Theoretical Computer Science 412 (2011) 3537–3544

Contents lists available at ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

Approximate string matching with stuck address bits☆

Amihood Amir[a,d], Estrella Eisenberg[a], Orgad Keller[a], Avivit Levy[b,c,*], Ely Porat[a]

[a] Department of Computer Science, Bar Ilan University, Ramat-Gan 52900, Israel
[b] Department of Software Engineering, Shenkar College, 12 Anna Frank, Ramat-Gan, Israel
[c] CRI, University of Haifa, Mount Carmel, Haifa 31905, Israel
[d] Department of Computer Science, Johns Hopkins University, Baltimore, MD 21218, United States

ARTICLE INFO

Article history:
Received 8 November 2010
Received in revised form 24 February 2011
Accepted 25 February 2011
Communicated by M. Crochemore

Keywords:
Pattern matching
String rearrangement metrics
Approximate string matching
Address errors

ABSTRACT

A string $S \in \Sigma^m$ can be viewed as a set of pairs $\{(s_i, i) \mid s_i \in S,\ i \in \{0, \ldots, m-1\}\}$. We follow the recent work on *pattern matching with address errors* and consider approximate pattern matching problems arising from the setting where errors are introduced to the location component (i), rather than the more traditional setting, where errors are introduced to the content itself (s_i). Specifically, we continue the work on string matching in the presence of address bit errors. In this paper, we consider the case where bits of i may be stuck, either in a consistent or transient manner. We formally define the corresponding approximate pattern matching problems, and provide efficient algorithms for their resolution.

1. Introduction

Background. Over 30 years ago one of the co-authors of this paper was busy writing a program that points an antenna to a given moving location. Having written a program that converts latitude and longitude to the appropriate azimuth, taking all geodesic information into consideration, the program was finally tested.

The frustrated programmer noticed that the antenna was pointing to the west, when it was supposed to point north. In those days, de-bugging meant halting the computer and looking at the memory contents through a panel register. The programmer halted the program after it loaded the bus with the azimuth and immediately prior to giving the device the signal to load the azimuth, and checked the value. To his surprise, the value matched his calculations. He then resumed running the program and the antenna pointed exactly to the required direction. However, running the program from beginning to end again achieved a wrong result.

The problem was that some bits on the bus settled on their value faster than others, thus when those bits had a 0 value and the value was changed to 1, it took longer to settle than when a 1 was changed to a 0, or when the value was not changed. A short wait helped.

1.1. Pattern matching with address errors

Motivation. An important implicit assumption in the traditional view of pattern matching was that there may indeed be errors in the *content* of the data, but the *order* of the data is inviolate. Consider a text $T = t_0 \ldots t_{n-1}$ and pattern

☆ A proceedings version of this paper appeared in SPIRE 2010.

* Corresponding author at: Department of Software Engineering, Shenkar College, 12 Anna Frank, Ramat-Gan, Israel. Tel.: +972 9 8627119.
E-mail addresses: amir@cs.biu.ac.il (A. Amir), kellero@cs.biu.ac.il (O. Keller), avivitlevy@gmail.com, avivitlevy@shenkar.ac.il (A. Levy), porately@cs.biu.ac.il (E. Porat).

0304-3975/$ – see front matter © 2011 Elsevier B.V. All rights reserved.
doi:10.1016/j.tcs.2011.02.044

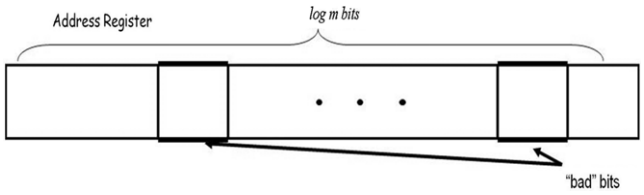

Fig. 1. Failures in the address bus due to 'bad' bits cause wrong addresses to be stored in the address register.

$P = p_0 \dots p_{m-1}$, both over an alphabet Σ. Traditional pattern matching regards T and P as *sequential* strings, provided and stored in sequence (e.g., from left to right). However, some non-conforming problems have been gnawing at the basis of this assumption. Selected examples are:

Text editing: The *swap* error, motivated by the common typing error where two adjacent symbols are exchanged [7,21], does not assume error in the content of the data, but rather, in the order. The data content is, in fact, assumed to be correct. The swap error seemed initially to be akin to the other Levenshtein errors, in that it could be added to the other edit operations and solved with the same dynamic programming [21]. However, when isolated, it turned out to be surprisingly simple to handle [10]. This scarcely seems to be the case for indels or mismatch errors.

Computational biology: During the course of evolution areas of the genome may be shifted from one location to another. Considering the genome as a string over the alphabet of genes, these cases represent a situation where the difference between the original string and resulting one is in the locations rather than contents of the different elements. Several works have considered specific versions of this biological setting, primarily focusing on the sorting problem (*sorting by reversals* [12,13], *sorting by transpositions* [11], and *sorting by block interchanges* [15]).

Bit torrent and video on demand: The inherently distributed nature of the web is already causing the phenomenon of transmission of a stream of data in tiny pieces from different sources. This creates the problem of putting scrambled data back together again [14].

Computer architecture: In computer architecture, it is by no means taken for granted that when seeking a word from a given address, no errors will occur in the address bits [18]. This problem is relevant even when reading a buffer of consecutive words since these words are not necessarily consecutive in the disk or in an interleaved cache.[1]

Therefore, the traditional view of strings is becoming, at times, too restrictive.

The model. In such cases, it is more natural to view the string as a set of pairs (σ, i), where i denotes a location in the string, and σ is the value appearing at this location. Given this view of strings, the problem of *approximate pattern matching* has been reconsidered in the last few years, and a new pattern matching paradigm – *pattern matching with address errors* – was proposed in [2]. In this model, the pattern *content* remains intact, but the relative positions (addresses) may change. Efficient algorithms for several different natural types of rearrangement errors were presented [3,4,8,20] (see also [9]). These types of address errors were inspired by biology, i.e., pattern elements exchanging their locations due to some external process.

Address bit errors. Another broad class of address errors inspired by computer architecture was studied in [1,6]. They consider errors which arise from a process of flipping some or all of the bits in the binary representation of $[1, m]$. Such errors represent situations where the text and the pattern are generated by two different systems, which may use different naming conventions. The error processes are inspired by address errors resulting from failures in the wires of the address bus, the wires connecting the CPU and the memory which are used to transmit the address of operands (see Fig. 1), or failure in the transmitted address bits. The errors handled by [1,6] were all bi-directional, jogging the memory of our programmer. Discussions with old cronies who, over the years, continued grappling with parallel transmissions over wires, resulted in the desire to study the situation where the "badness" of the bits means being "stuck" on a value, rather than changing it.

Our contribution. This paper follows the work of [6], but studies the situation of address bit errors caused in the presence of stuck bits (defined below), that was not considered by previous work. The contributions of this paper are two-fold:

1. to enhance the nascent body of work on pattern matching with rearrangements. In particular this paper requires a non-trivial use of network flow to solve one version of our problem. This is definitely not a technique in the traditional Pattern Matching toolkit.
2. to continue the study of pattern matching under address bit errors that was begun in [5]. This paper is still only a beginning. Discussions with practitioners suggest further directions, as will be presented in Section 4.

1.2. Problem definition

Consider a string $S \in \Sigma^m$. Using the definition of [5], the string is regarded as a set of pairs, $S = \{(\sigma, i) \mid \sigma \in \Sigma,\ i \in \{0, 1\}^{\log m}\}$. We consider two types of errors in the bits of the i entries:

[1] Practically, these problems are solved by means of redundancy bits, checksum bits, error detection and correction codes, and communication protocols.

Stuck bits. There exists a subset of bit positions $F \subseteq \{0, \ldots, \log m - 1\}$, such that in each i, all bits in positions $f \in F$ are either always changed to zero (i.e. 1 is turned into a 0 and 0 remains 0) or always changed to one (i.e. 0 is turned into a 1 and 1 remains 1).
For example, for the string $S = 1234 = \{(1, 00), (2, 01), (3, 10), (4, 11)\}$ and $F = \{0\}$, a resulting string is $S' = \{(1, 00), (2, 01), (3, 00), (4, 01)\}$.

Transient stuck bits. There exists a subset of bit positions $F \subseteq \{0, \ldots, \log m - 1\}$, such that in each i, the bits in positions $f \in F$ *may* remain unchanged, or may be changed to a "1" (of course the original string changes only if the intention was to output a "0").
As an example, for the string $S = 1234 = \{(1, 00), (2, 01), (3, 10), (4, 11)\}$ and $F = \{0\}$, the resulting string may be $S' = \{(1, 10), (2, 01), (3, 10), (4, 11)\}$ (the bit was changed to one for address 1 but not for address 2).

Note that the resulting set is actually a multi-set, and may not represent a valid string, as some locations may appear multiple times, while others not at all.

We consider approximate pattern matching problems associated with each of the above types of errors. Specifically, given a pattern P and text T, we wish to find:

- the smallest set F such that if the bits of F are consistently stuck, then P has a match in T. We call this problem the *stuck bits* problem.
- the smallest set F such that if the bits of F may be transiently stuck, then P has a match in T. We call this problem the *transient stuck bits* problem.

Following [5], we focus on developing efficient solutions for the case that the text and the pattern are both of length m. We discuss the situation of text longer than pattern in Section 4.

1.3. Our results

We provide the following results:

- an $O(m \log m)$ time solution for pattern matching with stuck bits, which also reports the stuck bits positions, where m is the length of both text and pattern.
(Theorem 1)
- a simple $O(m^{2.5})$ time solution for pattern matching with transient bits, which also reports the stuck bits positions. This algorithm is based on a reduction to finding perfect matching in a bipartite graph.
(Corollary 2)
- a flow-based $O(m^{2.2156} \log^2 m)$ time solution for pattern matching with transient bits, which also reports the stuck bits positions.
(Theorem 2)

Paper organization. The rest of the paper is organized as follows. In Section 2 we study the stuck bits problem and prove Theorem 1. In Section 3 we study the transient stuck bits problem and prove Corollary 2 and Theorem 2.

2. The stuck bits problem

The nature of the stuck bits problem, as opposed to the flipped bits problem of [5], is that a stuck bit necessarily *deletes* addresses and creates addresses with multiple symbols. We show below that it is possible to not only compute the number of stuck bits by a considering the address sets, but also to easily compute the stuck bits' positions.

Let T be a length-m text and $P = \{(\sigma, i) \mid \sigma \in \Sigma,\ \ i \in \{0, 1\}^{\log m}\}$ be a length-m pattern. Define $IP = \{i \mid \exists \sigma,\ (\sigma, i) \in P\}$ to be the set of character positions given in P.

Observation 1. *Assume m is a power of 2. Let n be the number of stuck bits and $\ell = \log m - n$. Then $|IP| = 2^{\ell}$.*

Algorithm StuckBits(P) below constructs the set IP from input P, and outputs a binary string k of length $\log m$, where $k[i] = 0$ if i is a stuck bit, and $k[i] = 1$ otherwise. The algorithm uses the boolean operator $\oplus$ – the *exclusive or* operation. Specifically, $a \oplus b$ is 0 if $a = b$ and 1 if $a \neq b$, for $a, b \in \{0, 1\}$. The definition below extends the boolean operation to strings in the natural manner.

Definition 1. Let $s, t \in \{0, 1\}^{\ell}$ i.e. $s = s[1], \ldots, s[\ell]$, $t = t[1], \ldots t[\ell] s[i], t[i] \in \{0, 1\}, i = 1, \ldots, \ell$, and let $\oplus$ be a boolean operator, exclusive or.

Define $s \oplus t$ as:

$$(s \oplus t)[i] = s[i] \oplus t[i]$$

for $i = 1, \ldots, \ell$.

Algorithm 1: StuckBits(P)

let k be a $\log m$-length bit-vector;
let i_0 be the lexicographic minimum i such that $i \in IP$;
if $\nexists j \in IP, \quad j \neq i_0$ **then return** $1^{\log m}$;
foreach $j \in IP$ *such that* $j \neq i_0$ **do**
    $k_j \leftarrow j \oplus i_0$;
end foreach
$k \leftarrow \bigwedge_{j \neq i_0} k_j$;
return k;

Example 1. Given the pattern $P = \{(1, 00), (2, 01), (3, 00), (4, 01)\}$, algorithm StuckBits calculates $(00 \oplus 01)$, and therefore, returns 01, where the 0 indicates a position of a stuck bit. We conclude that in this case the most significant bit is a stuck bit.

Lemma 1. *Every location in k – the output vector of StuckBits(P) – that equals zero, is a stuck bit.*

Proof. Note that, a bit i is stuck if and only if every location i has the same value (zero or one) in all addresses (second component) of P if and only if the exclusive or of all addresses of P has a zero in location i. □

2.1. Pattern matching under stuck bits errors

For two addresses $i, j \in \{0, \ldots, m\}$, we say that i *is equivalent to* j |it under possible stuck bits, and write $i \equiv j$, if all bits where i differs from j are stuck bits. Formally, $i \equiv j$ if and only if $(i \oplus j) \wedge \text{StuckBits}(P) = 0^{\log m}$. The following algorithm decides if there exists a set of bits that, if stuck, cause text T to become pattern P. In this case we say that P *matches* T *under stuck bits errors*. The idea of the algorithm is to gather all text symbols in locations whose addresses are indistinguishable due to the stuck bits, and compare these sets to the sets provided by the pattern. The algorithm works efficiently since Lemma 1 allows us to identify the locations of the stuck bits.

Algorithm 2: StuckMatch(T, P)

foreach $i \in IP$ **do**
    $B_i^P \leftarrow \{\sigma \mid (\sigma, i) \in P\}$;
    sort B_i^P;
    $B_i^T \leftarrow \{T[j] \mid j \equiv i\}$;
    sort B_i^T;
end foreach
$B \leftarrow \bigwedge_{i \in IP}(B_i^P = B_i^T)$;
return B;

Example 2. Given the pattern $P = \{(A, 000), (B, 001), (C, 000), (D, 001), (A, 000), (E, 001), (F, 000), (G, 001)\}$, algorithm StuckBits($P$) returns 101, means that the stuck bit is the second bit (0's for stuck bits). In this case,

$T = \langle A, B, C, D, A, E, F, G\rangle$,
$B_{000}^P = \langle A, A, C, F\rangle$,
$B_{001}^P = \langle B, D, E, G\rangle$,
$B_{000}^T = \langle A, A, C, F\rangle$, and
$B_{001}^T = \langle B, D, E, G\rangle$.

Therefore, algorithm StuckMatch(T, P) returns 1, i.e., there is a match.

2.2. Total time and space of the stuck bits problem

We obtain the following:

Theorem 1. StuckMatch(T, P) *can be solved in $O(m \log m)$ time, where m is the length of both text and pattern. For finite alphabets, or alphabet $\{1, \ldots, m\}$,* StuckMatch(T, P) *can be solved in linear time. The space complexity is: $\Theta(m \log m)$.*

Proof. Correctness follows from the above discussion. For the time complexity, we assume constant time operations on words of size $O(\log m)$ bits. Finding the stuck bits requires $\Theta(m)$ time. The B_i^P and B_i^T can be constructed in time $\Theta(m)$ as well. Sorting each of the B_i^P and B_i^T can be done in time $O(m \log m)$ in general, and by bucket sort for finite alphabets or alphabet $\{1, \ldots, m\}$. Finally, B is calculated in $\Theta(m)$ time. We conclude that the overall time is $O(m \log m)$, and linear in the case of finite alphabets, or alphabet $\{1, \ldots, m\}$.

It remains to show the space complexity. The pattern P and the text T are of space: $\Theta(m \log m)$, since we have m symbols and for each symbol an address of length $\log m$. The *StuckBits* algorithm does *xor* between pairs of addresses, it takes $\log m$ space. The *StuckMatch* algorithm sorts groups of size $\Theta(m)$, which take $O(1)$ extra space, and do *and* between $\Theta(m)$ boolean numbers, which take $O(1)$ extra space. The total space is, therefore: $\Theta(m \log m)$. □

3. Transient stuck bits problem

Similarly to the flipped bit problem of [5], the first step is comparing the histogram of characters in the text and pattern, i.e., for each alphabet symbol σ, the number of occurrences of σ in T and P needs to be equal, otherwise there can be no matching. Assume, therefore, that the histograms match.

In this section we use different tools as *bipartite graphs, maximal matching in a bipartite graph, flow network* and *maximal flow*. We, therefore, begin with definitions of these notions.

3.1. Background: bipartite graphs and flows

All definitions in this subsection are taken from [16].

Definition 2. A **bipartite graph** is an undirected graph $G = (V, E)$ in which V can be partitioned into sets V_1 and V_2 such that $(u, v) \in E$ implies either $u \in V_1$ and $v \in V_2$ or $u \in V_2$ and $v \in V_1$. That is, all edges go between the two sets V_1 and V_2.

Definition 3. Given an undirected graph $G = (V, E)$, a **matching** is a subset of edges $M \subseteq E$ such that for all vertices $v \in V$, at most one edge of M is incident on v. We say that a vertex $v \in V$ is *matched* by matching M if some edge in M is incident on v; otherwise, v is *unmatched*.

Definition 4. Given an undirected graph $G = (V, E)$, a **maximal matching** is a matching of maximum cardinality, that is, a matching M such that for any matching M', we have $|M'| \leq |M|$.

Definition 5. A **flow network** $G = (V, E)$ is a directed graph in which each edge $(u, v) \in E$ has a nonnegative *capacity* $0 \leq c(u, v)$. If $(u, v) \notin E$, we assume that $c(u, v) = 0$. We distinguish two vertices in the flow network: a *source* s and a **sink** t. We assume that every vertex lies on some path from the source to the sink. That is, for every vertex $v \in V$, there is a path $s \rightsquigarrow v \rightsquigarrow t$.

Definition 6. A **flow** in G is a real-valued function $f : V \times V \to \mathbf{R}$ that satisfies the following three properties:

- **Capacity constraint:** For all $u, v \in V$, we require $f(u, v) \leq c(u, v)$.
- **Skew symmetry:** For all $u, v \in V$, we require $f(u, v) = -f(v, u)$.
- **Flow conservation:** For all $u \in V - \{s, t\}$, we require $\Sigma_{v \in V} f(u, v) = 0$.

The quantity $f(u, v)$, which can be positive, zero or negative, is called the **flow** from vertex u to vertex v.
The **value** of a flow f is defined as: $|f| = \Sigma_{v \in V} f(s, v)$.

Definition 7. Given a flow network G with a source s and sink t, The **maximum flow problem** is to find a flow of maximum value.

3.2. Verifying the existence of a transient stuck bit matching

Let $T \in \Sigma^m$ and $P = \{(\sigma, i) \mid \sigma \in \Sigma,\ \ i \in \{0, 1\}^{\log m}\}$ of length m.

The following is a simple solution, reducing the problem to perfect bipartite matching. For a given $\sigma \in \Sigma$, we are seeking a bijection from the pattern pairs (σ, i) to the text locations where there are σ's, in a manner that if (σ, i) is matched to j then every bit location b that has a 1 in j has a 1 in i.

Such a bijection can be constructed via maximum perfect matching in the following bipartite graph:

Definition 8 (*The Bipartite Graph*)**.** Let $G_\sigma = (V_1, V_2, E)$ where each of V_1 and V_2 has m elements. The elements of V_1 are labeled by the pairs $(T[j], j)$, $j = 1, \ldots, m$. We label V_2 by the bijection $\ell : V_2 \to P$. Put an edge between node $v_1 \in V_1$ and node $v_2 \in V_2$ if their labels have the same symbols and if the address of v_1's label can be translated to the address of v_2's label via *transient stuck bit* errors. Formally, $E = \{(v_1, v_2)|$ where $v_1 = (\sigma_1, i),\ v_2 = (\sigma_2, j),\ \sigma_1 = \sigma_2$ and for every 1 in bit location b in i there is a 1 in bit location b in $j\}$.

Example 3. Given $T = ABAB$ and $P = \{(A, 01), (A, 10), (B, 11), (B, 11)\}$. Fig. 2 shows the bipartite graph constructed from the text and pattern as well as a maximal perfect matching.

Now all we need to do is verify if there is a perfect matching in G_σ. The Hopcroft and Karp algorithm [19] (denoted as *HopcroftKarp* in the pseudo-code below) finds the size of the maximum matching in time: $\Theta(E\sqrt{V_1 + V_2})$. In our case, since $V_1 = V_2 = m$ the time is $\Theta(E\sqrt{m})$, and in the worst case, $O(m^{2.5})$.

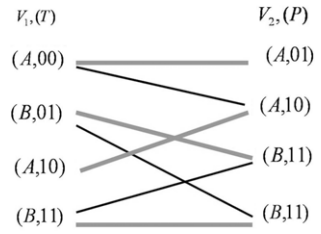

Fig. 2. The bipartite graph with a maximal matching (shown in the bold gray lines).

Algorithm 3: TransientBitsMatch(T, P)

```
1 Construct bipartite graph G_σ.
2 if |HopcroftKarp(G)| = m then return 1;
3 else return 0;
```

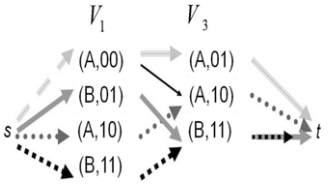

Fig. 3. Transient stuck bits flow-based solution.

3.3. Finding the stuck bits location

The remaining task is identifying the locations of the transient stuck bits.

Denote the sum of 1's in bit location b of the addresses in the text (V_1) as T_b, and the sum of 1's in bit location b of the addresses in the pattern (V_2) as P_b.

Consider node $v_1 \in V_1$ and assume there is a 1 in bit location b of its address. The node $v_2 \in V_2$ that was matched to v_1 by the perfect matching must have a 1 in bit location b of its address. Therefore, if the sum of 1's in bit location b of all nodes in V_1 (T_b) equals the sum of 1's in bit location b of all nodes in V_2 (P_b) then bit b cannot be a stuck bit.

On the other hand if $P_b > T_b$ then it means that b is sometimes stuck. Moreover, the number of times it is stuck is $P_b - T_b$.

It can never be the case that $T_b > P_b$ since then there is no possible perfect matching of size m.

Corollary 1. *The stuck bits are all the bit locations b where $P_b > T_b$ and the number of times it is stuck is $P_b - T_b$.*

Example 4. Given $T = ABAB$ and $P = \{(A, 01), (A, 10), (B, 11), (B, 11)\}$. We have: $T_0 = 2$ (the sum of bits in location 0 in T) and $P_0 = 3$ (the sum of bits in location 0 in P), which means that the bit in location 0 was stuck once. $T_1 = 2$ and $P_1 = 3$, which means that the bit in location bit 1 was also stuck once.

3.4. Faster verification of transient stuck bit matching

Assuming a transient stuck bit matching exists, Section 3.3 finds the location in time $O(m \log m)$. Thus the time complexity bottleneck is the transient stuck bits matching verification. In this subsection, we show a faster verification algorithm, based on network flow.

Let V_1 be as in Definition 8 and V_3 be the set of all *distinct* pairs $(\sigma, i) \in P$. Construct the following flow network.

Example 5. Given $T = ABAB$ and $P = \{(A, 01), (A, 10), (B, 11), (B, 11)\}$. Fig. 3 shows the flow-based solution constructed from this text and pattern.

Definition 9 (*The Flow Network*)**.** Let $G_{T,P} = (V, E)$ where $V = V_1 \cup V_3 \cup \{s\} \cup \{f\}$. s is the *source* and f is the *sink*.

E is constructed as follows. $\forall v \in V_1$ there is an edge $\overrightarrow{sv}$ (for every node v in V_1 there is an edge from the source to v). $\forall w \in V_3$ $\exists$*edge* $\overrightarrow{wf}$. For every $v = (\sigma_1, i) \in V_1$ and $w = (\sigma_2, j) \in V_3$, if $\sigma_1 = \sigma_2$ and for every 1 in bit location b in i there is a 1 in bit location b in j then there is an edge $\overrightarrow{vw}$. (This last condition is the same as that of Definition 8.)

We now define the edge capacities. Let $v = (\sigma, i) \in V_3$ and assume that v occurs c times in P. Then the capacity of edge $\overrightarrow{vf}$ is c. The capacity of every edge from the source to V_1 or from V_1 to V_3 is 1.

Lemma 2. *$G_{T,P}$ has a flow of value m if and only if there is a transient stuck bits matching between T and P.*

Proof. It is easy to see that a transient stuck bits matching defines a flow. Conversely, if there is a flow whose value is m, assign the nodes as defined by the flow. □

Max flow can be determined, using the Goldberg–Rao binary blocking flow algorithm [17], in time $O(|E| \min(|V|^{2/3}, \sqrt{|E|}) \log(|V|^2/|E|) \log U)$, where U is the network capacity. In our case we have $|V| = U = \Theta(m)$.

Corollary 2. *The transient stuck bits matching problem can be solved in time* $O(|E| \min(m^{2/3}, \sqrt{|E|}) \log^2 m)$.

We now need to determine the value of $|E|$.

Lemma 3. *$G_{T,P}$ has at most* $O(3^{\log m}) = O(m^{\log_2 3}) \approx O(m^{1.5489})$ *edges.*

Proof. There are always exactly m nodes from s to V_1, and between 1 and m nodes from V_3 to f. We consider the case where $|V_1| = |V_3|$, because that is the case with the most number of edges from V_1 to V_3. Every other case has a subset of nodes, thus a subset of edges from V_1 to V_3.

If $V_1 = V_3$ then for every node in V_1, if it has t zeros, it has 2^t outgoing edges, to all possible nodes in V_3 that a stuck bit can send it. Therefore, the total number of outgoing edges from V_1 is: $|E| = \sum_{t=0}^{\log m} \binom{\log m}{t} \cdot 2^t = 3^{\log m} \approx m^{1.5489}$ □

3.5. Total time and space of the transient stuck bits problem

From the above discussion we obtain the following:

Theorem 2. TransientStuckBits(T, P) *runs in* $\Theta(m^{\log_2 3+2/3} \log^2 m) = O(m^{2.2157} \log^2 m)$ *time, where m is the length of both text and pattern. The space complexity is* $\Theta(m^{1.5489})$.

Proof. The bound for the time complexity follows from the above discussion. It remains to show the space complexity. The pattern P and the text T are of space: $\Theta(m \log m)$, m symbols and for each symbol an address of length $\log m$. By Lemma 3 the number of edges in the bipartite graph is bounded by $O(m^{1.5489})$. The *Goldberg–Rao* binary blocking flow algorithm has linear space complexity. Thus, the total space is: $\Theta(m^{1.5489})$. □

4. Conclusions and open problems

This paper follows up recent work on a new paradigm for approximate pattern matching that, instead of content errors, considers location errors or rearrangement errors. Specifically, the problems of finding a match under stuck bits and transient stuck bits were studied and efficient solutions for these problems are provided. Most importantly, apart from the specific algorithmic results, this paper gives another evidence of the richness of the research field that is opened with the new paradigm.

We have solved the stuck bit problem only for the case where the text and pattern are of the same length. It can clearly be extended to a pattern matching setting where the text is of greater length and we would like to find the number of stuck bits for a every text location where there is a stuck bit matching. Our algorithm can, of course, be run for every text location separately. It would be interesting to know if a faster solution than $O(nm^{2.2156} \log^2 m)$ can be found.

This direction of research leads to more challenging questions. In reality, various types of address bit errors can occur. Some were considered in [5], some in this paper, and there are more. Different types of errors have different probabilities of occurrence. In some hardware configurations, even the "stuck" bits have a different probability of occurrence depending on where in the register they are located. It would be important to integrate various different errors into the Pattern Matching model and, in future work, consider the probabilities of the various errors as well.

Acknowledgements

The first author is supported by NSF grant CCR-09-04581, ISF grant 347/09, and BSF grant 2008217. The fifth author's work was supported by ISF, BSF and Google award.

References

[1] A. Amir, Asynchronous pattern matching, in: Proc. 17th Symposium on Combinatorial Pattern Matching, CPM, in: LNCS, vol. 4009, Springer, 2006, pp. 1–10. Invited Talk.
[2] A. Amir, Y. Aumann, G. Benson, A. Levy, O. Lipsky, E. Porat, S. Skiena, U. Vishne, Pattern matching with address errors: rearrangement distances, in: Proc. 17th ACM-SIAM Symp. on Discrete Algorithms, SODA, 2006.
[3] A. Amir, Y. Aumann, G. Benson, A. Levy, O. Lipsky, E. Porat, S. Skiena, U. Vishne, Pattern matching with address errors: rearrangement distances, Journal of Computer and System Sciences 75 (6) (2009) 359–370.
[4] A. Amir, Y. Aumann, P. Indyk, A. Levy, E. Porat, Efficient computations of ℓ_1 and ℓ_∞ rearrangement distances, Theoretical Computer Science 410 (43) (2009) 4382–4390.
[5] A. Amir, Y. Aumann, O. Kapah, A. Levy, E. Porat, Approximate string matching with address bit errors, in: Proc. 19th Symposium on Combinatorial Pattern Matching, CPM, in: LNCS, vol. 5029, Springer, 2008, pp. 118–129.
[6] A. Amir, Y. Aumann, O. Kapah, A. Levy, E. Porat, Approximate string matching with address bit errors, Theoretical Computer Science 410 (51) (2009) 5334–5346. Special Issue of CPM 2008 Best Papers.
[7] A. Amir, R. Cole, R. Hariharan, M. Lewenstein, E. Porat, Overlap matching, Information and Computation 181 (1) (2003) 57–74.
[8] A. Amir, T. Hartman, O. Kapah, A. Levy, E. Porat, On the cost of interchange rearrangement in strings, SIAM Journal on Computing 39 (4) (2009) 1444–1461.
[9] A. Amir, A. Levy, String rearrangement metrics: a survey, in: Algorithms and Applications, 2010, pp. 1–33.

[10] A. Amir, M. Lewenstein, E. Porat, Approximate swapped matching, Information Processing Letters 83 (1) (2002) 33–39.
[11] V. Bafna, P.A. Pevzner, Sorting by transpositions, SIAM Journal on Discrete Mathematics 11 (1998) 221–240.
[12] P. Berman, S. Hannenhalli, Fast sorting by reversal, in: D.S. Hirschberg, E.W. Myers (Eds.), Proc. 8th Annual Symposium on Combinatorial Pattern Matching, CPM, in: LNCS, vol. 1075, Springer, 1996, pp. 168–185.
[13] A. Carpara, Sorting by reversals is difficult, in: Proc. 1st Annual Intl. Conf. on Research in Computational Biology, RECOMB, ACM Press, 1997, pp. 75–83.
[14] Y.R. Choe, D.L. Schuff, J.M. Dyaberi, V.S. Pai, Improving vod server efficiency with bittorrent, in: Proceeding MULTIMEDIA '07 Proceedings of the 15th international conference on Multimedia, ACM, New York, NY, USA, 2007, pp. 117–126.
[15] D.A. Christie, Sorting by block-interchanges, Information Processing Letters 60 (1996) 165–169.
[16] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, McGraw-Hill, 1992.
[17] A. Goldberg, S. Rao, Beyond the flow decomposition barrier, Journal of the ACM 45 (5) (1998) 783–797.
[18] J.L. Hennessy, D.A. Patterson, Computer Architecture: A Quantitative Approach, 3rd ed., Morgan Kauffmann, 2002.
[19] J. Hopcroft, R. Karp, An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs, SIAM Journal on Computing 2 (4) (1973) 225–231.
[20] O. Kapah, G.M. Landau, A. Levy, N. Oz, Interchange rearrangement: the element-cost model, Theoretical Computer Science 410 (43) (2009) 4315–4326.
[21] R. Lowrance, R.A. Wagner, An extension of the string-to-string correction problem, Journal of the ACM (1975) 177–183.