports data collected in the field from Google's server fleet over a period of nearly 2.5 years [20] observing DRAM error rates that are orders of magnitude higher than previously reported in laboratory conditions. As an example, a cluster of 1000 computers with 4 gigabytes per node can experience one bit error every three seconds, with each node experiencing an error every 40 minutes. If errors are not corrected, they can lead to a machine crash or to applications using corrupted data. Silent data corruptions are a major concern in the reliability of modern storage systems, since even a few of them may be harmful to the correctness and performance of software. To cope with this, a recent trend is to design applications that are more tolerant to faults: this "robustification" of software involves re-writing it so that dealing with faults simply causes the execution to take longer. Unfortunately, most algorithms and data structures are far from being robust: since the contents of memory locations are supposed not to change throughout the execution unless they are explicitly written by the program, wrong steps may be taken upon reading corrupted

© John Q. Open and Joan R. Access;
licensed under Creative Commons License NC-ND
Conference title on which this volume is based on.
Editors: Billy Editor, Bill Editors; pp. 1–12

Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

values, yielding unpredictable results. Coping with memory faults appears to be of particular importance for all those applications handling massive data sets, for long-living processes, for safety critical applications in avionics systems, and for cryptographic protocols, that can be compromised by fault-based attacks that work by inducing very timing-precise bit flips.

Related work. Algorithmic research related to memory errors spans more than thirty years. Starting from the "twenty questions game" posed by Rényi and Ulam in the late 70's, many results have been obtained in the liar model: see, e.g., the extensive survey in [18]. More recently, sorting and selection have been studied in the "just noticeable difference model", where the outcome of comparisons is unpredictable if the compared values are within a fixed threshold [1]. All these works typically assume transient comparator failures, but no corruption of data. Destructive faults have been first investigated in the context of fault-tolerant sorting networks [17], and many subsequent works have focused on the design of resilient data structures in a variety of (hardly comparable) models. Pointer-based data structures are the subject of [2] and error-correcting data structures for fundamental problems related to membership have been presented in [7, 9]. The more restrictive problem of checking (but not recovering) the behavior of large data structures that reside in an unreliable memory has also received considerable attention [3, 8].

A variety of resilient algorithms have been designed in the faulty-memory random access machine (*faulty RAM*) introduced in [12], where an adversary can corrupt at most δ memory cells of a large unreliable memory during the execution of an algorithm. The algorithms can exploit knowledge of δ, which is a parameter of the model, and the existence of a constant number of incorruptible registers, but do not require error detection capabilities. Resiliency is achieved if a problem is solved correctly (at least) on the set of uncorrupted values. This relaxed definition of correctness fits naturally sorting and searching problems addressed so far in this model [10], as well as the design of resilient data structures such as dictionaries [11] and priority queues [15]. As an example, given a set of n values, it is possible to sort correctly the subset of uncorrupted values in a comparison-based model using optimal $\Theta(n \log n)$ time when $\delta = O(\sqrt{n})$ [10]. Resilient counters in the faulty RAM model are described in [6], showing different tradeoffs between the time for incrementing a counter and its additive error. Motivated by the impact of memory errors on applications operating with massive data sets, the connection between fault-tolerance and I/O-efficiency is investigated in [5], providing the first external-memory algorithms resilient to memory faults.

Our results. In spite of the wealth of results summarized above, it remains an open question whether powerful algorithmic techniques such as those based on dynamic programming can be made to work in the presence of faults. This has been regarded as an elusive goal for many years in a variety of faulty memory models. In this paper we provide the first positive answers to this question by showing how to implement a large class of dynamic programming algorithms resiliently in unreliable memories. We consider the faulty RAM model introduced in [12] and we illustrate our techniques using as a case study the problem of computing the edit distance between two strings of length n and m. A simple-minded resilient implementation of the standard dynamic programming algorithm for edit distance could be based on replicating all data (string symbols and table values) $2\delta + 1$ times. By applying majority techniques, this would allow to tolerate up to δ faults at the cost of a multiplicative $\Theta(\delta)$ overhead on both space usage and running time. Hence, only a *constant* number of faults could be tolerated while maintaining the standard $O(nm)$ time bound. In contrast, we devise algorithms that preserve this bound while tolerating, with high probability, up to an almost *linear* number of faults. We will prove that this is nearly optimal. More formally, we

show that the edit distance between two strings of length n and m can be correctly computed, with high probability, in $O\left(nm + \alpha\delta^{1+\varepsilon}\right)$ worst-case time and $O(nm + n\delta)$ space, when $\alpha \leq \delta$ faults occur during the computation, $n \geq m$, and ε is an arbitrarily small constant in $(0, 1]$. Our algorithms exploit knowledge of δ and only a constant number of private memory words. If the private memory can be enlarged to $O(\log \delta)$ words, the fault-dependent additive term in the running time becomes $O(\alpha\delta)$. The framework we provide is general enough to be applied to all local dependency dynamic programming problems, where updates to entries in the auxiliary table are determined by the contents of neighboring cells: this is a significant class of problems that includes, e.g., longest common subsequence and many sequence alignment problems in bioinformatics. Our framework can be made deterministic, yielding an algorithm that tolerates a logarithmic number of faults, and can be extended to incorporate well-known optimizations of dynamic programming, such as Hirschberg's space-saving technique [13] and Ukkonen's distance-sensitive algorithm [21]. Due to the lack of space, some proofs and details are omitted.

Techniques. Our resilient implementation does not rely on any cryptographic assumption. Instead, it hinges upon a novel combination of majority techniques (which are a typical but expensive error correction method), read and write Karp-Rabin fingerprints (to detect faults), and an asymmetric, hierarchical decomposition of the dynamic programming table into rectangular slices of height δ and decreasing width (to bound the cost of error recovery). We remark that, although fingerprints have been successfully used in the context of checking the correctness of data structures, they alone are not powerful enough in the faulty RAM model, where the goal is to recover the computation when a fault is detected without restarting it from scratch. Hence, to obtain the $O\left(\alpha\delta^{1+\varepsilon}\right)$ additive term in the running time, we exploit as a main ingredient a hierarchy of $O(1/\varepsilon)$ levels of data replication. At all levels, except for the last one, data are stored *semi-resiliently* in the unreliable memory, by replicating each variable $o(\delta)$ times. Notice that semi-resilient data could be corrupted by the adversary, but at the cost of a large number of faults: this will allow us to amortize the cost of a slice recomputation (semi-resilient variables need to be appropriately "refreshed" upon detection of faults, so that the cost of a slice recomputation can always be charged to distinct faults). $O(1/\varepsilon)$ *long-distance fingerprints* stored in safe memory make it possible to backtrack the computation, at any time, to a checkpoint that is safe with high probability. Combining semi-resiliency with refreshing and long-distance fingerprints allows us to guarantee the correctness of the table computation while bounding the error recovery cost.

A different technique must be used during the traceback process that computes an optimal solution from its optimal value: at this point, long-distance fingerprints are no longer available, and thus we have no guarantee that semi-resilient variables are correct. To overcome this issue, we proceed incrementally in $O(1/\varepsilon)$ passes: at each pass, either we increase our confidence that the computed path is correct, or we are guaranteed that the adversary has introduced a large number of faults.

2 Preliminaries

We assume a unit cost RAM with wordsize w. We distinguish between *unreliable*, *safe*, and *private* memory. Up to δ unreliable memory words may be corrupted during the execution of an algorithm by an adaptive adversary with unlimited computational power. We denote by $\alpha \leq \delta$ the actual number of faults occurring during the computation. No error-detection mechanism is provided. We have $O(1)$ safe memory words that the adversary can read but not overwrite: without this assumption, no reliable computation would be possible [12].

Similarly to [3, 10, 11], we also assume $O(1)$ private memory words, that the adversary cannot even read: this is necessary to prevent the adversary from discovering random bits used by the algorithms.

Resilient variables. A *resilient variable* x consists of $2\delta + 1$ copies of a standard variable [11]. A *reliable write* operation on x means assigning the same value to each copy. Similarly, a *reliable read* means calculating the majority value, which is correct since at most δ copies can be corrupted. This can be done in $\Theta(\delta)$ time with the majority algorithm in [4], which scans the $2\delta + 1$ values keeping a single majority candidate and a counter in safe memory. Throughout the paper we will also make use of *r-resilient* variables (with $r < \delta$), which consist of $2r + 1$ copies of a standard variable. A r-resilient read operation on an (at least) r-resilient variable is obtained by computing the majority value on $2r + 1$ copies. Notice that a r-resilient variable can be corrupted by the adversary, but at the cost of at least $r + 1$ faults.

Generation of random primes. Random primes are usually generated by selecting a number uniformly at random and testing it for primality with, e.g., the Miller-Rabin test [19]. If the test is successful, the selected number is returned, otherwise a new candidate is selected and the process is iterated. The Miller-Rabin test has one-sided error: it can output *prime* for a composite number with a provably small probability. We keep this scheme almost unchanged, except for bounding the number of iterations so as to avoid having an expected running time. Although the probability of failure in our case does not uniquely depend on the Miller-Rabin test, it is not difficult to prove that this probability remains small and that the algorithm can be executed in our model:

▶ **Lemma 1.** *For any constants $\gamma, c > 0$, it is possible to independently select α (not necessarily distinct) prime numbers in $I = [n^{c-1}, n^c]$, uniformly at random, with error probability bounded by α/n^γ. Each prime selection requires time polylogarithmic in n using a constant number of memory words.*

3 An $O(nm + \alpha\delta^2)$ algorithm for edit distance

Given two strings $X = x_1 \cdots x_n$ and $Y = y_1 \cdots y_m$ over a finite alphabet Σ, the edit distance (a.k.a. Levenshtein distance) between X and Y is the number of edit operations (insertions, deletions, or character substitutions) required to transform X into Y. Let $e_{i,j}$, for $0 \le i \le n$ and $0 \le j \le m$, be the edit distance between prefix $x_1 \cdots x_i$ of string X and prefix $y_1 \cdots y_j$ of string Y (the prefix is empty if $i = 0$ or $j = 0$). Values $e_{i,j}$ are defined as follows:

$$e_{i,j} := \begin{cases} e_{i-1,j-1} & \text{if } i,j > 0 \text{ and } x_i = y_j \\ 1 + \min\{e_{i-1,j}, e_{i,j-1}, e_{i-1,j-1}\} & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases} \tag{1}$$

where $e_{i,0} = i$, $e_{0,j} = j$, and $e_{n,m}$ represents the edit distance of X and Y. The standard dynamic programming algorithm stores values $e_{i,j}$, for $i, j > 0$, in a $n \times m$ table M whose entries can be computed, e.g., in column-major order in $\Theta(nm)$ time. While calculating the edit distance only requires to keep in memory two table columns, an optimal edit sequence can be obtained by a tracing back process that reads table M backward: in this case, the space usage of the standard implementation is $\Theta(nm)$.

When executed in faulty memories, such a dynamic programming algorithm could provide a wrong result, since undetected memory faults could corrupt either the input strings or the values stored in table M and could be easily propagated to $M[n, m]$. In the rest of this section we describe a basic resilient edit distance algorithm (`RED`) with running time

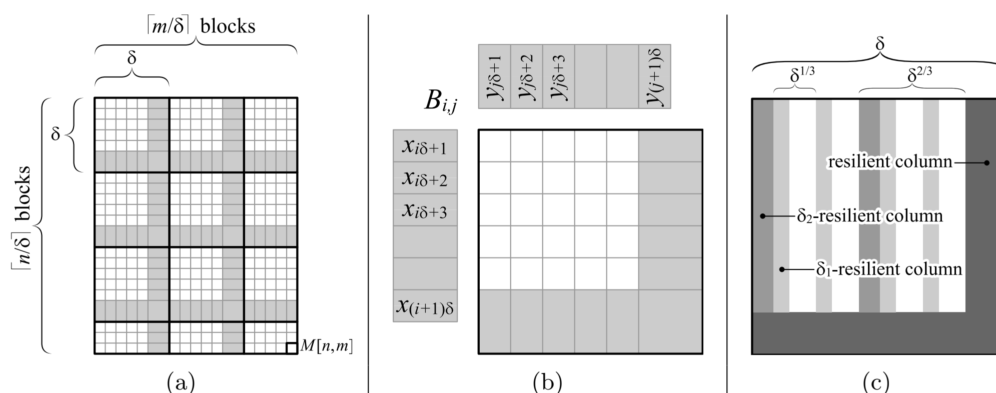

Figure 1 a) Table decomposition and resilient block boundaries (gray color) when $n \geq m \geq \delta$; b) block computation with unreliable input; c) hierarchical table decomposition for $k = 3$.

$O(nm + \alpha\delta^2)$. In Section 4 we will show how to decrease the fault-dependent additive term in the running time. W.l.o.g., throughout the paper we assume that $n \geq m$; we also assume that n and $|\Sigma|$ fit into a memory word of size w.

Algorithm RED mimics the behavior of the standard non-resilient dynamic programming approach, performing additional work in order to cope with memory faults. During the computation of table M, we compute fingerprints that allow us to determine whether some memory fault occurred in a given set of memory words. Once detected, a fault should not force us to recompute the entire table: this would result in a $\Theta(\delta)$ multiplicative overhead on the running time in the worst case. Hence, we divide the table in blocks and we write the boundaries of the blocks reliably in the unreliable memory. Upon detection of a failure, we recompute only the current block. We now describe the table decomposition into blocks, the computation of each block, and the usage of fingerprints to detect faults. We also describe how to handle faulty input strings.

Table decomposition. The $n \times m$ table M is split into square *blocks* of side length δ (see Figure 1a). Algorithm RED populates table M block by block, considering blocks in column-major order. The last row and column of each block are written reliably in the unreliable memory, using $2\delta + 1$ memory words for each value as described in Section 2. It follows that each block of δ^2 values takes roughly $5\delta^2$ memory words. Blocks are smaller (and not necessarily square) on the boundaries, whenever n or m are not divisible by δ: in this case the last row and/or column may not be written reliably.

Block computation. Let $B_{i,j}$ be an internal block (boundary blocks can be treated similarly). Entries of $B_{i,j}$ are processed in column-major order. The first column of $B_{i,j}$ is computed reliably: this is possible because row δ of block $B_{i-1,j}$, column δ of $B_{i,j-1}$, and entry $B_{i-1,j-1}[\delta, \delta]$ are written reliably in the unreliable memory. During the computation of column 1, a fingerprint φ_1 is calculated (details are given below). Let us now consider a generic column k, for $k > 1$. Each value $v_1, \ldots, v_\delta$ in column k is written unreliably in the faulty memory as soon as it is computed, except for the values in row δ and in column δ that are written reliably. While scanning column k top-down, the algorithm also computes two fingerprints: a fingerprint φ_k that is a function of values $v_1, \ldots, v_\delta$ *written* to column k, and a fingerprint $\overline{\varphi}_{k-1}$ that is a function of values *read* from column $k-1$. The two fingerprints, together with the fingerprint φ_{k-1} previously computed during the calculation of column $k-1$, are stored in the private memory. When column k is completed, algorithm

RED compares fingerprints φ_{k-1} and $\overline{\varphi}_{k-1}$: we call this a *fingerprint test.* If the fingerprints match, both of them are discarded and the computation of column $k+1$ begins. Otherwise, a memory fault has been detected: in this case the computation of the block is restarted.

Fault detection. We use Karp-Rabin fingerprints [16] to detect faults. The fingerprint for column k is defined as $\varphi_k = v_1 \circ v_2 \circ \ldots \circ v_\delta \bmod p$, where the δ values v_h of column k are considered as bit strings of length equal to the word size w and symbol $\circ$ denotes string concatenation. The concatenation $v_1 \circ v_2 \circ \ldots \circ v_\delta$ is thus an integer, upper bounded by $2^{w\delta}$, corresponding to the binary representation of the entire column k. Number p is a sufficiently large prime, chosen uniformly at random at the beginning of the execution of the algorithm and after each fault detection. Using logical shifts and Horner's rule, each fingerprint can be incrementally computed while generating the values v_h in time $\Theta(\delta)$. All computations related to the fingerprints are performed in $O(1)$ private memory words, so that no information regarding the prime number p is revealed to the adversary.

Handling the input strings. We assume each symbol in strings X and Y to be written reliably in the unreliable memory after its first reading (it is clearly impossible to detect a fault corrupting an input symbol before it is read for the first time). In order to compute matrix M in $O(nm)$ time, each comparison of the input symbols required by Equation 1 must be performed in constant (amortized) time. Let $B_{i,j}$ be a $\delta \times \delta$ block. The input values involved in the computation of $B_{i,j}$ are $x_{i\delta+1}, \ldots, x_{(i+1)\delta}$ and $y_{j\delta+1}, \ldots, y_{(j+1)\delta}$ (see Figure 1b). Consider a column k of block $B_{i,j}$. Character $y_{j\delta+k}$ is the only character of string Y needed to compute the values in column k: we read $y_{j\delta+k}$ reliably and amortize this $\Theta(\delta)$ operation on the cost of computing the δ values of column k. Conversely, all characters $x_{i\delta+1}, \ldots, x_{(i+1)\delta}$ are required to compute each column of $B_{i,j}$, and we cannot afford reading each of them reliably δ times. These input characters are thus read reliably once, while computing the first column of the block, producing a fingerprint φ_x that is kept in the private memory. While processing a column $k > 1$, values $x_{i\delta+1}, \ldots, x_{(i+1)\delta}$ are read unreliably (i.e., considering only one copy) and a fingerprint $\overline{\varphi}_{x,k}$ is computed. If fingerprint $\overline{\varphi}_{x,k}$ is different from φ_x, a memory fault has been detected: the resilient variables $x_{i\delta+1}, \ldots, x_{(i+1)\delta}$ are refreshed and the entire block is recomputed from scratch.

We now analyze algorithm RED, focusing first on correctness.

▶ **Lemma 2.** *For any constant $\beta > 0$, algorithm* RED *is correct with probability larger than $1 - 1/n^\beta$, when the upper bound δ on the number of memory faults is polynomial in n.*

Proof. Assume that the algorithm fails either when a composite number is generated instead of a prime, or when a fingerprint test does not detect a memory fault. This is an overestimation of the actual probability of error. Let B be a block that gets corrupted during its own computation. Assume for the time being that values in the boundaries of its neighboring blocks are correct. Then, the values written to column 1 of block B are also correct, because the neighboring entries used to compute column 1 and all input symbols involved are read reliably. By applying standard techniques, it is possible to prove that the probability that a fingerprint test does not detect a memory fault during the computation of B is at most $(\log n)/(\sigma n^{c-1}) < 1/(\sigma n^{c-2})$, for some constant $\sigma > 0$.

Now consider a game with two players. The game is divided into rounds. At each round player 1 (the algorithm) chooses uniformly at random a prime $p \in I$ and player 2 (the adversary) chooses a number $\mu \leq 2^{wd}$. If p divides μ, then player 2 wins, otherwise the next round begins. Player 1 wins if player 2 does not win in α rounds. This game models the behavior of algorithm RED, provided that no composite number is generated instead of a

prime. Namely, the probability for algorithm `RED` of being correct is lower bounded by the probability for player 1 of winning the game.

Let p_i and μ_i be the numbers chosen by the two players at round i. Let D_i be the event "player 2 does not win at round i". If player 2 did not win in rounds $1, \ldots, i-1$, the probability of D_i equals the probability that p_i does not divide μ_i. From the discussion above, we have $Pr\left\{D_i | \bigcap_{j=1}^{i-1} D_j\right\} \geq 1 - 1/(\sigma n^{c-2})$. The probability that player 1 wins is equal to $Pr\left\{\bigcap_{i=1}^{\alpha} D_i\right\}$, which is at least $1 - \alpha/(\sigma n^{c-2})$ by the chain rule of conditional probability.

We conclude by taking into account the probability for algorithm `RED` of generating at some round a composite number instead of a prime. By Lemma 1, the probability that all the α numbers are prime is at least $1 - \alpha/n^{\gamma}$, for any constant $\gamma > 0$. Hence, algorithm `RED` is correct with probability larger than or equal to $(1 - \alpha/(\sigma n^{c-2}))(1 - \alpha/n^{\gamma})$. Since $\alpha \leq \delta$ is polynomial in n, by appropriately choosing values c and γ the correctness probability can be made larger than $1 - 1/n^{\beta}$, for any constant $\beta > 0$. ◀

It is not difficult to see that the space usage of algorithm `RED` is $\Theta(nm)$ when $m = \Omega(\delta)$, and $\Theta(n\delta)$ otherwise. Lemma 3 addresses the running time of the algorithm.

▶ **Lemma 3.** *The worst-case running time of algorithm* `RED` *is* $O(nm + \alpha\delta^2)$*, where* $\alpha \leq \delta$ *is the actual number of memory faults occurring during the execution.*

Proof. Let us distinguish between *successful* and *unsuccessful block computations*. Unsuccessful block computations account for the time spent by the algorithm computing blocks that are then discarded due to the detection of a memory fault. This time also includes the generation of random primes, except for the first one. Successful block computations account for the remaining time, including the calculation of fingerprints.

Successful computations. Computing the first column of a block requires constantly-many reliable reads for each entry, i.e., $O(\delta^2)$ time. The same bound holds for the last column, that is written reliably. Computing any internal column requires instead $O(\delta)$ time, including the time to compute fingerprints incrementally. Since there are $O(\delta)$ columns in a block, the total time spent in a block is $O(\delta^2)$. The overall time for successful block computations is thus $O(nm)$, because the number of blocks is $\lceil n/\delta \rceil \times \lceil m/\delta \rceil$.

Unsuccessful computations. Each block recomputation is due to a fingerprint mismatch, that can only be caused by a memory fault (either in the matrix cells or in some input symbol from string X). Since all block cells are recomputed and, if necessary, the input symbols are refreshed reading their values reliably, each block recomputation can be charged to a distinct memory fault. It follows that at most α block computations can be discarded during the entire execution of algorithm `RED`. Refreshing δ input values and computing the block take time $O(\delta^2)$, which implies an overall time $O(\alpha\delta^2)$ for unsuccessful computations. The generation of (at most α) prime numbers does not affect this asymptotic running time (see Lemma 1). ◀

4 Error recovery via long distance fingerprints

Using a one-level decomposition of the dynamic programming table M into squares of side length δ yields an algorithm with an additive term $O(\alpha\delta^2)$ in the running time, due to recovery from errors (a single error determines the complete recomputation of a $\delta \times \delta$ block). In this section we show how to decrease this time to $O(\alpha\delta^{1+\varepsilon})$, for any arbitrarily small constant $\varepsilon \in (0, 1]$. The improved algorithm uses an asymmetric decomposition (see Figure 1c)

and $k = \lceil 1/\varepsilon \rceil$ different *resiliency levels*. At each level $i \in [1, k]$, it relies on $\lceil \delta^{i/k} \rceil$-resilient variables. To simplify the notation, we define $\delta_i = \lceil \delta^{i/k} \rceil$.

Consider a given $\delta \times \delta$ block B. Every δ_i columns, we write a δ_i-resilient column (and all δ_j-resilient versions of this column for $j < i$). In particular, the last column of each internal block is written at all resiliency levels. The non-resilient columns of matrix M are regarded as having resiliency level 0. During the computation of block B, for each resiliency level i we keep (in the private memory) the fingerprint of the last δ_i-resilient column. These long distance fingerprints, similarly to those described in Section 3, are computed while writing column values. For each resiliency level, we independently select a prime number for computing the fingerprints.

Upon detection of a fault, error recovery is done starting from the last δ_1-resilient column. Values in this column are read by majority; read values are used to recompute the fingerprint at level 1 which is then compared with the one stored in the private memory. If these fingerprints do not match, the recovery starts again from resiliency level 2, i.e., from the last δ_2-resilient column. In general, a level i fingerprint mismatch induces a recovery starting from the last δ_{i+1}-resilient column. When a fingerprint mismatch arises at resiliency level i, we generate a new random prime for level i, we read by majority all values of the last δ_{i+1}-resilient column (i.e., we perform δ read operations at resiliency level $i+1$), and we use these values to refresh all δ_j-resilient versions of this column, for $j \leq i$, recomputing their respective fingerprints.

Now consider the input symbols. As in Section 3, symbols from string Y are always read reliably, while symbols from X are read reliably only once, at the beginning of a block computation, and then verified by means of fingerprints. We store δ_i-resilient copies of the symbols in X at all resiliency levels. During the computation of a block, for each resiliency level (including level 0), we keep one fingerprint for the segment of X of length δ involved in that computation. All these fingerprints are obtained using independently selected prime numbers and are computed at the beginning of the block computation by reading reliably the input segment. Once a fingerprint mismatch on the input symbols is detected at level i, the δ_{i+1}-resilient copy of the input segment is used to refresh all copies at level $j \leq i$ (the previously computed fingerprint for resiliency level $i+1$ allows it to check the correctness of the read values). A new random prime for level i is then selected and the fingerprints for all refreshed levels are recomputed. Notice that a fingerprint mismatch at level 0 may arise during block computation, while a mismatch at level $i > 0$ can only arise during error recovery. Once the input symbols are correctly refreshed, normal computation is resumed by recomputing the current column.

▶ **Theorem 4.** *Let ε be an arbitrarily small constant in $(0, 1]$. The edit distance between two strings of length n and m, with $n \geq m$, can be correctly computed, with high probability, in $O\left(nm + \alpha\delta^{1+\varepsilon}\right)$ worst-case time and $O(nm + n\delta)$ space, when δ is polynomial in n.*

Proof. The correctness of the improved version of algorithm `RED` follows from Lemma 2 (details are deferred to the extended version of this paper). Similarly to the proof of Lemma 3, we analyze the running time by distinguishing between successful and unsuccessful computations. The asymptotic running time of successful computations is not affected by the additional $O(1/\varepsilon)$ fingerprints and by the δ_i-resilient columns. Now consider the unsuccessful computations. Recovery at resiliency level i discards at most $\delta \times \delta^{i/k}$ entries of table M and requires computing $O(\delta)$ majority values. Each δ_i-resilient read takes time $O(\delta^{i/k})$, thus yielding total $O(\delta^{1+i/k})$ time. A recovery at resiliency level i is due to at least $\delta^{(i-1)/k} + 1$ errors, either on the input symbols or in table M. Errors can be propagated by the algorithm (during both refresh operations and forward block computations) only if a fingerprint test

fails, which is a low probability event. Hence, a fingerprint mismatch at resiliency level $i-1$ may only arise if the majority value of some δ_{i-1}-resilient cell or input symbol has been corrupted by the adversary. This gives an amortized time per fault of $O(\delta^{1+1/k})$, which proves the theorem since $k = \lceil 1/\varepsilon \rceil$. ◀

Theorem 4 implies that, whenever $m = \Theta(n)$, algorithm RED matches the time and space complexity of the standard non-resilient implementation while tolerating almost linearly-many memory faults; specifically, up to $\delta = O\left(n^{2/(2+\varepsilon)}\right)$ faults. Notice that saving reliably the input strings requires time and space $\Omega(n\delta)$. Hence, any resilient algorithm which tolerates $\delta = \omega(n)$ memory faults must have time and space complexity $\omega(n^2)$. Hence, under the assumption that the time and space complexity of the standard non-resilient implementation cannot be exceeded, algorithm RED tolerates an almost optimal number of faults. If $O(\log \delta)$ private memory words are available (similarly to [3, 8]), the time bound given in Theorem 4 drops to $O(nm + \alpha\delta)$, thus allowing to tolerate an optimal linear number of faults.

5 Resilient traceback

Once table M has been computed, an optimal edit sequence transforming string X into string Y can be obtained by computing a traceback path from entry $M[n, m]$ to $M[0, 0]$. The predecessor of an entry $M[i, j]$ on the traceback path can be any of the neighboring entries $M[i-1, j]$, $M[i, j-1]$, and $M[i-1, j-1]$, satisfying Equation 1. It will be convenient to assume that there is an arc from $M[i, j]$ to its predecessor: the cost of the arc is 0 if $x_i = y_j$, and 1 otherwise. We define the cost of a traceback path as the sum of the costs of its arcs: this corresponds to the edit distance of X and Y. We now describe how the traceback process can be made resilient.

The computation proceeds backward block by block, starting from cell $M[n, m]$. Within each block traversed by the traceback path, we compute the corresponding subsequence S of the whole edit sequence, writing it reliably. Data replication on the resilient block boundaries and on the input symbols allows, once S is computed, to check whether some error occurred during the backward computation. In order to recover from an error at this point, we could recompute first the block involved (by applying algorithm RED), and then subsequence S. This would result in an additive overhead $O(\alpha\delta^2)$ on the total running time. To bound this cost by $O(\alpha\delta^{1+\varepsilon})$, we do not stick at computing each subsequence S reliably since the beginning, but proceed incrementally starting from resiliency level 1 up to $k = \lceil 1/\varepsilon \rceil$. To this aim, we exploit the δ_i-resilient columns written during the forward computation of matrix M. The fact that column fingerprints are no longer available makes the task harder.

We regard each subsequence S as being divided into (at most) $\delta^{1/k}$ segments, computed at resiliency level $k-1$. This subdivision proceeds hierarchically, down to resiliency level 1. As a base step, segments at resiliency level 0 are computed from the cells of matrix M: these segments correspond to single arcs of the traceback path. A segment S_i, at resiliency level i, spans two δ_i-resilient columns (the right/left column, in some cases, can be replaced by the bottom/top resilient row of the block). S_i is computed by combining all the $\delta^{1/k}$ sub-segments at resiliency level $i-1$ in which S_i is logically divided. Sub-segments are read, proceeding right to left, δ_{i-1}-resiliently, and their soundness is verified against the corresponding input symbols, which are read δ_i-resiliently: we call this a *consistency check* (the δ_i-resilient reads on the input symbols from string Y are performed on the first $2\delta_i + 1$ elements of the δ-resilient copy of Y). During this process, S_i is also written δ_i-resiliently.

If a consistency check fails at a given cell c, the input symbols corresponding to the row and column of c are refreshed and, if either endpoint of S lies on a resilient row, the

corresponding cell is also refreshed. The recovery then starts from the closest δ_i-resilient column to the left of c: all δ_j-resilient versions of this column, for $j < i$, are refreshed from the δ_i-resilient values (read by majority) and the block slice is recomputed by applying the improved version of algorithm RED. At the end of the slice computation, we check if the new values stored on the closest δ_i-resilient column to the right of c match the old ones: if this is not the case, recovery restarts at resiliency level $i+1$. At the end of the recovery phase, the computation of S_i restarts from sub-segments at resiliency level 1.

When the computation of a segment S_i is completed, the algorithm checks if the cost of the segment matches the difference between the cell values in matrix M corresponding to its endpoints. These cells lie on δ_i-resilient columns (or on resilient rows on the block boundaries) and their values are read δ_i-resiliently. Apart from refreshing the input values, upon detection of a mismatch in the edit sequence cost, recovery is performed as described above.

▶ **Theorem 5.** *Given table M computed by algorithm* RED*, an edit sequence of cost $M[n,m]$, if any, can be computed with high probability in time $O\left(n\delta + \alpha\delta^{1+\varepsilon}\right)$.*

Proof. The correctness of all edit operations is verified at all resiliency levels by consistency checks. Moreover, the edit cost of each segment is always verified against the edit distance. Memory faults from any resiliency level i are never propagated to higher levels of resiliency. Indeed, during error recovery, no δ_{i+j}-resilient value is modified starting from δ_i-resilient reads, for any $j \geq 0$. This implies that a segment at resiliency level i may be wrong only if at least $\delta^{i/k}+1$ memory faults occurred. Since the adversary can insert at most δ faults, the δ-resilient edit sequence, if constructed, is correct and has cost $M[n,m]$. This sequence may not be optimal or the traceback algorithm may not be able to reconstruct it only if a fingerprint test failed to detect a memory fault, which is a low probability event (see Lemma 2). We now focus on the running time, distinguishing between successful and unsuccessful segment computations.

Successful computation. The time spent to combine all δ_{k-1}-resilient segments is asymptotically higher than the time spent at all lower resiliency levels. This time is $O(n\delta)$, because the edit sequence traverses $O((n+m)/\delta)$ blocks, and each block costs time $O(\delta^2)$.

Unsuccessful computation. Consider a consistency check failure arising while computing a segment at resiliency level $i+1$. Such a failure is due to at least $\delta^{i/k}+1$ faults and costs $O(\delta^{1+(i+1)/k})$ time for recomputing $\delta \times \delta^{(i+1)/k}$ matrix cells. If the δ_{i+1}-resilient column used for recovery was correct, detected errors are removed from the matrix with high probability, with an amortized $O(\delta^{1+1/k})$ cost per memory fault. If the δ_{i+1}-resilient column used for recovery was corrupted, the adversary must have inserted at least $\delta^{(i+1)/k}+1$ faults and the recomputed cells of the matrix may still contain incorrect values after recovery. Two cases may happen: either the forward recomputation of the matrix slice finds an inconsistency with the following δ_{i+1}-resilient column, or no inconsistency is detected and a possibly wrong δ_{i+1}-resilient segment is computed. In both cases, the number of memory faults inserted by the adversary is large enough to obtain an amortized $O(\delta^{1+1/k})$ cost per fault, with recovery done at a higher resiliency level. ◀

6 Extensions

Reducing space. Hirschberg proposed a technique to compute an optimal edit sequence in time $O(nm)$ using only linear space [13]. In our model, $\Omega(n\delta)$ space is required for storing the input reliably. This is better than $\Theta(nm)$ when $\delta = o(m)$. We now show that this bound

can be achieved by adapting Hirschberg's technique to work in faulty memories. Hirschberg's algorithm is recursive. In the first step, it computes the edit distances between the first half of string X and all prefixes of string Y, and between the remaining half of X reversed and all prefixes of Y reversed. It then finds an optimal point to split Y, constructs a segment of the optimal edit sequence, and recursively solves two smaller subproblems. We use algorithm `RED` to obtain the edit distances on the forward and reversed substrings: since no traceback is required, we discard a block as soon as all its neighboring blocks have been processed. The space usage is thus $O(n\delta)$. The split point can be computed reliably in $O(m\delta)$ time. Each recursive call pushes on the stack only $O(1)$ variables, that are stored and reloaded reliably. We end each branch of the recursion when the subproblem matrix has size bounded by $\delta \times \delta$, i.e., fits in a single block. It can be proved that this algorithm computes an optimal edit sequence resiliently in time $O(nm + \alpha\delta^{1+\varepsilon})$ and optimal space $\Theta(n\delta)$.

Taking advantage of string similarity. Ukkonen proved that an optimal edit sequence can be computed in time and space $O(e\min\{m,n\})$, where e is the edit distance between the input strings [21]. Since $e \geq n - m$, this improves over the standard dynamic programming algorithm only when $m = n - o(n)$, which implies $\min\{m,n\} = m = \Theta(n)$. The main idea is to assume that the edit distance e is upper bounded by a small value k and to compute only $\Theta(k)$ diagonals of matrix M. If no edit sequence of cost $\leq k$ exists, k is doubled and the computation is repeated. In the resilient implementation, we avoid considering blocks that have empty intersection with the set of diagonals that have to be computed in the current iteration. This results in a time and space complexity $O(n\max\{e,\delta\})$, matching the result from Ukkonen when $\delta = O(e)$. If $\delta = \omega(e)$, the $O(n\delta)$ time and space bounds match those required to handle the input reliably.

Local dependency dynamic programming. In the description of algorithm `RED` we exploit no specific properties of the edit distance problem. Instead, the analysis benefits from a few structural properties of the dynamic programming recurrence relation, that are also typical of many other problems. The techniques described in this paper can be applied to a variety of problems that can be solved via dynamic programming and, in particular, to local dependency dynamic programming problems, where each update to an entry in the auxiliary table is determined by the contents of the neighboring cells. The framework in which our technique can be applied successfully can be described as follows.

Let us consider a generic d-dimensional dynamic programming algorithm, for any constant $d \geq 2$. Assume that the problem input consist of d sequences $S_1, \ldots, S_d$, each of length n. The sequences are not necessarily distinct and the description can be easily generalized to deal with different lengths. The algorithm computes an auxiliary table M of dimension $(n+1)^d$. Each cell $M[i_1, \ldots, i_d]$, with $0 \leq i_1, \ldots, i_d \leq n$, is computed using a recurrence relation. In particular, if any index is equal to 0, then $M[i_1, \ldots, i_d]$ is initialized with a value that depends only on $i_1, \ldots, i_d$. Otherwise, $M[i_1, \ldots, i_d]$ is recursively computed from the values of the $2^d - 1$ neighboring cells, where a cell $M[j_1, \ldots, j_d]$ is a neighbor of a distinct cell $M[i_1, \ldots, i_d]$ if, for each dimension h, either $j_h = i_h - 1$ or $j_h = i_h$. Besides the neighboring cells, the computation of $M[i_1, \ldots, i_d]$ can also use d input symbols, i.e., the i_h-th symbol from sequence S_h for $1 \leq h \leq d$. We assume that the table is computed according to a fixed regular pattern (e.g., along rows, columns, or diagonals when $d = 2$), and that $M[n, \ldots, n]$ contains the solution. Using blocks of dimension δ^d, we can generalize our approach obtaining the following result:

▶ **Theorem 6.** *Let ε be an arbitrarily small constant in $(0,1]$. A d-dimensional local dependency dynamic programming table M of size n^d can be correctly computed, with high*

probability, in $O(n^d + \alpha\delta^{d-1+\varepsilon})$ *worst-case time and* $O(n^d + n\delta)$ *space, when* δ *is polynomial in* n*. Tracing back can be done with high probability in additional time* $O(n\delta + \alpha\delta^{d-1+\varepsilon})$.

This yields resilient algorithms for d-dimensional problems that have the same running time as the non-resilient implementations and can tolerate with high probability $O(n^{d/(d+\varepsilon)})$ memory faults.

References

1. M. Ajtai, V. Feldman, A. Hassidim, and J. Nelson. Sorting and selection with imprecise comparisons. In *ICALP (1)*, pages 37–48, 2009.
2. Y. Aumann and M. A. Bender. Fault tolerant data structures. In *FOCS*, pages 580–589, 1996.
3. M. Blum, W. S. Evans, P. Gemmell, S. Kannan, and M. Naor. Checking the correctness of memories. *Algorithmica*, 12(2–3):225–244, 1994.
4. R. S. Boyer and J. S. Moore. Mjrty: A fast majority vote algorithm. In *Automated Reasoning: Essays in Honor of Woody Bledsoe*, pages 105–118, 1991.
5. G. S. Brodal, A. G. Jørgensen, and T. Mølhave. Fault tolerant external memory algorithms. In *WADS*, pages 411–422, 2009.
6. G. S. Brodal, A. G. Jørgensen, G. Moruz, and T. Mølhave. Counting in the presence of memory faults. In *ISAAC*, pages 842–851, 2009.
7. V. Chen, E. Grigorescu, and R. de Wolf. Efficient and error-correcting data structures for membership and polynomial evaluation. In *STACS*, pages 203–214, 2010.
8. M. Chu, S. Kannan, and A. McGregor. Checking and spot-checking the correctness of priority queues. In *ICALP*, pages 728–739, 2007.
9. R. de Wolf. Error-correcting data structures. In *STACS*, pages 313–324, 2009.
10. I. Finocchi, F. Grandoni, and G. F. Italiano. Optimal resilient sorting and searching in the presence of memory faults. *Theor. Comput. Sci.*, 410(44):4457–4470, 2009.
11. I. Finocchi, F. Grandoni, and G. F. Italiano. Resilient dictionaries. *ACM Transactions on Algorithms*, 6(1), 2009.
12. I. Finocchi and G. F. Italiano. Sorting and searching in faulty memories. *Algorithmica*, 52(3):309–332, 2008.
13. D. S. Hirschberg. A linear space algorithm for computing maximal common subsequences. *Commun. ACM*, 18(6):341–343, 1975.
14. B. L. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann, 2008.
15. A. G. Jørgensen, G. Moruz, and T. Mølhave. Priority queues resilient to memory faults. In *WADS*, pages 127–138, 2007.
16. R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBM J. Res. Dev.*, 31(2):249–260, 1987.
17. F. T. Leighton, Y. Ma, and C. G. Plaxton. Breaking the $\theta(n \log^2 n)$ barrier for sorting with faults. *J. Comput. Syst. Sci.*, 54(2):265–304, 1997.
18. A. Pelc. Searching games with errors - fifty years of coping with liars. *Theor. Comput. Sci.*, 270(1–2):71–109, 2002.
19. M. O. Rabin. Probabilistic algorithm for testing primality. *J. of Number Th.*, 12(1):128–138, 1980.
20. B. Schroeder, E. Pinheiro, and W. D. Weber. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS/Performance*, pages 193–204, 2009.
21. E. Ukkonen. Algorithms for approximate string matching. *Information and Control*, 64(1–3):100–118, 1985.