Two Glass Balls and a Tower

Amihood Amir

1 Introduction

For many years, one of the favorite questions that interviewers for high-tech companies in Israel asked prospective job applicants was the following.

Suppose a company developed a very resilient glass. They created identical glass balls and desire to find the height from which a fall of the glass ball will cause it to shatter. The tests are conducted by dropping glass balls from various floors of an n-story building. We require the floor number k such that the glass breaks if dropped from it, but does not break from floor $k-1$.

We would like to know what is the minimum number of tests (ball drops) necessary to find the number k, in the worst case.

Three situations are considered:

1. **One ball:** The new applicant is presented with a single ball for the tests (due to the high cost of the new material).
   Clearly, the applicant has no choice but to start from floor 1 and drop the ball, if the ball breaks then the answer is 1. As long as the ball does not break, the tester needs to sequentially move up floor by floor, and drop the ball from each floor. When eventually the ball breaks upon the drop from floor k, that is the result. The number of tests is then $O(n)$.

2. **Unbounded number of balls:** The tester gets as many balls as her requires.
   Clearly, this is a case for the *divide-and-conquer* approach. Binary search over the n floors of the building can guarantee the result in time $O(\log n)$.

3. **Two balls:** The tester has exactly two balls.
   This is the intriguing case. Superficially, it seems like another ball can not improve the situation over the one-ball case by more than a constant. A second look at the problem shows that divide-and-conquer can still be used, but only to depth 2. This is a *bounded*

depth divide-and-conquer. Simply divide the n floors to groups of $\sqrt{n}$ sequential floors. This creates $\sqrt{n}$ such groups. Use the first ball to find the *group* where the ball breaks, in $O(\sqrt{n})$ tests. Then use the second ball to "fine tune" the result and find what floor within the group is the critical one. Since the group size is $\sqrt{n}$, the number of tests for this level is also $O(\sqrt{n})$, for a total of $O(\sqrt{n})$ tests.

The above problem demonstrates in a very clear way the bounded-depth divide-and-conquer technique. This technique has been playing a role in pattern matching for over two decades.

String matching, the problem of finding all occurrences of a given pattern in a given text, is a classical problem in computer science. The problem has pleasing theoretical features and a number of direct applications to "real world" problems. The Boyer-Moore [Boyer and Moore, 1977] algorithm is directly implemented in the *emacs* "`s`" and *UNIX* "`grep`" commands.

Advances in Multimedia, Digital Libraries and Computational Biology have shown that a much more generalized theoretical basis of string matching could be of tremendous benefit [Pentland, 1992; Olson, 1995]. To this end, string matching has had to adapt itself to increasingly broader definitions of "matching". Two types of problems need to be addressed – *generalized matching* and *approximate matching*. In generalized matching, one still seeks all exact occurrences of the pattern in the text, but the "matching" relation is defined differently. The output is all locations in the text where the pattern "matches" under the new definition of match. The different applications define the matching relation. Examples can be seen in Amir and Farach's *less-than matching* ([Amir and Farach, 1995]) or Amir, Iliopoulos, Kapah and Porat's *weighted matching* ([Amir *et al.*, 2005]). The second model is that of approximate matching. In approximate matching, one defines a distance metric between the objects (e.g. strings, matrices) and seeks all text location where the pattern matches the text by a pre-specified "small" distance.

One of the earliest and most natural metrics is the *Hamming distance*, where the distance between two strings is the number of mismatching characters. Let n be the text length and m the pattern length. Abrahamson [Abrahamson, 1987] showed that the Hamming distance problem, also known as the *string matching with mismatches* problem can be solved in time $O(n\sqrt{m \log m})$, i.e. within these time bounds one can find the hamming distance of the pattern at *every* text location. This is an asymptotic improvement over the $O(nm)$ bound even in the worst case.

All problems mentioned above, i.e. the *less-than matching, weighted matching,* and *Hamming distance*, were all solved by the bounded-depth divide-and-conquer method. We will review these problems and point out

the application of this method.

2 Hamming Distance

2.1 Problem Definition and Preliminaries

DEFINITION 1.

1. Let $a, b \in \Sigma$. Define

$$neq(a,b) =^{def} \begin{cases} 1 & \text{if } a \neq b; \\ 0 & \text{if } a = b. \end{cases}$$

2. Let $X = x_0x_1...x_{n-1}$ and $Y = y_0y_1...y_{n-1}$ be two strings over alphabet Σ. Then the *Hamming distance* between X and Y ($ham(X,Y)$) is defined as

$$ham(X,Y) =^{def} \sum_{i=0}^{n-1} neq(x_i, y_i).$$

3. The *The Hamming Distance* Problem is defined as follows:

   Input: Text $T = t_0...t_{n-1}$, pattern $P = p_0...p_{m-1}$, where $t_i, p_j \in \Sigma$, $i = 0, ...n-1$; $j = 0, ..., m-1$.

   Output: For every text location i, output $ham(P, T^{(i)})$, where $T^{(i)} = t_it_{i+1}...t_{i+m-1}$.

Abrahamson [Abrahamson, 1987] developed a seminal algorithm that finds $ham(P, T^{(i)})$ $\forall i$ in total time $O(n\sqrt{m \log m})$, using bounded-depth divide-and-conquer. We present a simplified version of that algorithm.

The two glass balls symbolize two different techniques to solve the problem. One easily handles the case of a small alphabet, and the other handles the case of every alphabet symbol appearing a small number of times. We will make a slight change in the problem requirements, though. We will count *matches* rather than *mismatches*. The Hamming distance can be easily calculated from the number of matches since it is simply the difference between the length of the pattern (m) and the number of matches.

2.2 Small alphabet

We consider the case where P has a *small* alphabet, e.g. less than $\sqrt{m}$ different alphabet symbols. We will use convolutions, as introduced by Fischer and Paterson [Fischer and Paterson, 1974]. Fischer and Paterson observed that string matching is a special case of a generalized convolution.

DEFINITION 2. Let $X = \langle x_0, ..., x_m \rangle$, $Y = \langle y_0, ..., y_n \rangle$ be two given vectors, $x_i, y_i \in D$. Let $\otimes$ and $\oplus$ be two given functions where

$$\otimes : D \times D \to E,$$

$$\oplus : E \times E \to E, \quad \oplus \text{ associative.}$$

Then the *convolution of X and Y with respect to $\otimes$ and $\oplus$* is:

$$X\langle \otimes, \oplus \rangle Y = \langle z_0, ..., z_{n+m} \rangle$$

where

$$z_k = \bigoplus_{i+j=k} x_i \otimes y_j \qquad \text{for} \quad k = 0, ..., m+n.$$

EXAMPLE 3.

Boolean Product: $\otimes$ is $\wedge$ and $\oplus$ is $\vee$.

Polynomial product: $\otimes$ is $\times$ and $\oplus$ is $+$.

Exact string matching: $\otimes$ is $=$ and $\oplus$ is $\wedge$ but the pattern is transposed.

For all matches of pattern $b\ a\ a$ in text $b\ a\ a\ b\ a$, do $b\ a\ a\ b\ a \langle =, \wedge \rangle b\ a\ a^R$.

$$\begin{array}{ccccccccc}
 & & & b & a & a & b & a \\
 & & & & & a & a & b \\
- & - & - & - & - & - & - & - \\
 & & & 1 & 0 & 0 & 1 & 0 \\
 & & 0 & 1 & 1 & 0 & 1 & \\
 & 0 & 1 & 1 & 0 & 1 & & \\
- & - & - & - & - & - & - & - \\
0 & 0 & & 1 & 0 & 0 & 1 & 0 \\
 & & & - & - & - & & \\
\end{array}$$

Note that $(X\langle =, \wedge \rangle Y)_k = 1$ iff $\langle x_{k-n}, ..., x_k \rangle = \langle y_n, ..., y_0 \rangle$ for $n \leq k \leq m$. We conclude that there is a match in position 2, i.e.

$$\begin{array}{cccccc}
 & b & a & a & b & a \\
 & - & - & - & & \\
\langle & a & a & b & \rangle^R & \\
\end{array}$$

The problem is that most such convolutions require time $O(nm)$ to compute. An exception is polynomial multiplication that can be achieved in time $O(n \log m)$ using the Fast Fourier Transform (FFT). Thus it is necessary to reduce the desired convolution to polynomial multiplication in the complex field in order to take advantage of the FFT algorithm. We show how this is done in the counting matches problem.

We need some definitions first.

DEFINITION 4. Define

$$\chi_\sigma(x) = \begin{cases} 1 & \text{if } x = \sigma \\ 0 & \text{if } x \neq \sigma \end{cases} \qquad \chi_{\overline{\sigma}}(x) = \begin{cases} 1 & \text{if } x \neq \sigma \\ 0 & \text{if } x = \sigma \end{cases}$$

If $X = x_0 \ldots x_{n-1}$ then $\chi_\sigma(X) = \chi_\sigma(x_0) \ldots \chi_\sigma(x_{n-1})$. Similarly define $\chi_{\overline{\sigma}}(X)$.
For string $S = s_0 ... s_{n-1}$, S^R is the reversal of the string, i.e. $s_{n-1} ... s_0$.

We return to the match problem for small alphabets. The product $\chi_\sigma(T)$ by $\chi_\sigma(P^R)$ is an array where the number in each location is the number of matches of a σ text element with a σ in the pattern. If we multiply $\chi_\sigma(T)$ by $\chi_\sigma(P)^R$, for every $\sigma \in \Sigma$, and add the results, we get the total number of matches. Since polynomial multiplication can be done in time $O(n \log m)$ using FFT, and we do $|\Sigma|$ multiplications, the total time for finding all matches using this scheme is $O(|\Sigma| n \log m)$.

Time: Our alphabet size is $O(\sqrt{m})$, so the problem can be solved in time $O(n\sqrt{m} \log m)$.

Notice that this technique knows how to count matches for "groups" of symbol occurrences, all those who are equal to the symbol being considered.

2.3 Every Symbols occurs only a Few Times in Pattern

Consider now the case where the alphabet is large, but also where every alphabet symbol appears in the pattern only a small number of times, e.g. no more than $\sqrt{m}$ times.

The most naive algorithm for counting matches is, for every text location i, to count all matches of the pattern and text, and record it in $M[i]$, where M is a *number of matches* array. This can be implemented by the following simple algorithm.

Naive Algorithm version A
{ Initialize M to 0 }
$M \leftarrow 0$
{ Main Loop }
For $i = 0$ to $n - 1$ do:
  For $j = 0$ to $m - 1$ do:
  If $t_{i+j} = p_j$ then $M[i] \leftarrow M[i] + 1$
  enfFor j
endFor i
end Algorithm

It is clear that version A is equivalent to version B below, where we simply

check all pattern element versus the text element scanned at the moment and update the appropriate counters.

Naive Algorithm version B
{ Initialize M to 0 }
$M \leftarrow 0$
{ Main Loop }
For $i = 0$ to $n-1$ do:
  For $j = 0$ to $m-1$ do:
  If $t_i = p_j$ then $M[i-j] \leftarrow M[i-j]+1$
  enfFor j
endFor i
end Algorithm

The time for both above algorithms is, naturally, $O(nm)$. However, note that in version B, we may make a slight improvement. We can pre-process the pattern and create, for every alphabet symbol σ, a list $L_\sigma[0], ..., L_\sigma[\ell_\sigma]$ of locations where σ occurs in the pattern, i.e. all i such that $p_i = \sigma$.

EXAMPLE 5. Let

	0	1	2	3	4	5	6	7	8	9
$P =$	A	A	B	A	C	B	A	B	C	B

Then A's list is: $L_A = 0, 1, 3, 6$, B's list is: $L_B = 2, 5, 7, 9$, and C's list is: $L_C = 4, 8$.

We can modify version B to only consider t_i's list in the comparison, rather than go through the entire pattern. This achieves the following algorithm:

Not-So-Naive Algorithm C
{ Initialize M to 0 }
$M \leftarrow 0$
{ Main Loop }
For $i = 0$ to $n-1$ do:
Let $\sigma \leftarrow t_i$
  For all $j = 0$ to ℓ_σ do:
  $M[i - L_\sigma[j]] \leftarrow M[i - L_\sigma[j]] + 1$
  enfFor j
endFor i
end Algorithm

In the worst case, Algorithm C has exactly the same time as version B. However, in the fortunate case we are considering, where every symbol occurs at most $\sqrt{m}$ times in the pattern, the lengths of the L lists never

exceeds $\sqrt{m}$ and algorithm C's running time is then $O(n\sqrt{m})$.

Notice that this second glass ball counts matches *within* groups of equal symbol occurrences. We sequentially go through the entire list of indices of every symbol.

2.4 General Alphabets

We are now ready to present the general algorithm.

DEFINITION 6. A symbol that appears in the pattern at least $\sqrt{m \log m}$ times is called *frequent*. Otherwise, it is called *rare*.

It is clear from the definition that there are at most $O(\sqrt{m}/\sqrt{\log m})$ frequent symbols, thus we can count matches of all frequent symbols in time $O(n\sqrt{m/\log m}\log m) = O(n\sqrt{m \log m})$ as shown in Subsection 2.2.

The matches of rare symbols are counted as in Algorithm C of Subsection 2.3. Since a rare symbol appears at most $\sqrt{m \log m}$ times, this stage is done in time $O(n\sqrt{m \log m})$.

3 Less-Than Matching

The Less-Than Matching problem was introduced by Amir and Farach [Amir and Farach, 1995] as a tool for solving the approximate matching problem in non-rectangular two dimensional images. The problem is defined as follows.

The *less-than matching problem* is:

Input: Text string $T = t_0, ..., t_{n-1}$ and pattern string $P = p_0, ..., p_{m-1}$ where $t_i, p_i \in \mathbb{N}$ (the set of natural numbers).

Output: All locations i in T where $t_{i+k-1} \geq p_k, \quad k = 1, ..., m$.

In words, every matched element of the pattern is not greater than the corresponding text element. If the text and pattern are drawn schematically, we are interested in all position where the pattern lies completely below the text. See Figure 1.

The less-than matching problem was also solved by the bounded-depth divide-and conquer technique [Amir and Farach, 1995]. As in the Hamming distance case, we also have two algorithms. The first handles small alphabets (groups of floors) and is solved by the FFT. The second handles the situation within the groups. This process is trickier than in the mismatch case that we saw in Section 2.3.

3.1 Small Alphabet

Assume that there are g different elements in the pattern.

NOTATION 7. For $\sigma, x \in \mathbb{N}$, let

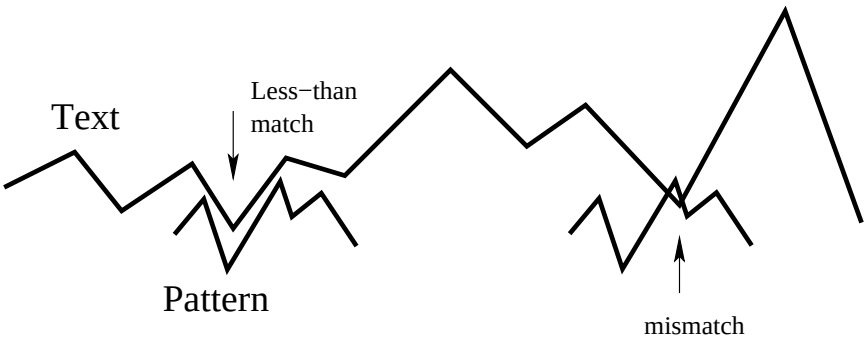

Figure 1. The first appearance of the pattern lies completely below the text, thus there is a match. The second is a mismatch.

$$\chi_\sigma(x) = \begin{cases} 1 & \text{if } x = \sigma \\ 0 & \text{if } x \neq \sigma \end{cases}$$

$$\chi_{<\sigma}(x) = \begin{cases} 1 & \text{if } x < \sigma \\ 0 & \text{if } x \geq \sigma \text{ or } x = \phi \end{cases}$$

If $X = x_1, \ldots, x_n$ then $\chi_\sigma(X) = \chi_\sigma(x_1), \ldots, \chi_\sigma(x_n)$. Similarly define $\chi_{<\sigma}(X)$.

We would like to know for each element of the pattern, where it is lined up with something less than it. We can achieve this by computing, for each σ in P, $\chi_{<\sigma}(T) \otimes \chi_\sigma(P^R)$ (where $\otimes$ is polynomial multiplication), and considering all non-zero locations.

Let $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_g\}$ be the set of all different numbers appearing in P. Let $M_i = \chi_{<\sigma_i}(T) \otimes \chi_{\sigma_i}(P^R)$ (where $\otimes$ is polynomial multiplication). Then M_i is non-zero at position t iff there is a σ_i in the pattern matched with something smaller than σ_i when the pattern is lined up at t. These cases are exactly when we get a mismatch. If we let M be the sum of all the M_i's we get a non-zero if there was a mismatch caused by any $\sigma \in \Sigma$. By using FFT we can calculate each of the polynomial multiplications in time $O(n \log m)$, for a total of $O(g\ n \log m)$.

3.2 The General Case

As we had mentioned, dealing with the elements within groups is not immediate. We need to define the groups in a somewhat different manner.

We therefore provide here the general framework and then show how to efficiently "fine tune" within groups.

Our input is text $T = t_0, \ldots, t_{n-1}$ and pattern $P = p_0, \ldots, p_{m-1}$. Without loss of generality we may assume that the text alphabet is the same as the pattern alphabet. If this is not the case, replace every text number by the largest pattern number that does not exceed it.

In this case also frequent numbers are handled differently from rare numbers. A number is *frequent* if it appears at least $\sqrt{m \log m}$ times, otherwise it is *rare*. There are at most $\sqrt{m}/\sqrt{\log m}$ frequent numbers.

We can use the FFT in the manner described in Subsection 3.1 to disqualify all text locations where there is a text element smaller than a frequent element. This can be done in time $O(n\sqrt{m \log m})$. Now in all remaining locations we just need to make sure that the text location is at least as large as the pattern element corresponding to it. This is done by splitting into groups.

Dividing into Groups:

Let $p_{j_0}, \ldots, p_{j_{g_1}}$ be the non-frequent elements of P.

Consider every pattern element as a pair $\langle s, d \rangle$ where s is a number and d is the location of the number in the array P. We get a list $L = \langle p_{j_0}, j_0 \rangle$, $\langle p_{j_1}, j_1 \rangle$, ..., $\langle p_{j_{g_1}}, j_{g_1} \rangle$.

Sort L lexicographically. (There are no more than m elements in L.) Call the sorted array L'.

Divide L' into $g = O(\sqrt{m/\log m})$ blocks, each containing no more than $2\sqrt{m \log m}$ elements, in a manner that no number appears in more than one block. We are assured that such a division is possible because all remaining numbers are non-frequent.

Possible Implementation: Put the first $\sqrt{m}$ occurrences in block 1. Let σ be the symbol of the last occurrence in block 1. Make sure all occurrences of σ are added to block 1 (since σ is rare it will not add more than $\sqrt{m \log m}$ elements. Continue in a similar fashion to create all blocks.

EXAMPLE 8. Let the pattern P be:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
G	A	H	A	C	B	D	E	C	B	F	F	E	G	A	B

$\sqrt{16} = 4$.

Sorting the occurrences gives: $\langle A, 1 \rangle, \langle A, 3 \rangle, \langle A, 14 \rangle, \langle B, 5 \rangle, \langle B, 9 \rangle, \langle B, 15 \rangle, \langle C, 4 \rangle, \langle C, 8 \rangle, \langle D, 6 \rangle, \langle E, 7 \rangle, \langle E, 12 \rangle, \langle F, 10 \rangle, \langle F, 11 \rangle, \langle G, 0 \rangle, \langle G, 13 \rangle, \langle H, 2 \rangle$.

The suggested implementation produces the blocks:
block 1: $\langle A, 1 \rangle, \langle A, 3 \rangle, \langle A, 14 \rangle, \langle B, 5 \rangle, \langle B, 9 \rangle, \langle B, 15 \rangle$.
block 2: $\langle C, 4 \rangle, \langle C, 8 \rangle, \langle D, 6 \rangle, \langle E, 7 \rangle, \langle E, 12 \rangle$.
block 3: $\langle F, 10 \rangle, \langle F, 11 \rangle, \langle G, 0 \rangle, \langle G, 13 \rangle$.
block 4: $\langle H, 2 \rangle$.

Notice that we have 4 blocks, none larger than 6 elements.

Construct the Text and Pattern of Representatives:
For each block B_i, $i = 0, \ldots, g \leq \sqrt{m/\log m}$ let b_i be the smallest (leftmost) element in the block; call b_i the *representative* of block B_i.

Let T' and P' be T and P such that every t_i and p_i is replaced by the representative of the block it is in. This step can be implemented by a sequential scan of L'.

We now find all less-than matches of P' in T', by the FFT. P' and T' can be considered "flattened out" versions of P an T. When we seek all less-than matches of P' in T' we only detect the "large" mismatches, i.e., those between elements that are so different that they are in different blocks. However, mismatches between elements of the same block are undetected. At this stage we must "fine tune" our approximate solution. We now come to the second level of our bounded divide-and-conquer. The one that needs to adjust for the mismatches that may exist but were not recorded. These are precisely the cases where a text number is smaller than a pattern number that is **in the same block** as the text number.

3.3 Checking Elements within Blocks

Scan T and for every element t_i of T only compare it to the $O(\sqrt{m \log m})$) elements of P that are in t_i's block. The subroutine is very similar to algorithm C in Section 2.3.

Fine Tuning Algorithm

For $i = 0$ to $n - 1$ do:
  Let B_{t_i} be the block of L' that t_i is in,
  Let $P^{B_{t_i}} \leftarrow \{\langle s, d\rangle | \langle s, d\rangle \in B_{t_i}\}$
  For every element $\langle s, d\rangle$ in $P^{B_{t_i}}$ { at most $2\sqrt{m \log m}$ elements }
    if $t_i < s$ then $M[i - d] \leftarrow M[i - d] + 1$
  endFor
endFor i

end Algorithm

The vector M is now correct since the first part of the algorithm included all the errors between blocks and the last part found all the errors within a block.

Time: $O(m \log m)$ for sorting and $O(n \log m)$ for reducing to alphabet of pattern.
$O(n\sqrt{m \log m})$ for less-than matching of the representatives
$O(n\sqrt{m \log m})$ for correcting mismatches within blocks

Total: $O(n\sqrt{m \log m})$

4 Approximate Matching in Weighted Sequences

Weighted sequences have been recently introduced by Iliopoulos et al. [Iliopoulos *et al.*, 2003] as a tool to handle a set of sequences that are not identical but have many local similarities. The weighted sequence is a "statistical image" of this set, where the probability of every symbol's occurrence at every text location is given.

DEFINITION 9. A *weighted sequence* $T = t_0, ..., t_n$ over alphabet Σ is a sequence of sets $t_i,\ i = 0, ..., n$. Every t_i is a set of pairs $(s_j, \pi_i(s_j))$, where $s_j \in \Sigma$ and $\pi_i(s_j)$ is the probability of having symbol s_j in location i. Formally,

$$t_i = \{(s_i, \pi_i(s_j)) \mid s_j \neq s_\ell \text{ for } j \neq \ell, \text{ and } \sum_j \pi_i(s_j) = 1\}.$$

For a finite alphabet $\Sigma = \{a_1, ..., a_{|\Sigma|}\}$ we can view a weighted sequence as a $|\Sigma| \times n$ matrix T of numbers in $[0, 1]$, where $T[j, i] = \pi_i(a_j)$. For the rest of this paper we assume a finite fixed alphabet Σ.

EXAMPLE 10. Let $\Sigma = \{A, B, C, D\}$, then an example of a text of length 7 is:

A	1/2	1/3	1/4	1	1/6	1/4	0
B	1/4	1/3	1/4	0	1/6	0	1/5
C	1/4	0	1/4	0	1/6	1/2	3/5
D	0	1/3	1/4	0	1/2	1/4	1/5

DEFINITION 11. $P = p_0, ..., p_m$ is a *solid sequence* over alphabet Σ if $p_i \in \Sigma,\ i = 0, ..., m$.

We say that *solid pattern* P (or simply *pattern* P) *occurs in location* i *of weighted text* T *with probability at least* α if $\prod_{j=0}^{m} \pi_j(p_j) \geq \alpha$.

EXAMPLE 12. Let T be the weighted text in the previous example, $P = ADCC$, and $\alpha = 0.1$. Then P occurs at location 3 with probability at least α, since the probability of P at location 3 is $\frac{3}{2 \cdot 2 \cdot 5} = \frac{3}{20} = 0.15 > 0.1$, but P does not appear in locations $0, 1$ and 2 since the probability at each of these locations is 0.

DEFINITION 13. The *exact weighted matching problem* is defined as follows:
Input: Weighted text T over alphabet Σ, solid pattern P over alphabet Σ, and probability $\alpha \in [0, 1]$.
Output: All locations i in T where pattern P occurs with probability at least α.

The following is a straightforward efficient algorithm for exact weighted matching.

Algorithm
1. Convert all values of T to their logarithms.
2. For each $\sigma \in \Sigma$ do:
   { Denote by T_σ the σ-th row of T, i.e. the list of probabilities of σ in all locations. }
   $S_\sigma \leftarrow T_\sigma \otimes \chi_\sigma(P)$.
   endFor
3. $Sum \leftarrow \sum_{\sigma \in \Sigma} T_\sigma$
4. For $i = 0$ to $n - m$ do: if $Sum[i] \geq \log \alpha$ then there is a match at location i

end Algorithm

Algorithm Time: Steps 1., 3. and 4. are trivially done in time $O(|\Sigma|n) = O(n)$ (since Σ is a fixed finite alphabet). Step 2. is done in time $O(|\Sigma|n \log m) = O(n \log m)$. Total time: $O(n \log m)$.

What interests us in the context of our bounded-depth divide-and-conquer method is the Hamming distance in weighted sequences problem.

Computing the Hamming distance between two (solid) strings assumes that a number of symbols were replaced. The Hamming distance is the number of these replaced symbols. In the case of weighted subsequences it makes a difference where these symbols were replaced. The simpler case, which we consider in this section, assumes replacement in the text. The assumption is that some text symbols are erroneous and, in fact, there should have been a probability 1 for the symbol that happens to match the pattern, rather than the probabilities that appear in the text.

EXAMPLE 14. For the text in example 10, consider pattern $P = BADC$ and $\alpha = 0.25$. There is no exact match. However, if we allow one mismatch, there is a match in location 2. Simply assume that the probability of having a B in location 2 is 1. At that point, the total probability at location 2 is $1 \cdot 1 \cdot 1 \cdot \frac{1}{2} \cdot \frac{1}{2} = \frac{1}{4}$. In location 0 even with one mismatch the probability is still $\frac{1}{48}$. In location 1 and one mismatch the probability is $\frac{1}{72}$, and in location 3 with one mismatch the probability is $\frac{1}{40}$.

Note that by this definition, allowing enough mismatches can guarantee a match at every location, no matter how close to 1 we choose α.

DEFINITION 15. The *Weighted Hamming Distance with Mismatches in the Text problem* is the following:
Input: Weighted text T over alphabet Σ, solid pattern P over alphabet Σ,

and probability $\alpha \in [0,1]$.
Output: For every location i in T, the minimum k such that if k text probabilities were changed to 1 then pattern P would occur at location i with probability at least α.

There does not seem to be a natural way to use the powerful constraint that the numbers in the weighted text are probabilities. However, it seems like we can solve the problem without it. We reduce the weighted Hamming distance with mismatches in the text problem to the *minimum ignored mask bits problem*. The idea is to consider a text whose elements are non-positive numbers, and a pattern which is a mask, i.e. its symbols are 0's and 1's. Suppose we are interested in finding out, for each text location i, the sum of the text numbers that are aligned with 1's in the pattern.

EXAMPLE 16. $T = -1, -3, -17, 0, -5, -6, -1$, $P = 1001$. Then the result at location 0 is -1, at location 1 is -8, at location 2 is -23 and at location 3 is -1.

Clearly this is a simple convolution of the pattern and text. However, we add a complication, we also have a non-positive integer α and for every text location i we seek the smallest number of mask bits that, if set to 0, would make the sum of text numbers that are aligned with (the remaining) 1's in the pattern, be no less than α.

EXAMPLE 17. In the example above, if $\alpha = -5$ then the result at location 0 is -1, with 0 dropped 1 bits from the mask, at location 1 it is -3 if the last mask bit is 0-ed, i.e. the mask becomes 1000, or -5 if the first mask is 0-ed, i.e. the mask becomes 0001. At location 2 we need to 0 two 1 bits in the mask to make the mask 0000 and the result 0. At location 3 the result is -1 with 0 mask bits altered.

We formally define the problem.

DEFINITION 18. The *Minimum Ignored Mask Bits problem* is the following:
Input: Solid text T of length $n+1$ whose elements are non-positive integers, solid pattern P of length $m+1$ over alphabet $\{0,1\}$, and integer $\alpha \leq 0$.
Output: For every location i in T, the minimum k such that if k pattern bits are changed from 1 to 0, and M' is the pattern resulting from those k changes, then $\sum_{j=0}^{m} T[i+j]M[j] \geq \alpha$.

CLAIM 19. The weighted Hamming distance with mismatches in the text problem is linearly reducible to the minimum ignored mask bits problem.

Proof: Given weighted text T in matrix format, where the value in $T[i,j]$

is $\log \pi_j(s_i)$, let solid text T' be a linear listing of matrix T in column-major order, i.e. $T' = T[1,0], T[2,0], T[3,0], ..., T[|\Sigma|,0]$,
$T[1,1], T[2,1], T[3,1], ..., T[|\Sigma|,1], ...,$
$T[1,n], T[2,n], T[3,n], ..., T[|\Sigma|,n]$. Let M be a string of length $|\Sigma|(m+1)$ over $\{0,1\}$ where M is the concatenation of strings $B(p_0), B(p_1), ..., B(p_m)$. $B(a)$ is defines as follows. Let $a = s_\ell$, where $\Sigma = \{s_1, s_2, ..., s_{|\Sigma|}\}$. Then $B(a)$ is a bit string of length Σ, where the ℓ-th element is 1 and all other elements are 0.

EXAMPLE: If $\Sigma = \{A, B, C, D\}$ and $P = BBAD$,
then $M = 0100\ 0100\ 1000\ 0001$.

Clearly, the reduction is linear. It is also clear that turning a 1 bit in the mask M to 0, is equivalent to changing the probability in the text position corresponding to it to 1. Thus a solution to the minimum ignored mask bits problem will provide the solution to the weighted Hamming distance with mismatches in the text problem. □

Algorithm's Idea:

We consider the two cases and show an easy efficient solution for each of them. Subsequently, we use the bounded-depth divide-and-conquer strategy, that splits a general input into the two straightforward cases, and thus solves each separately.

4.1 Bounded Alphabet

The first special case is one where the domain of numbers appearing in the text is bounded, i.e. there are only r different numbers that can appear as text elements. Formally, let $R = \{n_1, ..., n_r\} \subset \mathbb{Z}^- \cup \{0\}$, and let T be a text over R. Assume that $n_1, ..., n_r$ are sorted in descending order. Once we know that the only possibilities for text values are from R, we can calculate, for every location i the r sums $S_{i,j}$, $j = 1, ..., r$, where $S_{i,j}$ is the sum of n_j's that are matched to 1's in the mask at location i.

Since we are interested in finding the smallest number k of mask 1 bits that, when turned to 0 will make the sum greater than α, and since **all numbers are non-positive**, the following observation is crucial to the algorithm:

OBSERVATION 20. For any location i where $\sum_{j=1}^{r} S_{i,j} < \alpha$, the solution to the minimum ignored mask bits problem can be found by sequentially adding numbers that participate in the sum starting from the ones that contribute least to decreasing it, i.e. the largest (n_1). Stop adding them when the remaining sum is no longer less than α.

This elimination would normally require $O(m)$ work per location. However, since there are only r different values, and we know how many instances of each value participate in the sum at location i ($S_{i,j}/n_j$), we can do this

in time $O(r)$ per location.

This gives rise to the following algorithm:

Algorithm Bounded Alphabet
1. For $j = 1$ to r do:
    { Denote by S_j the array whose elements are $S_{0,j}, S_{1,j}, ..., S_{n,j}$,
    i.e. $S_j[i] = S_{i,j}$.}
    $S_j \leftarrow \chi_{n_j}(T) \otimes M$
  endFor
2. For $i = 0$ to $n - m$ do:
    $S, j \leftarrow 0$
    While $S \geq \alpha$ do:
      $j \leftarrow j + 1$
      $S \leftarrow S + S_j[i]$
    endWhile
    { The situation is $S < \alpha$ but $S - S_j[i] \geq \alpha$.}

$$k_i \leftarrow \left(\sum_{\ell=1}^{j-1} \frac{S_\ell[i]}{n_\ell} \right) + \left\lfloor \frac{(S - \alpha)}{n_j} \right\rfloor$$

  endFor
end Algorithm

Algorithm's Time: $O(rn \log m)$ for step 1. and $O(rn)$ for step 2. for a total of $O(rn \log m)$. Note that step 2. could be done faster by binary search, but since it is dominated by the time of step 1., we wrote the simpler pseudocode.

4.2 Bounded Number of Large Numbers

A second special case we consider is when there is no bound on the number of different text elements, but we do know that for every text substring of length m there are at most r elements greater than α. This means that for location i, there is no point in even considering all elements except those r.

Here the algorithm is much simpler. Assume the r elements are sorted in non-increasing order. For every text location i, it suffices to consider the r elements from largest to smallest. For each element check whether it correspond to a 1 bit in the mask. If so, add it to the sum and check if it is still above α. When all elements are considered, or when the sum drops below α, if ℓ elements were added to the sum and if the mask has s 1 bits, then k is $s - \ell$.

Formally, let $B_i = \{\langle b_{i,1}, \ell_{i,1}\rangle, \langle b_{i,2}, \ell_{i,2}\rangle, ..., \langle b_{i,r}, \ell_{i,r}\rangle\}$ be the set of pairs

such that $b_{i,1}, b_{i,2}, ..., b_{i,r}$ are the values of $\{T[i], T[i+1], ..., T[i+m]\}$ that are greater than α, sorted in non-increasing order, and $\ell_{i,j}$ is the index of $b_{i,j}$ in T, $j = 1, ..., r$, i.e. $T[\ell_{i,j}] = b_{i,j}$.

The algorithm is as follows:

Algorithm Bounded Relevant Numbers
For $i = 0$ to $n - m$ do:
$S \leftarrow 0$ { S is the masked sum at location i. }
$j \leftarrow 0$ { j is a counter of the relevant text elements. }
$x \leftarrow 0$ { x is a counter of the number of elements in the masked sum. }
While $S \geq \alpha$ and $j < r$ do:
$j \leftarrow j + 1$
If $M[\ell_{i,j}] = 1$ then $S \leftarrow S + b_{i,j}$
$x \leftarrow x + 1$
endWhile
$k_i \leftarrow s - x + 1$
endFor
end Algorithm

Algorithm's Time: $O(nr)$.

4.3 Divide and Conquer

We are now ready to present our divide-and-conquer algorithm. Assume first, that the text length is at most $2m$. This is a standard assumption and can be made without loss of generality because of the following lemma.

LEMMA 21. *Assume that there exists an algorithm that solves the less-than matching problem in time $O(mf(m))$ for $n \leq 2m$, then there exists an algorithm that solves the less-than matching problem in time $O(nf(m))$ for any n-length text.*

Proof: Simply divide the text into $2\frac{n}{2m}$ overlapping $2m$-length segments (see Fig. 2) and solve the matching problem separately for each. Clearly, if there is a match in any location of T it will appear in one of the segments. □

We now have a situation where the text is of size $2m$, the pattern of size m. Sort all text elements and split them into r blocks of size at most $2|\Sigma|\frac{m}{r}$ each.

The idea is to use *Algorithm Bounded Alphabet* on the blocks, and *Algorithm Bounded Relevant Numbers* to find the border of the numbers participating in the sum within the block that tips under α.

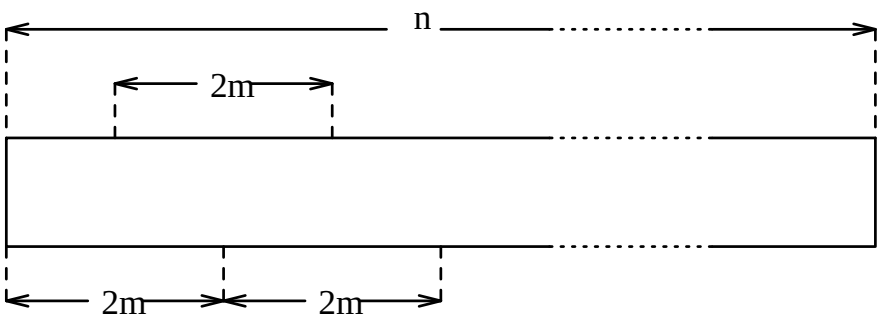

Figure 2. Slicing problem to smaller problems.

Since the greatest contribution to the masked sum lies with the larger numbers, it is clear that for the mismatches, our strategy should be replacing the smaller numbers where necessary. Therefore, we go block by block, from the largest down, and compute the following two sets of values for every location:

1. How many numbers from each block participate in the masked sum for that location.
2. What is the sum of each block for that location.

Given the above two values, we can compute in time $O(r)$ for every location, within the granularity of blocks, what is the Hamming distance. Note that there will be one block at the "seam" where some values could possibly participate in the product and yet still not plunge below α. We need to adjust for these values. This will be done by *Algorithm Bounded Relevant Numbers* and take time $O(\frac{m}{r})$ per text location.

Algorithm's Time: The time for this algorithm is $O(rf(m)) + O(m\frac{m}{r})$, where $f(m)$ is the time it takes to compute the block information. We do it by convolutions, as in *Algorithm Bounded Alphabet* so $f(m) = m \log m$. The optimal r is then the one where

$$rm \log m = m\frac{m}{r}$$

$$r^2 = \frac{m}{\log m}$$

$$r = \sqrt{\frac{m}{\log m}}.$$

Thus the algorithm's time is $O(n\sqrt{m \log m})$.

5 Conclusion

We considered three problems, the Hamming distance problem, the less-than matching problem and the Hamming distance problem in weighted sequences. Muthukrishnan [Muthukrishnan and Palem, 1994] showed that those problems can not be solved by convolutions alone in time faster than $O(nm)$. Nevertheless the bounded-depth divide-and-conquer technique allows them all to be solved in time $O(n\sqrt{m \log m})$.

6 Acknowledgements

Professor Dov Gabbay was my instructor in undergraduate logic and then in a number of advanced logic courses and seminars. He was also my Ph.D. advisor and my thesis was on functional completeness in temporal logics. Since then I have taken some other turns, to complexity, computational biology, and algorithms. This paper is a purely algorithmic paper, but Professor Gabbay's influence on it is apparent to all who know him.

One of the traits that made taking courses with Professor Gabbay fun was the anecdotal examples accompanying the mathematics. This certainly helped garner a better intuitive understanding of the material and this is what I try to employ in my classes, and in this "glass balls and tower" exposition. I would like to point out several examples from Professor Gabbay's undergraduate logic class that illustrate this concept.

When explaining to the class the difference between *inclusive or* and *exclusive or* Professor Gabbay gave the following two examples:

"Suppose there was a sign on a movie theater saying that the entrance costs 2 Liras **or** show a soldier's ID. If a simple soldier went and bought a ticket, he would certainly not be denied entrance.

On the other hand, if you walk in the street and see a mother followed by a wailing child who then threatens the child: "stop crying **or** I will smack you". If the terrified child shuts up we will look unkindly at the mother if she will hit the child and triumphantly declare: "**inclusive or!!!**. Therefore," concluded Gabbay, "if you ever find yourself in a bank when a character in a sky mask and brandishing an *Uzi* barges in and shouts: "your money or your life!", be sure that you ascertain whether he means *inclusive* or *exclusive or*."

Another monumental statement of Professor Gabbay was the following. When he defined *tautology*, Professor Gabbay explained that it is a statement that is always true. "Therefore," he said, "if you are a politician, you should attempt to always say tautologies."

The next day (and you will soon find out how dated this story is) the following headline appeared in the newspaper:

Kissinger: There may or may not be important developments.

I prepared a sign with both above statements and it hung on my walls for many years:

> Gabbay: If you are a politician, you should attempt to always say tautologies.
>
> Kissinger: There may or may not be important developments.

For teaching me to do research, and for showing me that it should be fun - Thanks.

More Acknowledgements

Partially supported by NSF grant CCR-01-04494 and ISF grant 35/05.

BIBLIOGRAPHY

[Abrahamson, 1987] K. Abrahamson. Generalized string matching. *SIAM J. Comp.*, 16(6):1039–1051, 1987.

[Amir and Farach, 1995] A. Amir and M. Farach. Efficient 2-dimensional approximate matching of half-rectangular figures. *Information and Computation*, 118(1):1–11, April 1995.

[Amir *et al.*, 2005] A. Amir, C. Iliopoulos, O. Kapah, and E. Porat. Approximate matching in weighted sequences. submitted for publication, 2005.

[Boyer and Moore, 1977] R.S. Boyer and J.S. Moore. A fast string searching algorithm. *Comm. ACM*, 20:762–772, 1977.

[Fischer and Paterson, 1974] M.J. Fischer and M.S. Paterson. String matching and other products. *Complexity of Computation, R.M. Karp (editor), SIAM-AMS Proceedings*, 7:113–125, 1974.

[Iliopoulos *et al.*, 2003] C. S. Iliopoulos, L. Mouchard, K. Perdikuri, and A. Tsakalidis. Computing the repetitions in a weighted sequence. In *Proceeding of the Prague Stringology Conference*, pages 91–98, 2003.

[Muthukrishnan and Palem, 1994] S. Muthukrishnan and K. Palem. Non-standard stringology: Algorithms and complexity. In *Proc. 26th Annual Symposium on the Theory of Computing*, pages 770–779, 1994.

[Olson, 1995] M. V. Olson. A time to sequence. *Science*, 270:394–396, 1995.

[Pentland, 1992] A. Pentland. Invited talk. NSF Institutional Infrastructure Workshop, 1992.