Pattern Matching Algorithms: An Overview

Shoshana Neuburger

Department of Computer Science

The Graduate Center, CUNY

September 15, 2009

Abstract

Pattern Matching addresses the problem of finding all occurrences of a pattern string in a text string. Pattern matching algorithms have many practical applications. Computational molecular biology and the world wide web provide settings in which efficient pattern matching algorithms are essential. New problems are constantly being defined. Original data structures are introduced and existing data structures are enhanced to provide more efficient solutions to pattern matching problems. In this survey, we review pattern matching algorithms in one and two dimensions. We focus on several specific problems, among them small space pattern matching, parameterized matching, and dictionary matching. We also discuss the indexing problem, for which the classical data structure is the suffix tree, or alternatively, the suffix array.

Contents

1 Exact Matching

The *Pattern Matching Problem* is defined as follows:

Let Σ be an alphabet.

INPUT: *Text* string $T = t_1 t_2 \ldots t_n$ and *pattern* string $P = p_1 p_2 \ldots p_m$, t_i, $p_i \in \Sigma$.

OUTPUT: All locations i in T where there is an occurrence of the pattern, i.e., $T[i+k] = P[k+1]$, $0 \leq k < m$.

A naive solution to exact matching views each text position as a possible pattern start and compares the pattern at each text position in $O(nm)$ time. More sophisticated methods achieve linear complexity, i.e. O(m+n). Knuth, Morris, and Pratt presented a linear-time automata method to find a pattern in text. Boyer and Moore devised an alternate linear-time solution which jumps over parts of text and achieves average case performance time $O(\frac{n}{m})$.

The naive algorithm for pattern matching and its improvements such as the Knuth-Morris-Pratt algorithm are based on a *positive* view of the problem; the algorithm tries to prove at each text location that the position is an occurrence of P. In applications, T often contains relatively few occurrences of P. Hence the *negative* view can fit the situation better than the positive one. An algorithm that adopts the negative point of view quickly eliminates candidate positions by proving that there can*not* be an occurrence of P at each text location. This forms the *elimination* phase of the algorithm. A separate *checking phase* then examines whether or not a potential occurrence really contains P. This leads to fast algorithms because the elimination phase needs to examine only a small fraction of T on average. The Boyer-Moore algorithm has this flavor.

1.1 Linear-Time Pattern Matching

Knuth, Morris, and Pratt developed a linear-time method to find a pattern in text [52]. We will hereafter refer to their method as KMP. The KMP automaton maintains a failure link for each prefix of the pattern. The failure link of a position points to its longest suffix that is also a prefix. In other words, the failure link encapsulates the longest border of each prefix of a pattern.

The Boyer-Moore algorithm provides another linear-time solution to pattern matching [13]. Its execution time can be sublinear. It does not always examine every character of the text, but rather skips some of them. Generally, the algorithm achieves better speed for longer patterns. The combination of right to left verification with left to right shifts of the pattern forms a faster algorithm. The average case performance of the algorithm is $O(\frac{n}{m})$ while the worst case performance remains linear. The extra space required to preprocess the pattern is alphabet dependent, i.e., $O(m + |\Sigma|)$.

1.2 Randomization

The Rabin and Karp fingerprinting algorithm for string matching is a randomized algorithm whose performance is linear on average [49]. Their algorithm saves comparisons by computing the hash value of several characters, thereby comparing a group of characters with a single comparison. The preprocessing consists of hashing the pattern. The algorithm computes the hash value for each substring of the text that is the same length as the pattern. Each substring's hash value is compared to the pattern's hash value. When the hash values match, the pattern is compared to the text, character by character. If the hash values do not agree, the text is surely different from the pattern. Thus, character comparisons are often skipped. The trick is to use a hash function that can be incrementally computed as the text is scanned. The hash value should be updatable in $O(1)$ time when we eliminate the leftmost character of the substring and include the next text character to the right of the substring.

In the best case, when none of the hash values match that of the pattern, the Rabin-Karp algorithm achieves sublinear time complexity. Yet, the Knuth-Morris-Pratt and Boyer-Moore algorithms often have better performance than the Rabin-Karp algorithm when searching for a single pattern. However, Rabin-Karp is an algorithm of choice for multiple pattern search. Multiple pattern matching, also known as dictionary matching, is discussed in Section 2.

1.3 Witnesses and Duels

Vishkin introduced the dueling paradigm for pattern matching [68]. For pattern P, let π represent min(length of the period, m/2). Periodicity is defined in Section 6.1. For every i, $0 < i < \pi$, there exists some position k, $k > i$ in P such that $P_l \neq P_{k-i}$. We call k a *witness* against i (being

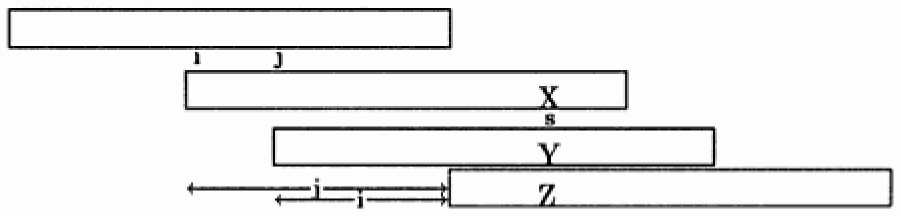

Figure 1: Duel: $X \neq Y$; therefore we cannot have $Z = X$ and $Z = Y$

a period of P). Vishkin suggested the construction of a witness array in the preprocessing of the pattern. The size of the witness array is π. Then, for every pair of close candidates in the text, a *duel* is performed to eliminate one or both of the candidates. Suppose i and j are candidates and $0 < j - i < \pi$, with $h = witness[j - i]$. Since $P_h \neq P_{h+i-j}$, at most one of them is equal to T_{i+h}. After all duels are performed, the remaining candidates are *consistent*. It is sufficient to compare each text position to the corresponding position in any candidate among a consistent set. This alphabet independent method is used for fast pattern matching in a parallel setting.

1.4 Deterministic Sampling

In 1991, Vishkin introduced deterministic sampling, which mimics randomization deterministically [69]. The method carefully selects a pattern sample whose size is at most logarithmic in the pattern size. Then, the sample is found in the text. For a non-periodic pattern, all but one occurrence of the pattern within each “neighborhood” of the text is disqualified, which provides sparse verification.

A periodic pattern can be reduced to a non-periodic pattern as follows. Suppose π is the length of the period. The pattern prefix p of size $2\pi - 1$ is a non-periodic string. Search for p in the text and link neighboring occurrences of p if the linking character is present. The verification of candidates consists of determining if a chain is long enough to form a pattern occurrence.

The deterministic sample is a set A of positions of the pattern and a number x, such that if the positions match a candidate i in the text, then all other candidates in the interval $[i-x+1, i+\frac{m}{2}-x]$ are eliminated. The deterministic sample is crucial for very fast string matching algorithms since a very small deterministic sample can be found.

How is the sample constructed? Consider P shifted and stacked $\frac{m}{2}$ times. Since the pattern is

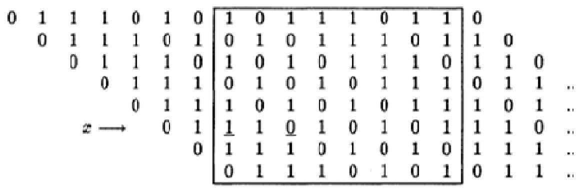
Figure 2: A 2-size deterministic sample for 8 shifts: A = 2, 4 and x=6

not periodic, the copies are all different and every column contains at least two different symbols. Choose any witness between the first and last copies; let it be column j. Select the less frequent of the two symbols, which can occur in at most half of the copies. Eliminate the columns that have a different symbol in the witness column. At most, half of the copies remain. This procedure is repeated at most $log\ \frac{m}{2}$ times until only one copy remains at row x, $1 \leq x \leq \frac{m}{2}$. The procedure to find a small deterministic sample can be parallelized. Figure 2 shows a deterministic sample for a pattern.

Galil compares efficient pattern matching to the task of hunting lions in the desert [31]. He uses a process called a diet to limit the size of the lions being sought in the desert. The number of positions to search is also limited. The algorithm is work optimal, which means it finds a pattern of size m in a text of size n in constant time with n processors. This method is based on Vishkin's deterministic sampling.

2 Dictionary Matching

The *Dictionary Matching Problem* is defined as follows:

Preprocess a set of of pattern strings $D = \{P_1, P_2, \ldots, P_k\}$ called a *dictionary.*

Subsequently, for every INPUT: *text* string $T = t_1 t_2 \ldots t_n$.

OUTPUT: All locations in the text where there is a match with any pattern in the dictionary.

A naive solution to the dictionary matching problem would search for each pattern in the text independently. Then, the text is scanned k times. More clever solutions examine the text only once, achieving linear time complexity.

2.1 Aho Corasick Automaton

Aho and Corasick [1] extended the KMP automaton to pose an efficient solution to the dictionary matching problem. We will hereafter refer to their method as AC. The AC automaton finds any element of a set of k patterns, each of size m, in a text of size n in $O(km)$ preprocessing and $O(n\ log\ |\Sigma| + occ)$ search time, where there are *occ* occurrences of patterns in the text. If the patterns are of different sizes, the preprocessing time is proportional to the sum of the lengths of the patterns. The search time depends on the size of the alphabet, but is not related to the number of patterns in the dictionary. The transition function of the KMP automaton is determined solely by the current state while the AC automaton's transition is a function of the current state and the character encountered. The number of branches emanating from a node is limited by the size of the alphabet.

The AC automaton can be built in linear time for integer alphabets as described by Dor and Landau [25]. Construction of the automaton is strictly proportional to the dictionary size and does not depend on the alphabet size. Search time still includes the *log* factor. The new algorithm is based on an observation regarding the failure function of KMP and AC. That is, the failure function is closely connected to the generalized suffix tree of the patterns. The automaton is constructed with the use of suffix arrays to sort the patterns, as well as suffix trees to gather information about the relationship between the patterns.

Breslauer has an approach to dictionary matching that does not only use equal/unequal comparisons [14]. This way the *log* factor in the AC algorithm is eliminated and the algorithm is truly linear in the size of the text. The algorithm compares the symbol in the current text position to the median of the alphabet symbols that appear in all potential occurrences in the column which is aligned with this text position.

In dynamic dictionary matching, the dictionary can change. An efficient algorithm must accommodate efficient insert and update operations on the dictionary. The first parallel algorithm achieving optimal amortized work for dictionary matching and updating the dictionary, in the case of an unbounded alphabet, was presented by Muthukrishnan and Palem [60]. Ferragina and Luzzio offer an algorithm with superior complexity bounds [27]. Their results also apply to dynamic parameterized dictionary matching.

2.2 Faster Dictionary Matching

Expected sublinear search time is achieved by the Boyer-Moore algorithm in the search for a single pattern. Similarly, algorithms have been developed to solve multiple pattern matching in time smaller than the size of the text.

The Commentz-Walter algorithm [18] combines the ideas of the Aho-Corasick and Boyer-Moore algorithms to form an algorithm that is generally faster than the AC automaton. A trie is built for the reversed patterns to mimic the right to left scanning of a pattern. Shifts are based on the value of the matching characters and its appearance in the patterns. The preprocessing, which consists of setting up the trie and the shift function, is still linear in the size of the dictionary.

Wu and Manber use the Boyer-Moore technique in dictionary matching to skip segments of text that cannot contain any pattern occurrence. They work with blocks of the pattern and the text rather than individual characters. The shift table is computed for hash values of pattern blocks. In the text scanning phase, the hash value of each text block is looked up in the table. [71]

Kim and Kim describe a bit-theoretic approach that offers an improvement over the Wu-Manber algorithm [51]. Their algorithm is based on a compact encoding and hashing scheme for the patterns.

3 2D Matching

The *Two Dimensional Pattern Matching Problem* is defined as follows:
Let Σ be an alphabet.
INPUT: *text* array $T[1 \ldots n, 1 \ldots n]$ and *pattern* array $P[1 \ldots m, 1 \ldots m]$.
OUTPUT: All locations (i, j) in T where there is an occurrence of the pattern, i.e., $T[i+k, j+l] = P[k+1, l+1]$, $0 \leq k, l < m$.

The main application area is pattern matching in images, such as optical character recognition and biomedical imaging.

3.1 Bird / Baker Algorithm

The first linear-time algorithm for two dimensional pattern matching with bounded alphabets was obtained independently by Bird and Baker, in [12] and [11], who convert the 2D pattern matching problem into a 1D string matching problem. Each row of the pattern is viewed as a metacharacter. An AC automaton is built for the rows of the pattern in time and space proportional to the pattern size. Like any AC automaton, the complexity is alphabet dependent. The Bird / Baker algorithm solves 2D pattern matching in $O(n^2 log|\Sigma|)$ time and space. Not only is this algorithm online, it is real-time since each part of the text is only examined once, as it is encountered.

Bird / Baker Algorithm:

```
Phase I: Preprocess pattern
   Give each row of P a unique name using AC automaton of pattern rows.
   Represent P as a 1D vector and construct the 1D KMP automaton.
Phase II: Row matching
   Label positions of T where suffix matches row of P using AC automaton.
Phase III: Column matching
   Use KMP on named columns of T to find pattern occurrences.
```

(Note: Phases II and III can be done simultaneously.)

The Bird / Baker algorithm easily extends to dictionary matching. If the KMP automaton is replaced with an AC automaton, the above algorithm solves the 2D dictionary matching problem.

3.2 2D Witness Table

Amir, Benson, and Farach [4] use the concepts of dueling and 2D periodicity to perform alphabet-independent linear-time pattern matching in two dimensions. Like its one dimensional counterpart, the algorithm preprocesses the pattern to form a witness table, then duels are performed when the text is scanned. The pattern is preprocessed to form two witness tables, one for each direction of a duel. The witness table is linear in the size of the pattern. A witness is a position of conflict between two overlapping copies of the pattern. The text scanning consists of two phases, consistency and verification. At the culmination of the text consistency phase, all remaining candidates for pattern occurrence are pairwise consistent. That is, they agree on the expected text characters. The verification phase determines which of the consistent candidates are actual occurrences of the pattern, by comparing text characters and pattern characters. The order in which the positions are verified is significant. Galil and Park [32] developed alphabet-independent 2D witness computation, which can be used to improve the preprocessing of the pattern matching algorithm. Cole et. al. [17] showed in 2004 that computation of the 2D witness table can even be parallelized.

3.3 2D Deterministic Sampling

Crochemore et. al. introduce deterministic sampling in two dimensions [21]. The authors assume that the pattern has a "good" sample. That is, the pattern segment of size $10 * log\, m$ in the $\frac{m}{2} + 1$ row beginning in the $\frac{m}{2} + 1$ column does not occur anywhere else in the pattern. Then, there is a square field of fire of size $m/2$ around an occurrence of the special segment. That is, in a text block of size $3m/2 \times 3m/2$, there can only be one pattern candidate. The candidate is found by looking for the 1D pattern sample. Then the pattern occurrence at a candidate position is verified naively. Since there are so few candidates, this algorithm runs in time proportional to the text size. Furthermore, most 2D patterns have good samples. This follows from simple calculations based on probability.

Kärkkäinen and Ukkonen describe several methods for 2D deterministic sampling [46]. They propose a generic algorithm for 2D sampling. It consists of first selecting a sample size, a template shape, and a sampling scheme for the given pattern and text. The scheme must provide exactly one witness for each occurrence of P in T. The preprocessing consists of constructing a suitable data structure to quickly find all pattern samples in a given text string. The text scanning consists of two phases, elimination and checking. During elimination, text positions that are not compatible with the pattern template are eliminated by using the data structure constructed in the preprocessing stage. Candidates that survive elimination can be verified naively in the checking stage. A change in sampling scheme results in a different concrete algorithm. A sampling scheme is designed to minimize the number of text positions inspected during elimination, as the text is typically much larger than the pattern. Linear samples and square samples are considered at length. Samples are small so the extra space required by the algorithm is minimal.

Crochemore et. al. extend deterministic sampling to two dimensions [19]. It is the basis for an alphabet-independent parallel algorithm for 2D matching. Unlike most fast parallel algorithms for string matching, their algorithm does not rely on 2D periodicity. Only *h-periodicity* (or *horizontal periodicity*) is used. Like other algorithms for alphabet-independent 2D matching, the preprocessing consists of computing witnesses for the pattern.

To classify the 2D periodicity of a matrix, two overlapping copies of the matrix are examined. See Section 6.2 for more details. A period vector, or a symmetry vector, that is horizontal is called a *horizontal period vector* or *hpv* of P. If P has a valid *hpv*, it is called *h-periodic*. The length of the smallest *hpv* of P is the least common multiple (LCM) of the periods of the rows of P considered as strings. As in Vishkin's 1D deterministic sampling technique, the algorithm of [19] assumes that the pattern is h-aperiodic. If P is h-periodic, then an h-aperiodic portion of P is preprocessed. The preprocessing of the pattern works with blocks of width α to find a logarithmic-sized pattern sample.

4 Pattern Matching in Small Space

Applications arise in which additional storage space is limited. Thus, effort has been invested in developing pattern matching algorithms that are linear in time but require only $O(1)$ extra space. The first time-space optimal pattern matching algorithm is from Galil and Seiferas [33]. Crochemore and Perrin developed "Two-Way String Matching" which blends the classical Knuth-Morris-Pratt and Boyer-Moore algorithms but computes pattern shifts as needed [23].

Rytter presented a constant-space, yet linear-time version of KMP in 2003 [63]. The algorithm relies on small space computation of both approximate periods and lexicographically maximal suffixes, which leads to $O(1)$ space computation of periods. Space-efficient real-time searching is discussed by Gasieniec and Kolpakov [34]. Their innovative algorithm uses a partial *next* function to save space.

An $O(1)$ space string matching algorithm that uses deterministic sampling with KMP is developed by Gasieniec et. al. [35]. The sampling stage limits the overall number of comparisons to $2n$.

A series of papers was published in the search for a small space 2D pattern matching algorithm.

The quest culminated when the achievements of Crochemore et. al. were published [20]. Theirs is the first alphabet independent algorithm for 2D pattern matching in linear time and small space. Pattern preprocessing requires $O(log\, m)$ extra space and the searching phase works in $O(1)$ space. The main concepts are 2D deterministic sampling and the innovative technique of *zooming sequences*. A zooming sequence is a sequence of nonperiodic regular objects of decreasing size, which can be computed recursively. The algorithm exploits properties of 2D periodicity.

There is a dearth of deterministic algorithms that perform dictionary matching in linear time and small space even in one dimension. Chan et. al. [15] derived a small space dictionary searching mechanism. The search phase of their algorithm introduces an extra log factor; it is not of linear time. Fredriksson presented an attempt at small space dictionary matching in 2008 [29]. Experimental results demonstrate that the algorithm achieves optimal average time. The algorithm is based on a combination of compressed self-indexes [28] and backward DAWG matching [62].

5 Parameterized Matching

Baker formalized the problem of parameterized matching [9]. Her motivation was the software maintenance problem, which seeks to identify duplication in a large software system. A parameterized match between two sections of code means that one section can be transformed into the other by replacing the identifiers of one section with the identifiers of the other via a one-to-one function. She developed a framework to efficiently solve the problem in 1D that includes parameterized suffix trees.

Parameterized strings or *p-strings* are strings that contain both ordinary symbols from an alphabet Σ and parameter symbols from an alphabet Π. Two p-strings are a parameterized match, or *p-match* if one p-string can be transformed into the other by applying a one-to-one function that renames the parameter symbols. For example, if $\Pi = \{x, y, v\}$ and $\Sigma = \{A, B, C\}$, then $S = AxAyBxyCBy$ p-matches $T = AyAvByvCBv$ when x and y of S are renamed as y and v, respectively, in T. The parameterized matching problem is that of finding all locations of the text that p-match the pattern.

Determining if two strings p-match each other can be done with a straightforward comparison between the two strings while maintaining a table to represent the characters that have already been mapped. Parameterized matching is not as simple. Baker's parameterized suffix tree is based on forming a *predecessor string*, also called *next(s)*, which can be done in linear time. The predecessor string of a string S has the distance between i and the location containing the previous appearance of the symbol at each location i. The first appearance of a symbol is replaced with a 0. For example, the predecessor string of $xxyyxyx$ is 0101322. Two strings that p-match have the same predecessor string. The difficulty in constructing the parameterized suffix tree lies in the possible absence of suffix links since the *distinct right context* property is not valid. With a parameterized suffix tree of the text, parameterized matching is solved in $O(n|\Pi|log|\Sigma|)$ time.

Idury and Schäffer use a modified AC automaton to perform parameterized dynamic dictionary matching [43]. They show that it is possible to design an automaton algorithm to solve the problem. An interesting feature of their algorithm is that it works with two dual representations of a parameterized pattern, *prev* and *next*, and one of the representations is computed only implicitly but never stored or printed. The time complexity of their algorithm depends on the size of the text and the number of characters in the dictionary. The automaton is stored in space linear in the dictionary size.

Amir et. al. point out [5] that Baker's algorithm has an overhead that depends on both Σ and Π. They developed an alternative algorithm for p-string matching that requires only $O(n\ log\pi)$ time, where $\pi = min(m, |\Pi|)$. Its complexity does not depend on Σ. Furthermore, they prove that their algorithm achieves optimal bounds. They reduce the p-string matching problem to the m-matching problem that is only concerned with matching the parameter symbols. The algorithm is a modification of KMP in which the definition of the $=$ operator is modified and works with text blocks of length $2m$.

The algorithm of Fredriksson and Mozgovoy [30] for p-string matching is sublinear on average. Their algorithm is based on the Shift-Or and Backward DAWG Matching algorithms. BDM is optimal on average. That is, its average running time is $O(n\,log\,m/m)$. This approach to parameterized pattern matching easily generalizes to the case of multiple patterns.

The 2D parameterized matching problem models the search for color images that accommodates changing color maps. Many algorithms that solve 2D matching do not work for 2D parameterized matching. In 2D parameterized matching there is a strong dependency between pieces of the pattern; they cannot be evaluated individually. Air et. al. [2] solve 2D parameterized matching in time $O(n^2 log^2 m)$. They form a graph for each parameterized character in the pattern, with edges between positions that are the closest occurrences of the character for some linearization of the pattern. Then one dimensional parameterized matching is used on these linearizations to ensure that any two elements that are chained will be evaluated to map to the same text value. Their algorithm uses convolutions for comparison.

The algorithm of [42] is more efficient for 2D parameterized matching in large texts as it achieves time complexity $O(n^2 + m^{2.5} polylog\, m)$. They use the dueling paradigm. Unlike in exact matching, where a single witness suffices to disqualify a candidate, pairs of witnesses are used for parameterized matching.

Suffix arrays are more efficient than suffix trees for many applications. Baker introduced the parameterized suffix tree along with a linear-time construction algorithm. A linear-time direct construction of the parameterized suffix array, with the p-LCP table was presented by Deguchi et. al. [24]. Their algorithm is for strings drawn from Σ and Π, where $|\Pi| = 2$. There is a strong similarity between parameterized strings on a binary alphabet and standard strings. Linear-time algorithms for direct construction of standard suffix arrays do not work for the p-suffix array. This is because the lexicographic order between suffixes of two strings is not necessarily preserved, even when they share a common prefix. Their construction algorithm uses the dual of the *prev* function, *fw*, which finds the distance to the closest occurrence of each character to its right. *fw* is the *next* function used by Idury and Schäffer. The suffix array of *fw* is constructed directly from the suffix array of the original string.

Apostolico and Giancarlo [8] investigated periodicity and repetition in parameterized strings. When it comes to periodicity, parameterized strings are truly different objects than ordinary strings. Yet, binary parameterized strings behave in much the same way as non-parameterized ones with respect to periodicity and repetitions. Thus, many constructs that were originally devised for ordinary strings naturally extend to parameterized binary strings.

Baker discusses methods for parameterized edit distance based on the operations of insertion, delete, and parameterized match [10]. Parameterized matching replaces global substitution. Each operation has unit cost. Her algorithm finds the parameterized edit distance, with efficiency close to that of algorithms for computing the classical edit distance. It turns out that the operation of parameterized replacement relaxes the problem to an easier problem. This is because two substrings that participate in two parameterized replacements are independent of each other, in the parameterized sense.

The problem of approximate parameterized matching is addressed by Hazay et. al. [41]. They allow up to k mismatches. When the pattern and text are of the same size, the problem is equivalent to the maximum matching problem on graphs. Their algorithm solves the problem in $O(nk^{1.5} + mk\, log\, m)$ time. The results extend to 2D approximate parameterized matching.

6 Useful Terminology

6.1 Periodicity

Computing the period of a string is a crucial task in the pattern preprocessing stage of most string matching algorithms. It is stored in different forms by efficient algorithms. The sequential KMP and BM methods compute a failure table, and the parallel string matching algorithms use witnesses.

A periodic pattern may contain several locations where the pattern can be superimposed on itself without mismatch. We say a pattern is *non-periodic* if the origin is the only position before $\frac{m}{2}$ at which the pattern can be superimposed without mismatch. If we know that a pattern is aperiodic, we could narrow down the number of *candidates* for a pattern occurrence in the text in a manner that insures that they are all “sufficiently far” from each other. Verification of a candidate can follow in a naive sequence of character-by-character comparison, yet the time would still be linear because the candidates do not overlap.

In a periodic string, a smallest period can be found whose concatenation generates the entire string. A string S is periodic if its longest prefix that is also a suffix is at least half the length of S. A proper suffix that is also a prefix of a string is called a *border*. There is duality between periods and borders. The longest border of a string corresponds to its shortest period.

More formally, a string S is *periodic* if $S = \pi'\pi^k$ where π' is a (possibly null) proper suffix of π, and $k \geq 2$. A periodic string S can be expressed as $\pi'\pi^k$ for one unique primitive π. A string π is *primitive* if there is no prefix u of π for which $\exists k \geq 2$ such that $\pi = u^k$. We refer to both π and $|\pi|$ as "the period" of S, although S can have several non-primitive periods. The period of S can also be defined as $|S| - b$ where b is the longest border of S.

For example, consider the periodic string S = *abcabcabcab*. The longest border of S is $b =$*abcabcab*. Since $|b| \geq \frac{|S|}{2}$, S is periodic. $\pi = abc$ is the period of S. Another way of concluding that S is periodic is by the observation that $|\pi| < \frac{|S|}{2}$.

Main and Lorentz published an algorithm in 1984 that finds all periodic substrings of a pattern of size n in $O(n\ logn)$ time [56]. Their algorithm uses a divide and conquer approach to find the repetitions that cross the middle character and then those on either side of the middle character.

6.2 2D Periodicity

Multidimensional periodicity is not as simple as it is in strings. Amir and Benson defined the notion of two-dimensional periodicity based on self-overlap [3]. Although the concepts are developed for square arrays, they can be generalized to rectangular arrays quite easily. Let A be a two-dimensional array. A prefix of A is a rectangular subarray that contains one corner element of A. A sufix of A is a rectangular subarray that contains the diagonally opposite corner. A is periodic if the largest prefix that is also a suffix has dimensions greater than half those of A. This implies that A may overlap itself if the prefix of one copy of A is aligned with the suffix of a second copy of A. Figure 3 of [3] depicts two-dimensional periodicity.

Because of the symmetry, we assign the prefix to either the upper left or lower left corners of A. This clearly gives us two directions in which A can be periodic. We classify the type of periodicity of A based on whether it has periodicity in either or both of these directions.

We can divide the array into four quadrants, labeled in a counterclockwise direction from upper left, quadrants *I*, *II*, *III*, and *IV*. Given two copies of A, one directly on top of the other, the two copies are said to be *in register* when some of the elements overlap, and overlapping elements contain the same symbol. If we can slide the upper copy over the lower copy to a point where the copies are again in register, then at least one of the corner elements of the upper array will overlap an element of the lower array. The element in the lower copy that is under this corner is the *source*. We say that the array is *quadrant I symmetric* if an overlapping corner is element $A(0,0)$. The element in the lower copy is a *quadrant I source*. Quadrant *II*, *III*, and *IV* symmetries and sources are similarly defined.

Let the array be quadrant I symmetric and let the upper and lower copies be in register when element $A(0,0)$ overlaps element $A(r,c)$, the source. Then there exists a *quadrant I symmetry vector* $\vec{v_I} = r\vec{y} + c\vec{x}$ where $\vec{x}$ is the horizontal unit vector in the direction of increasing column index and $\vec{y}$ is the vertical unit vector in the direction of increasing row index. We can define symmetry vectors for the other quadrants similarly.

The 2D periodicity of a matrix is described by its symmetry vectors. The length of a symmetry vector is either the difference between the row coordinates of its endpoints or the difference between the column coordinates of its endpoints, whichever is larger. A symmetry vector is periodic if its length is $< m/2$. For the classification scheme, we need to pick the shortest symmetry vector in each of quadrants I and *II*. The four classes of 2D periodicity depend on the arrangement of sources in their respective quadrants.

A 2D pattern can be categorized by one of the four periodicity classes:

- A *non-periodic* pattern has no periodic vectors.
- The sources of a *line periodic* pattern fall on a straight line. One of quadrants I and II has a periodic vector and the other does not.
- The sources of a *lattice periodic* pattern fall on a lattice. Both quadrants I and II have a periodic basis vector which form a lattice.
- In a *radiant periodic* pattern, the sources are not along a line. The sources fall on several lines which radiate from the quadrant's corner.

Using a suffix tree of the linearized matrix, a serial algorithm can find the 2D periodicity of a matrix in linear time and space, i.e. $O(m^2)$. The process can be parallelized within the same time

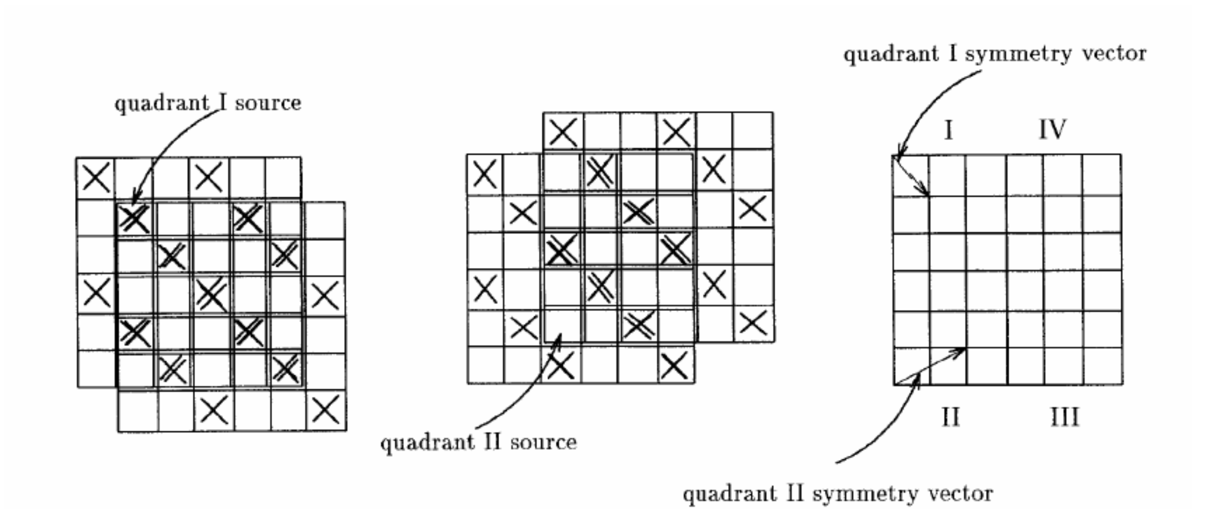

Figure 3: 2D periodicity: sources in quadrants I and II and their respective symmetry vectors

and space complexity [3].

6.3 Conjugacy

Two words, x and y, are said to be *conjugate* if $x = uv$, $y = vu$ for some words u, v. Two words are conjugate if they differ only by a cyclic permutation of their letters. We can determine whether x and y are conjugate in time proportional to the combined lengths of x and y [55].

A *Lyndon word* is a word which is strictly smaller than any of its conjugates for the alphabetic ordering. In other terms, a word x is a Lyndon word if for any factorization $x = uv$ with u, v nonempty, one has $uv < vu$. A Lyndon word is primitive.

Any primitive word has a conjugate which is a Lyndon word, namely its least conjugate. Computing the smallest conjugate of a word is a practical way to compute a standard representative of the conjugacy class of a word. This procedure, often called *canonization*, can de done in linear time [55].

6.4 Empirical Entropy

Consider a text $T[n]$ with symbols from an alphabet Σ. In the Kolmogorov sense it takes $O(nlog|\Sigma|)$ bits to store T. A text that exhibits some kind of regularity can be stored in less space. One can describe the *compressibility* of T without making any probabilistic assumptions about the data. Manzini defines *empirical entropy* [58]:

Let n_a be the number of times that character a occurs in T. Assume $0 \log 0 = 0$. The *zeroth* *order empirical entropy* of T is defined as

$$H_0 = H_0(T) = -\sum_{a\in\Sigma} \frac{n_a}{n} \, log \, \left(\frac{n_a}{n}\right).$$

$H_0 n$ is a lower bound for compression schemes that assign a fixed codeword to each symbol in T. If one takes into account the k characters which precede the character to be encoded next, the definition generalizes to k'th order empirical entropy.

Let w_T denote the string of symbols following the occurrences of w in T, reading T from left to right. The *k'th order empirical entropy* of T is defined as

$$H_k = H_k(T) = \frac{1}{n} \sum_{w\in\Sigma^k} |w_T| H_0(w_T).$$

$H_k n$ is a lower bound for compression schemes that assign fixed code-words to symbols based on the preceding k characters. We have the hierarchy $H_k \leq H_{k-1} \leq \cdots H_0 \leq log|\Sigma|$.

7 Indexing

Indexing is an important paradigm in searching. The text is preprocessed so that queries of the form "*does pattern P occur in text T?*" are answered in time proportional to the pattern, rather than the text. Two popular indexing structures are the suffix tree and the suffix array. These data structures enable efficient solutions to many common string problems. Both suffix trees and suffix arrays can be generalized to two or more dimensions. Recent work has compressed these data structures, formed dynamic data structures and developed full-text indices. A full-text index gathers all the relevant information about text so that the actual text can be discarded; it can attain better space complexity than the original text.

7.1 Suffix Tree

A suffix tree is a tree-like structure that represents all suffixes of the text under consideration. Once a suffix tree is constructed for a text, the search time for a pattern is linear in the pattern size; that is, without considering the one-time preprocessing of the text. A special character is appended to the string before construction of the suffix tree. This guarantees that a suffix of the string cannot be a substring of another suffix.

A suffix tree can be used to index several patterns. Either the patterns can be concatenated with unique characters separating them or a generalized suffix tree can be constructed. The generalized suffix tree, as described by Gusfield [39] does not mark suffixes that span several patterns. It combines the suffixes of the individual patterns in a single data structure.

Suffix trees can be used to solve a variety of string problems. A substring of text can be considered a prefix of some suffix of the text. Suffix trees can be applied to the problems of string search, longest palindrome, longest common substring, and longest repeated substring. Harel and Tarjan demonstrated that an n-node tree can be preprocessed in $O(n)$ time to answer longest common ancestor (LCA) queries in constant time [40]. The LCA of two nodes in a suffix tree corresponds to the longest common prefix of the strings they represent.

Each suffix of a string corresponds to a leaf in its suffix tree. A suffix tree is a concise representation of a trie. In a trie, or a k-ary position tree, each edge represents one character of the alphabet. To conserve space, a PATRICIA trie collapses a node that has only one outgoing edge. In the compacted representation, an edge can be labeled with more than one character. A suffix tree requires even less space than the compacted PATRICIA trie. The suffix tree for a string of size n is represented in $O(n)$ space by using indices of constant size, rather than substrings of arbitrary length, to label the edges of the tree.

The naive algorithm for constructing a suffix tree requires quadratic time with respect to the size of the input. Linear-time construction of the suffix tree was first introduced by Weiner in 1973 as a position tree [70]. The construction was greatly simplified by McCreight in 1976 [59]. Weiner's algorithm scans the text from right to left and begins with the shortest suffix, while McCreight's scans the text from left to right and initializes the data structure with the longest suffix. In the 1990's, Ukkonen provided an online construction of suffix trees in linear time [67]. Ukkonen overcame the problem of extending each suffix on each iteration by introducing a special edge pointer, *, to represent the current end of the string. For more details on the development of

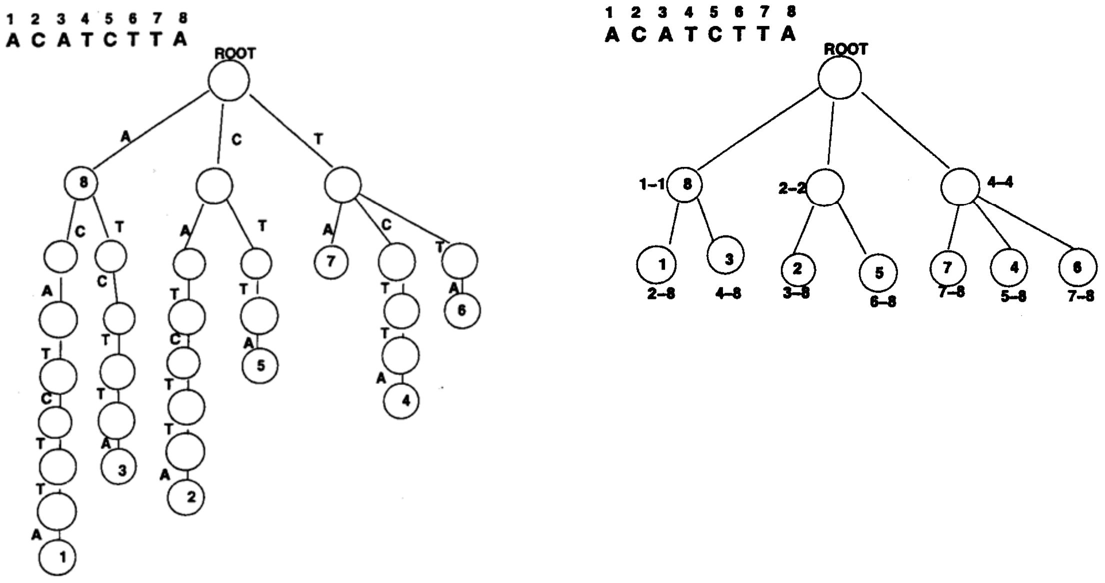

Figure 4: Suffix Trie (left) and Suffix Tree (right) of the string ACATCTTA

the linear-time suffix tree construction algorithms and the distinctions between them, see [36].

Suffix links are an implementation trick necessary to achieve linear time and space bounds in suffix tree construction algorithms. Suffix links allow an algorithm to move quickly to a distant part of the tree. A suffix link is a pointer from an internal node labeled xS to another internal node labeled S, where x is an arbitrary character and S is a possibly empty substring. The suffix links of Weiner's algorithm are alphabet dependent as they work in reverse to represent the insertion of any character before a suffix.

Kosaraju improved the suffix tree of Weiner to an alphabet independent data structure. Amir and Nor [6] combined Kosaraju's quasi-real time algorithm [54] with Ukkoknen's algorithm in 2008, to produce a real-time linear suffix tree construction algorithm. That is, they deamortized an existing online algorithm. Their main contribution is in changing the order in which suffixes are inserted into the tree.

The directed acyclic graph, or DAG, is a further compaction of the suffix tree. Isomorphic subgraphs are merged. Hence, directed edges are necessary. The DAG saves space by eliminating redundancy in the graph, but it introduces complexity that increases the time of some applications

[22].

7.1.1 Dynamic Suffix Tree

The suffix tree can be generalized to represent a set of strings in such a way that strings can be inserted or deleted efficiently. We call such a suffix tree a dynamic suffix tree. An efficient implementation for the dynamic suffix tree is described in [16]. The update operations take time proportional to the string being inserted or removed from the tree. Yet, the tree never stores a reference to a string that has been removed, and the space complexity is bounded by the total length of the strings stored in the tree. Their method requires a two-way pointer for each edge, which can be done in space linear in the size of the tree.

7.1.2 Compressed Suffix Tree

The suffix tree that was described until now requires $O(n)$ words or $O(n\, log\, n)$ bits. Grossi and Vitter showed that this can be condensed to $O(n)$ bits, forming the compressed suffix tree [38]. It is based upon a construction algorithm for compressed suffix arrays that they introduce.

Sadakane introduced new data structures for compressed suffix trees that occupy $O(n\, log|\Sigma|)$ [65]. This is the first linear-size data structure for suffix trees that supports all operations efficiently. Any algorithm running on a suffix tree can also be executed on this compressed suffix tree with a slight slowdown of a factor of $polylog(n)$. It is also based on a compressed suffix array, CSA, that occupies $O(n)$ bits.

7.2 Suffix Array

A suffix array stores the lexicographic order of the suffixes of the text under consideration. The suffix array achieves greater space efficiency than the suffix tree. Augmented with an LCP array to store the longest common prefix between adjacent suffixes, a binary search on the suffix array can locate all instances of a pattern in the text.

The naive approach to suffix array construction is to sort the suffixes using a string sorting algorithm. This approach ignores the fact that the suffixes are related. The worst case complexity of such an algorithm is proportional to the sum of the lengths of all suffixes, $O(n^2)$. A suffix array can be built by preorder traversal of a suffix tree for the same text, and the LCP array by constant-time lowest common ancestor queries on the tree. This indirect construction does not achieve better space complexity than the suffix tree since the suffix tree is constructed along the way.

Suffix	11	8	5	2	1	10	9	7	4	6	3
Index	1	2	3	4	5	6	7	8	9	10	11

Table 1: Suffix Array of the string MISSISSIPPI

Manber and Myers introduced the suffix array in 1993 [57]. Their algorithm employs a sort and search paradigm. Simply sorting the suffixes is not enough to produce an efficient pattern search mechanism. The precomputation and utilization of the longest common prefix (LCP) between the suffixes make the technique very efficient. In the worst case, preprocessing text of length n requires $O(n\, log\, n)$ time to sort the suffixes and $O(n)$ time to compute the LCPs. The preprocessing is all done in $O(n)$ space. Their algorithm achieves expected running time of $O(n)$, independent of the alphabet size. Searching for a pattern of length m takes $O(m\, log\, n)$ time using a binary search on the suffix array and the text. This is improved to $O(m + log\, n)$ time using the LCP array. The enhanced suffix array searches for a pattern of length m in $O(m|\Sigma|)$ time. This requires an additional array, the *child* table.

The efficient suffix sort is based on the *doubling technique* of Karp, Miller, and Rosenberg [48]. The idea is to assign a *rank* to all substrings whose length is a power of two. The rank tells the lexicographic order of the substring among substrings of the same length. The method iteratively applies a linear sorting mechanism, such as bucket sort or radix sort. The algorithm sorts each suffix by its first k letters, then doubles k to use the results of the previous round. There are $log\, n$ phases of sorting. In total, $O(n\, log\, n)$ work is done to sort the suffixes. Since the method is iterative, the space is bounded by the text size. Once the suffix array has been constructed, the longest common prefix (LCP) table can be created in linear time and space.

In 2003, three different algorithms were introduced to directly construct a suffix array in linear time. They were published by of Kärkkäinen and Sanders [45], Ko and Aluru [53], and Kim et. al.

[50]. Kärkkäinen and Sanders apply a divide and conquer approach to the algorithm of Manber and Myers. First the suffix array of the suffixes starting at positions $i \, mod \, 3 \neq 0$ is constructed, then a suffix array of the other third of the suffixes is constructed. The two suffix arrays are merged by comparison.

A suffix array with an LCP array can simulate the bottom-up traversal of its suffix tree and can answer LCA queries. With an additional table, an enhanced suffix array can simulate top-down traversal of a suffix tree for a full range of suffix tree functionality. This can all be done in time and space linear in the size of the text. The LCP array can be computed by efficient range minimum queries.

Suffix sorting algorithms can be used to perform the Burrows-Wheeler transform (BWT). The Burrows-Wheeler transformation permutes the order of the characters and is easily reversed to obtain the original string. The BWT requires sorting cyclic permutations of a string, not suffixes. We can append to the string a special character which sorts lexicographically before every other character. Sorting cyclic permutations is then equivalent to sorting suffixes. The BWT is more readily compressed than the original string, with either run-length encoding or move-to-front encoding. The BWT array can be computed from the suffix array using the equations

$$BWT[i] = T[SA[i] - 1], \quad \text{when } SA[i] \neq 0$$
$$BWT[i] = T[n - 1], \quad \text{when } SA[i] = 0$$

7.2.1 Dynamic Suffix Array

Recent work by Salson et. al. [66] proposes an algorithm for dynamically updating the suffix array of a text that has been edited after the construction of its suffix array. This is more efficient than rebuilding the suffix array from scratch. Even if the theoretical worst-case time complexity is $O(n \, log \, n)$, it appears to perform much better in practice than the quickest methods. The online version has yet to be developed.

7.2.2 Compressed Suffix Array

Since the suffix array is an array of n indices, it can be stored in $n \, log \, n$ bits. The compressed suffix array was introduced by Grossi and Vitter [38] to reduce the size of the suffix array from $n \, log \, n$

bits to $O(n\, log|\Sigma|)$ bits. This is at the expense of increasing access time from $O(1)$ to $O(log^{\epsilon}n)$ time where ϵ is any constant with $0 < \epsilon < 1$.

Sadakane modified the compressed suffix array so that it is a self-index [64]. That is, the text can be discarded and the index suffices to answer queries. He also reduced the size of the structure from $O(n\, log|\Sigma|)$ bits to $O(n\, logH_0)$ bits. Pattern matching using his compressed suffix array has the same time complexity as in the uncompressed suffix array. Grossi et. al. [37] further reduced its size to $nH_h + o(n)$ bits for any $h \leq \alpha log_{|\Sigma|}n$ with $0 < \alpha < 1$. A pattern can be found in $O(m\, log|\Sigma| + polylog(n))$ time.

Ferragina and Manzini developed the first compressed suffix array to encode the index size with respect to the high-order empirical entropy [28]. Their self-indexing data structure is known as the FM-index. The FM-index is based on the Burrows-Wheeler transform and uses backward searching. The compressed self-index exploits the compressibility of the text so its size is a function of the compressed text length. Yet an index supports more functionality than standard text compression. Navarro and Makinen improved the FM-index [61].

Many self-indexes state their space requirement as a function of the empirical entropy of the indexed text. This is useful because it gives a measure of the index size with respect to the size achieved by the best k^{th}-order compressor, thus relating the index size to the compressibility of the text. Empirical entropy is defined in Section 6.4.

7.3 Word-Based Index

Recently, optimal time and space construction algorithms have been developed for the word-based suffix tree and suffix array. These structures can store a subset of text positions, namely the ones that correspond to word (or pattern) beginnings. A word-based index naturally applies to a word or phrase level index for documents written in a natural language. We assume that a text of size n contains k distinct words delimited by predetermined delimiters. It is easy to construct a word-based index if $O(n)$ space is allowed at construction time. Simply index the enire text, and then discard positions which do not correspond to word beginnings. A more clever approach limits the working space to $O(k)$, the size of the word index. The $O(n)$ construction time cannot be improved upon since this is the time needed to scan the input. It should be noted that the more general problem of constructing a sparse index is not as simple as that of a word-based index.

Traditional linear-time suffix tree construction algorithms rely heavily on the fact that all suffixes are to be inserted in the tree. Thus, it was a challenge to develop a word-based suffix tree within $O(k)$ working space. Karkkainen and Ukkonen show how to adapt Ukkonen's suffix tree construction algorithm to efficiently construct a sparse suffix tree [47]. A sparse suffix tree indexes the text at regular intervals, i.e., every j^{th} suffix. Their construction algorithm requires only $O(\frac{n}{j})$ space, with the usual $O(n)$ time. The word based suffix tree is even more innovative. Anderson et. al. showed that the word-based suffix tree can be constructed in $O(n)$ expected time and $O(k)$ working space [7]. In 2006, Inenaga and Takeda [44] improved this result by providing an online algorithm which runs in $O(n)$ time and $O(k)$ additional working space in the worst case. Their algorithm is based on Ukkonen's online suffix tree construction algorithm.

Ferragina and Fischer presented the word suffix array in 2007 [26]. Their algorithm is time and space optimal. The construction takes $O(n)$ time and only requires $O(k)$ additional space. The word-based LCP array can also be computed in $O(n)$ time and $O(k)$ space. The word-based suffix tree is easily constructed from the word-based suffix and LCP arrays so the algorithm of [26] is an alternative to [44].

As far as pattern matching queries are concerned, it is easy to adapt to word-based suffix arrays the classical pattern searches over full-text suffix arrays. Since a suffix array is lexicographically sorted, searching takes $O(n \, log \, k)$ time using binary search. Like the original suffix array of [57], longest common prefixes can be precomputed so that searching the word-based suffix array takes $O(n + log \, k)$ time. The extra space needed is $O(k)$.

8 Research Directions

This survey presents some of the recent advances in pattern matching algorithms. There are still a large number of open problems that have practical applications. Our research has taken the direction of developing a *small space dictionary matching algorithm.* We have also been seeking a time-space optimal solution to 2D compressed dictionary matching.

A variety of fast dictionary matching algorithms for one and two dimensional data exist. Nonetheless, we have not found a deterministic method that works in linear time and small space. For single pattern matching there are variations of KMP that can work in constant space. We believe that recent results in compressed self-indexing will facilitate the development of a solution to the small space dictionary matching problem.

The compressed matching problem is that of identifying occurrences of a compressed pattern in a compressed text. The nature of most compression algorithms requires matching to be performed on uncompressed data. A factor that often motivates data compression is the lack of space to store the data in its original form. Thus, there is the interest in strongly inplace compressed pattern matching. An algorithm is called *strongly inplace* if the extra space it uses is proportional to the optimal compression of all patterns of such size. We are currently working on a strongly inplace algorithm for 2D dictionary matching in a Lempel-Ziv compressed text.

References

[1] A. V. Aho and M. J. Corasick. Efficient string matching: an aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975.

[2] A. Amir, Y. Aumann, R. Cole, M. Lewenstein, and E. Porat. Function matching: Algorithms, applications, and a lower bound. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 2003.

[3] A. Amir and G. Benson. Two-dimensional periodicity and its applications. In *SODA '92: Proceedings of the third annual ACM-SIAM symposium on Discrete algorithms*, pages 440–452, Philadelphia, PA, USA, 1992. Society for Industrial and Applied Mathematics.

[4] A. Amir, G. Benson, and M. Farach. An alphabet independent approach to two-dimensional pattern matching. *SICOMP: SIAM Journal on Computing*, 23, 1994.

[5] A. Amir, M. Farach, and S. Muthukrishnan. Alphabet dependence in parameterized matching. *IPL: Information Processing Letters*, 49, 1994.

[6] A. Amir and I. Nor. Real-time indexing over fixed finite alphabets. In *SODA '08: Proceedings of the nineteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 1086–1095, Philadelphia, PA, USA, 2008. Society for Industrial and Applied Mathematics.

[7] A. Andersson, N. J. Larsson, and K. Swanson. Suffix trees on words. *ALGRTHMICA: Algorithmica*, 23, 1999.

[8] A. Apostolico and R. Giancarlo. Periodicity and repetitions in parameterized strings. *Discrete Appl. Math.*, 156(9):1389–1398, 2008.

[9] B. S. Baker. A theory of parameterized pattern matching: algorithms and applications. In *STOC*, pages 71–80, 1993.

[10] "'B. S. Baker". "parameterized diff". In *"SODA '99: Proceedings of the tenth annual ACM-SIAM symposium on Discrete algorithms"*, pages 854–855, Philadelphia, PA, USA, "1999". Society for Industrial and Applied Mathematics.

[11] T. J. Baker. A technique for extending rapid exact-match string matching to arrays of more than one dimension. *SIAM J. Comp*, 3(7):533–541, 1978.

[12] R. S. Bird. Two dimensional pattern matching. *Information Processing Letters*, 6(5):168–170, 1977.

[13] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10):762–772, 1977.

[14] D. Breslauer. Dictionary-matching on unbounded alphabets: Uniform length dictionaries. *Combinatorial Pattern Matching*, 184, 1995.

[15] H. L. Chan, W. K. Hon, T. W. Lam, and K. Sadakane. Compressed indexes for dynamic text collections. *ACM Transactions on Algorithms*, 3(2), 2007.

[16] Y. Choi and T. W. Lam. Dynamic suffix tree and two-dimensional texts management. *IPL: Information Processing Letters*, 61, 1997.

[17] R. Cole, Z. Galil, R. Hariharan, S. Muthukrishnan, and K. Park. Parallel two dimensional witness computation. *INFCTRL: Information and Computation (formerly Information and Control)*, 188, 2004.

[18] B. Commentz-Walter. A string matching algorithm fast on the average. In *ICALP*, pages 118–132, 1979.

[19] M. Crochemore, L. Gasieniec, R. Hariharan, S. Muthukrishnan, and W. Rytter. A constant time optimal parallel algorithm for two-dimensional pattern matching. *SIAM J. Comput.*, 27(3):668–681, 1998.

[20] M. Crochemore, L. Gasieniec, W. Plandowski, and W. Rytter. Two-dimensional pattern matching in linear time and small space. In *STACS: Annual Symposium on Theoretical Aspects of Computer Science*, 1995.

[21] M. Crochemore, L. Gasieniec, and W. Rytter. Two-dimensional pattern matching by sampling. *IPL: Information Processing Letters*, 46, 1993.

[22] M. Crochemore, C. Hancart, and T. Lecroq. *Algorithms on Strings.* Cambridge University Press, 2007.

[23] M. Crochemore and D. Perrin. Two-way string-matching. *J. ACM*, 38(3):650–674, 1991.

[24] S. Deguchi, F. Higashijima, H. Bannai, S. Inenaga, and M. Takeda. Parameterized suffix arrays for binary strings. In Jan Holub and Jan Žďárek, editors, *Proceedings of the Prague Stringology Conference 2008*, pages 84–94, Czech Technical University in Prague, Czech Republic, 2008.

[25] S. Dori and G. M. Landau. Construction of aho corasick automaton in linear time for integer alphabets. *IPL: Information Processing Letters*, 98, 2006.

[26] P. Ferragina and J. Fischer. Suffix arrays on words. In Bin Ma and Kaizhong Zhang, editors, *Combinatorial Pattern Matching, 18th Annual Symposium, CPM 2007, London, Canada, July 9-11, 2007, Proceedings*, volume 4580 of *Lecture Notes in Computer Science*, pages 328–339. Springer, 2007.

[27] P. Ferragina and F. Luccio. On the parallel dynamic dictionary matching problem: New results with applications. In *ESA '96: Proceedings of the Fourth Annual European Symposium on Algorithms*, pages 261–275, London, UK, 1996. Springer-Verlag.

[28] P. Ferragina and G. Manzini. Indexing compressed text. *J. ACM*, 52(4):552–581, 2005.

[29] K. Fredriksson. Succinct backward-DAWG-matching. *ACM Journal of Experimental Algorithmics*, 13, 2008.

[30] K. Fredriksson and M. Mozgovoy. Efficient parameterized string matching. *IPL: Information Processing Letters*, 100, 2006.

[31] Z. Galil. A constant-time optimal parallel string-matching algorithm. In *STOC '92: Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 69–76, New York, NY, USA, 1992. ACM.

[32] Z. Galil and K. Park. Alphabet-independent two-dimensional witness computation. *SIAM J. Comput.*, 25(5):907–935, 1996.

[33] Z. Galil and J. Seiferas. Time-space-optimal string matching (preliminary report). In *STOC '81: Proceedings of the thirteenth annual ACM symposium on Theory of computing*, pages 106–113, New York, NY, USA, 1981. ACM.

[34] L. Gasieniec and R. M. Kolpakov. Real-time string matching in sublinear space. In *CPM*, pages 117–129, 2004.

[35] L. Gasieniec, W. Plandowski, and W. Rytter. Constant-space string matching with smaller number of comparisons: Sequential sampling. In *CPM: 6th Symposium on Combinatorial Pattern Matching*, 1995.

[36] R. Giegerich and S. Kurtz. From ukkonen to mccreight and weiner: A unifying view of linear-time suffix tree construction. *Algorithmica*, 19(3):331–353, 1997.

[37] R. Grossi, A. Gupta, and J. S. Vitter. High-order entropy-compressed text indexes. In *SODA '03: Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 841–850, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[38] R. Grossi and J. S. Vitter. Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract). In *STOC*, pages 397–406, 2000.

[39] D. Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology.* Cambridge University Press, 1997.

[40] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SICOMP: SIAM Journal on Computing*, 13, 1984.

[41] C. Hazay, M. Lewenstein, and D. Sokol. Approximate parameterized matching. *ACM Transactions on Algorithms*, 3(3), 2007.

[42] C. Hazay, M. Lewenstein, and D. Tsur. Two dimensional parameterized matching. In *CPM*, pages 266–279, 2005.

[43] R. M. Idury and A. A. Schaffer. Multiple matching of parameterized patterns. In *CPM: 5th Symposium on Combinatorial Pattern Matching*, 1994.

[44] S. Inenaga and M. Takeda. On-line linear-time construction of word suffix trees. In *in Proc. 17th Ann. Symp. on Combinatorial Pattern Matching (CPM06*, pages 60–71. Springer-Verlag, 2006.

[45] J. Karkkainen and P. Sanders. Simple linear work suffix array construction. In *ICALP: Annual International Colloquium on Automata, Languages and Programming*, 2003.

[46] J. Kärkkäinen and E. Ukkonen. Two and higher dimensional pattern matching in optimal expected time. In *SODA '94: Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, pages 715–723, Philadelphia, PA, USA, 1994. Society for Industrial and Applied Mathematics.

[47] J. Karkkainen and E. Ukkonen. Sparse suffix trees. In *COCOON: Annual International Conference on Computing and Combinatorics*, 1996.

[48] R. M. Karp, R. E. Miller, and A. L. Rosenberg. Rapid identification of repeated patterns in strings, trees and arrays. In *4th*, pages 125–136, Denver, CO, 1972.

[49] R. M. Karp and M. O. Rabin. Efficient randomized pattern-matching algorithms. *IBMJRD: IBM Journal of Research and Development*, 31, 1987.

[50] D. K. Kim, J. S. Sim, H. Park, and K. Park. Linear-time construction of suffix arrays. In *Proc. Annual Symp. on Combinatorial Pattern Matching*, volume 2676 of *LNCS*, pages 186–199. Springer Verlag, 2003.

[51] S. Kim and Y. Kim. A fast multiple string-pattern matching algorithm. In *In Proceedings of 17th AoM/IAoM Conference on Computer Science*, 1999.

[52] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM J. Comput.*, 6(2):323–350, 1977.

[53] P. Ko and S. Aluru. Space efficient linear time construction of suffix arrays. In *Proc. Annual Symposium on Combinatorial Pattern Matching*, volume 2676, pages 200–210. Springer Verlag, 2003.

[54] S. R. Kosaraju. Real-time pattern matching and quasi-real-time construction of suffix tree. In *Proceedings of the 26th ACM Symposium on the Theory of Computing*, pages 310–316, Montréal, Canada, 1994. ACM Press.

[55] M. Lothaire. *Applied Combinatorics on Words (Encyclopedia of Mathematics and its Applications)*. Cambridge University Press, New York, NY, USA, 2005.

[56] M. G. Main and R. J. Lorentz. An O(n log n) algorithm for finding all repetitions in a string. *ALGORITHMS: Journal of Algorithms*, 5, 1984.

[57] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. *SICOMP: SIAM Journal on Computing*, 22, 1993.

[58] G. Manzini. An analysis of the burrows-wheeler transform. *JACM: Journal of the ACM*, 48, 2001.

[59] E. M. McCreight. A space-economical suffix tree construction algorithm. *J. ACM*, 23(2):262–272, 1976.

[60] S. Muthukrishnan and K. Palem. Highly efficient dictionary matching in parallel. In *SPAA '93: Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 69–78, New York, NY, USA, 1993. ACM.

[61] G. Navarro and V. Mäkinen. Compressed full-text indexes. *ACM Comput. Surv.*, 39(1), 2007.

[62] G. Navarro and M. Raffinot. Fast and flexible string matching by combining bit-parallelism and suffix automata. *ACM Journal of Experimental Algorithmics*, 5:4, 2000.

[63] W. Rytter. On maximal suffixes and constant-space linear-time versions of kmp algorithm. *Theor. Comput. Sci.*, 299(1-3):763–774, 2003.

[64] K. Sadakane. New text indexing functionalities of the compressed suffix arrays. *J. Algorithms*, 48(2):294–313, 2003.

[65] K. Sadakane. Compressed suffix trees with full functionality. *Theory Comput. Syst*, 41(4):589–607, 2007.

[66] M. Salson, T. Lecroq, M. Leonard, and L. Mouchard. Dynamic extended suffix arrays. submitted to Journal of Discrete Algorithms, January 2009.

[67] E. Ukkonen. On-line construction of suffix trees. *Algorithmica*, 14(3):249–260, September 1995.

[68] U. Vishkin. Optimal parallel pattern matching in strings. *Information and Control*, 67(1-3):91–113, October/November/December 1985.

[69] U. Vishkin. Deterministic sampling—A new technique for fast pattern matching. *SIAM Journal on Computing*, 20(1):22–40, February 1991.

[70] P. Weiner. Linear pattern matching algorithm. In *Proceedings of the 14th Annual IEEE Symposium on Switching and Automata Theory*, pages 1–11, Washington, DC, 1973.

[71] S. Wu and U. Manber. A fast algorithm for multi-pattern searching. Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, AZ, 1994.