Parameterized Matching: Solutions and Extensions

Juan Mendivelso\textsuperscript{1} and Yoan Pinzón\textsuperscript{2}

\textsuperscript{1} Fundación Universitaria Konrad Lorenz, Bogotá, Colombia
juanc.mendivelso@konradlorenz.edu.co
\textsuperscript{2} Universidad Nacional de Colombia, Bogotá, Colombia
ypinzon@unal.edu.co

Abstract. Parameterized matching is a string searching variant in which two equal-length strings parameterized-match if there exists a bijective function $g$ for which every text symbol in one string is equal to the image under $g$ of the corresponding symbol in the other string. Baker was the first researcher to have addressed this problem \cite{Baker}, and many others since have followed Baker’s work. She did, indeed, open up a wide field of extensive research. Over the years, other lines of research that have been pursued are: parameterized matching under edit and Hamming distance, multiple parameterized matching, 2-dimensional parameterized matching, structural matching and function matching. This accelerated research could only be justified by the usefulness of its practical applications such as in software maintenance, image processing and bioinformatics. In this paper, we present an overview of the most notable contributions in this area.

Keywords: parameterized matching, string matching, software maintenance, function matching

1 Introduction

String searching is inarguably one of the foremost computational primitives \cite{Fischer}. The input to the string matching problem consists of two strings: the pattern $P = P_{1...m}$ and the text $T = T_{1...n}$. The output should list all the occurrences of the pattern in the text. Given that, for specific applications, it is useful to find inexact occurrences of the pattern, many variants of the string matching problem have been proposed. One of these variants is parameterized matching. Two equal-length strings parameterized-match if there exists a bijective function $g$ for which every text symbol in one string is equal to the image under $g$ of the corresponding symbol in the other string. The symbols in the strings are drawn from two alphabets: the constant alphabet $\Sigma_C$ and the parameter alphabet $\Sigma_P$. The mapping of the symbols from $\Sigma_C$ must be identity.

More formally, two length-$m$ strings $X = X_{1...m}$ and $Y = Y_{1...m}$, defined over $(\Sigma_C \cup \Sigma_P)^*$, are said to be a parameterized-match, or a $p$-match, if there exists a bijective function $g : \Sigma_C \cup \Sigma_P \mapsto \Sigma_C \cup \Sigma_P$ such that $g(Y_i) = X_i$, $1 \leq i \leq m$ so that $g$ is identity for the the symbols from $\Sigma_C$. Note that, in the worst case, $g$ can be chosen from $|\Sigma_P|!$ different possible mapping functions. For instance, given the constant alphabet $\Sigma_C = \{b\}$ and the parameter alphabet $\Sigma_P = \{x, y, z\}$, let us consider two strings $X = xbxyxbx$ and $Y = zbxxzbz$ defined over $\Sigma_C \cup \Sigma_P$. We can say that $X$ and $Y$ are a parameterized-match given that we can map $Y$ to $Y' = xbxyxbx = X$ by means of the bijective function $r : (b, x, y, z) \mapsto (b, y, z, x)$. Notice that the mapping of the symbol $b$, the only symbol from the constant alphabet in this example, is identity.

Furthermore, two equal-length strings $X$ and $Y$ that parameterized-match have the same structure. Let us suppose that $i$ and $j$ are the only occurrences of the symbol $\alpha$ in $Y$. Then, the existence of a bijective function $g$ that maps the symbols in $Y$ to the
symbols in $X$ implies that $g(\alpha) = X_i = X_j = \beta$ and that $\beta$ has no other occurrences in $X$. As this applies for all the distinct symbols $\alpha$ in $Y$, we can conclude that the following facts hold: (i) $X$ and $Y$ have the same number of distinct symbols; (ii) the first occurrence of each distinct symbol $\alpha$ in $Y$ takes place in the same position of the first occurrence of the symbol $g(\alpha)$ in $X$; and (iii) the relative distances among the different occurrences of each $\alpha$ in $Y$ are the same relative distances among the occurrences of $g(\alpha)$ in $X$. Therefore, two strings that parameterized-match are the same except for a systematic change of the symbols.

In this sense, parameterized matching has important applications in different areas. However, it was initially defined as a tool for software maintenance. This was motivated by the observation that programmers introduce duplicate code into large software systems when they add new features or fix bugs. Instead of adapting working sections of code, programmers prefer to copy and slightly modify new instances of those sections in order to avoid making major revisions and introducing new bugs. They do it especially when the working sections were written by another programmer. Then, the code is considered as a sequence of tokens (variables, constants, operands, reserved keywords and procedure names) where the constant alphabet $\Sigma_C$ is comprised by the operands and the reserved keywords while the parameter alphabet $\Sigma_P$ is comprised by the variables, constants and procedures’ names [13].

With time, the amount of duplicate code is highly increased and the code gets larger, more complex and more difficult to maintain. For instance, when a new issue in a determined part of the program is fixed, it will not be automatically fixed in the other copies of that section of code and sometimes they may be hard to find. Experimental results on a large subsystem of over a million lines of code showed that 22% of the lines was involved in parameterized matching [13]. This is a great amount of duplicate code, given that a proportional percentage of the code could be shrunk by using better programming techniques like procedures and functions. A reduction of this magnitude would make the code much more simple and easier to maintain.

In this paper, we review, organize and summarize some of the most important works on parameterized matching. The outline of the article is as follows. The definitions of the different parameterized matching problems are presented in Section 2. The solutions of parameterized matching are reviewed in Section 3 and its extensions are presented in Section 4. Finally, some of the most important applications of this pattern matching variant are shown in Section 5 and the conclusions are drawn in Section 6.

2 Basic Problems

A parameterized string or a $p$–string is defined as a string of symbols in $(\Sigma_C \cup \Sigma_P)^*$. Then, two length–$m$ $p$–strings $X = X_1 \ldots m$ and $Y = Y_1 \ldots m$ are said to be a parameterized–match or a $p$–match, if one $p$–string can be transformed into the other by bijectively renaming its parameters. The basic parameterized matching problem is parameterized pattern matching. It consists of finding all the parameterized–matches of a pattern $P = P_1 \ldots m$ in a text $T = T_1 \ldots n$. Note that, at each position $i$ of $T$, a different $g_i$ can be considered to determine the existence of a parameterized-match between the pattern and the text window starting at position $i$. This problem is also referred as Parameterized Fixed Pattern Matching (PFPM) [54].

Some other problems related to parameterized matching have been defined to be able to support more applications. One of them is finding the maximal $p$–matches of
a p-string text $T = T_{1...n}$ over a threshold length $t$. A maximal p-match is a p-match between a pair of p-substrings (of $T$) that is neither left-extensible nor right-extensible. Let us consider a p-match between $T_{i...i+k}$ and $T_{j...j+k}$. This p-match is said to be left-extensible if $T_{i-1...i+k}$ and $T_{j-1...j+k}$ are a p-match and is right-extensible if $T_{i...i+k+1}$ and $T_{j...j+k+1}$ are a p-match, where $1 \leq i \leq i + k \leq n$, $1 \leq j \leq j + k \leq n$ and $i \neq j$. Note that the length of the strings in a p-match is $k + 1$. Because the output of the problem is the set of maximal p-matches whose length is at least $t$, then $k + 1 \geq t$ for each match.

On the other hand, parameterized matching has been extended to search for multiple parameterized patterns [54]. For a given fixed set $D$ of p-string patterns over $\Sigma_C \cup \Sigma_P$, the Parameterized Multiple Pattern Matching (PMPM) problem consists of preprocessing $D$ as an aid to later determine the p-matches (for all of the patterns in $D$) in a query text $T$. A dynamic variant of this problem, called Parameterized Dynamic Dictionary Matching (PDDM), has also been considered [51]. In this problem, a dictionary $D$ of p-string patterns is preprocessed and maintained with available operations of inserting/deleting patterns into/from $D$ and searching a query text $T$ for p-matches for the patterns currently in $D$.

3 Solutions

The problem of finding the maximal p-matches of a p-string text over a threshold length was the first parameterized matching problem to ever be considered. Baker tackled this problem motivated by the observation that there was a considerable amount of duplicate code in large software systems. Therefore, she presented a program, called DUP [13], as an aid to find all the duplicate sections of code with a minimum length specified by the user. DUP simplifies the problem to an exact matching problem by replacing all the parameters with a determined symbol and then looks for the p-matches among the exact matches found. The algorithm is based on recursions over the suffix tree of the text. Experiments with real data showed that the algorithm is inefficient given that just a few of the exact matches found correspond to p-matches. For this reason, the same author proposed a more elaborate theory [15,17].

A procedure called prev was defined to yield efficient solutions for parameterized matching [17]. Given a length-$m$ p-string $X = X_{1...m}$ defined over $\Sigma_C \cup \Sigma_P$, $\text{prev}(X)$ is a string in $(\Sigma_C \cup \mathbb{N})^*$ where every constant symbol in $X$ remains the same in $\text{prev}(X)$ but the parameters are replaced by non-negative integers: the leftmost occurrence of a determined parameter is represented by a 0 and the other occurrences are represented by the difference in position compared to the previous occurrence of this parameter. The numbers that represent difference in position are called parameter pointers. The time complexity of the computation of $\text{prev}$ is $O(m)$ and the space complexity is $O(|\Sigma_P|)$ by means of a table containing the last occurrence position of each parameter.

Notice that $\text{prev}(X)$ is calculated in such a way that it does not matter what the parameters of $X$ are; what is really relevant is the relative distance among the different occurrences of the same parameter (represented by the parameter pointers) which provides valuable information about the structure of the p-string. Thus, two p-strings $X$ and $Y$ are a p-match, iff $\text{prev}(X) = \text{prev}(Y)$. For example, given $\Sigma_C = \{b\}$, $\Sigma_P = \{x,y,z\}$, $X = xbygzbz$ and $Y = zbxxzbz$, we find that $\text{prev}(X) = 0601462 = \text{prev}(Y)$ and therefore $X$ and $Y$ are a p-match. The $\text{prev}$ of any substring of a p-string
X can be calculated from \( \text{prev}(X) \). This is because any symbol of the substring is the same as in \( \text{prev}(X) \) except when it is a parameter pointer that points to a position before \( i \); in such case, it will correspond to the first occurrence of the parameter in the substring so it must replaced by a 0.

Reminiscing about the use of suffix trees for exact matches in DUP, Baker defined a new data structure called parameterized–suffix tree, or \( p\text{-suffix tree} \), to aid in directly searching for parameterized–matches \([17]\). The \( p\text{-suffix} \) tree of a \( p\text{-string} X = X_{1...m} \) is a compacted trie that stores the \( p\text{-suffixes} \) of \( X \). The \( i\text{-th} \) \( p\text{-suffix} \) of a \( X \) is defined as \( \text{psuffix}(X, i) = \text{prev}(X_{i...m}), \) for \( 1 \leq i \leq m \) \([17]\). So we can calculate each \( p\text{-suffix} \), just like the \( \text{prev} \) of any substring of \( X \), by copying the corresponding symbols of \( \text{prev}(X) \) except when they are parameter pointers that point to a symbol outside the substring (in which case they are replaced by 0).

An algorithm to construct \( p\text{-suffix} \) trees, called \textsc{lazy}, was proposed \([17]\). It is based on McCreight’s algorithm for constructing suffix trees \([65]\). This algorithm is linear in the \( p\text{-string} \) length in both time and space for fixed alphabets. For variable alphabets, the time complexity is \( O(n(|\Sigma_P|\log(|\Sigma_C|+|\Sigma_P|))) \). Later, Baker proposed a new algorithm to build \( p\text{-suffix} \) trees, called \textsc{eager}, that improved the time complexity for variable alphabets to \( O(n(|\Sigma_P|+\log(|\Sigma_C|+|\Sigma_P|))) \) \([15]\). The time complexity of both \textsc{lazy} and \textsc{eager} can be reduced to \( O(n\log n) \), for the variable alphabet case, by using auxiliary data structures like concatenable queues \([2]\) and Sleator-Tarjan dynamic trees \([55]\). However, the use of these structures makes the algorithms not practical. Then, Kosaraju proposed an algorithm whose time complexity is \( O(n\log(|\Sigma_P|+|\Sigma_C|)) \) \([58]\). Other authors devised randomized algorithms \([36,60,61]\).

Two solutions for the parameterized matching problem that use \( p\text{-suffix} \) trees were developed \([15]\). Let us consider the pattern \( p\text{-string} P = P_{1...m} \) and the text \( p\text{-string} T = T_{1...n} \). One of the algorithms consists of following the path determined by the symbols of \( \text{prev}(P) \) on the \( p\text{-suffix} \) tree of \( T \) to find out if \( \text{prev}(P) \) is identical to a length–\( m \) substring of \( T \). Retrieving all the positions in \( T \) where there is a \( p\text{-match} \) with \( P \), for fixed alphabets, takes \( O(m+occ) \) time and \( O(n) \) space, where \( occ \) is the number of \( p\text{-matches} \). The time complexity of this operation is \( O(m \log(|\Sigma_C|+|\Sigma_P|)+occ) \) for variable alphabets. The other algorithm consists of searching in a \( p\text{-suffix} \) tree for \( P \) through an adaptation of the corresponding algorithm for strings \([35]\). Its space complexity is \( O(m) \) and its time complexity is \( O(n) \) for fixed alphabets; for variable alphabets, its time complexity is \( O(n(|\Sigma_P|+\log(|\Sigma_C|+|\Sigma_P|))) \). Nevertheless, it could also be improved to \( O(n\log(|\Sigma_C|+|\Sigma_P|)) \) by using some auxiliary data structures for computing lowest common ancestors \([17,73]\).

On the other hand, an algorithm, called \textsc{pdup}, for finding the maximal \( p\text{-matches} \) over a threshold length of a text \( T = T_{1...n} \) was devised \([17]\). \textsc{pdup} is similar to \textsc{dup}, but constructs a \( p\text{-suffix} \) tree of the text instead of a suffix tree. This algorithm generalizes to \( p\text{-strings} \) the algorithm for finding maximal \( p\text{-matches} \) over a threshold length in a string \([13]\). In this generalization, it is necessary to augment the \( p\text{-suffix} \) tree with lists that store data that makes possible to determine whether there is left–extensibility in the \( p\text{-matching} \) substrings. The time complexity of \textsc{pdup} is \( O(n+occ) \) even for variable alphabets. The efficiency of this algorithm to detect duplicate code was evaluated through an experiment in \([19]\).

In order to improve the memory usage and access locality provided by \( p\text{-suffix} \) trees, \textit{parameterized suffix arrays}, or \( p\text{-suffix} \) arrays, were defined \([41]\). Specifically, \( p\text{-suffix} \) arrays are defined with respect to \( p\text{-suffix} \) trees in an analogous manner as traditional suffix arrays are defined with respect to suffix trees. It is well-known
that the combination of suffix arrays and the \textit{longest common prefix} (LCP) yields an efficient solution for traditional string matching \cite{64,46,76,1}. This also applies for p-strings \cite{26}. Then, most of the operations on a p-suffix tree can be simulated with the use of the corresponding p-suffix array and an array that contains the lengths of the longest common prefixes of the p-suffixes. The latter, called \textit{parameterized longest common prefix array}, is denoted as p–LCP. Thus, parameterized pattern matching can be solved using p-suffix arrays and p–LCP by means of a binary search in \(O(m + \log n + \text{occ})\) \cite{41}.

The construction of a p-suffix array can be achieved by traversing the corresponding p-suffix tree. Deguchi et.al. were the first to directly construct p-suffix arrays, i.e. without constructing the p-suffix tree \cite{41}. Specifically, they provided a linear algorithm to construct p-suffix arrays and p–LCP for binary alphabets. Then, I et.al. proposed an algorithm for constructing p-suffix arrays and p–LCP for non-binary alphabets \cite{51}. Moreover, they were the first to consider the p-suffix sorting problem, which consists of sorting all the p-suffixes of a p-string in lexicographic order. It is important to remark that traditional suffix sorting techniques cannot be applied to p-suffixes because of their dynamic nature. In particular, the first two algorithms that addressed p-suffix sorting are based in \textsc{QuickSort} and \textsc{RadixSort}; they take \(O(n^3)\) and \(O(n^2)\), respectively \cite{51}. Later, Beal and Adjeroh devised a solution that generates and lexicographically sorts fingerprints and arithmetic codes that correspond to the p-suffixes \cite{22,25}. This algorithm has expected linear time and \(o(n^2)\) worst-case time complexity, which improves the complexity given in \cite{51}.

However, the definition of p-suffix arrays took place in recent years. Chronologically, Baker’s solution based on p-suffix trees was followed by other works. For instance, Amir et.al. defined a related model called \textit{mapped matching} which is a special case of parameterized matching where all symbols are in the parameter alphabet \cite{5}. Through this model, an algorithm that extends the KMP algorithm \cite{57} to parameterized matching, and runs in \(O(n \log \min(m, |\Sigma_P|))\) time, was proposed \cite{5}. It was proven that the \(\log \min(m, |\Sigma_P|)\) factor is inherent to any algorithm for parameterized matching in the comparison model and, consequently, that the provided algorithm is optimal. This demonstration was achieved through a reduction from the element distinctness problem to parameterized matching.

This new research may have motivated Baker to look for parameterized matching solutions based on classical exact string matching algorithms \cite{16}. Given that the \textsc{Boyer–Moore} algorithm \cite{33} is one of the most efficient, she attempted to generalize it to p-strings but found out that its worst case performance was poor. Therefore she turned to one of its variants: \textsc{TurboBM} \cite{37}. Her non-trivial generalization of \textsc{TurboBM} to p-strings, called \textsc{PturboBM}, runs in \(O(n \log \min(m, |\Sigma_P|))\) time and \(O(n)\) space; the preprocessing time is \(O(m \log \min(m, |\Sigma_P|))\). Its time complexity is the same as the generalization of KMP complexity so it is optimal \cite{5}. Nevertheless some experiments show that \textsc{PturboBM} works better for long patterns over different alphabet sizes. Anyhow, for variable alphabets, both of these algorithms are notably better than then p-suffix tree based parameterized matching algorithms.

Other important contributions were made by Idury and Schäffer who proposed some variants of the basic problem (see Section 2) and solutions for all of them \cite{54}. For the Parameterized Multiple Pattern Matching Problem, they proposed an algorithm that uses a modified Aho–Corasick automaton and runs in \(O(n \log(|\Sigma_C| + |\Sigma_P|) + \text{occ})\) time. As for the Parameterized Dynamic Dictionary Problem, they devised an automaton algorithm that supports different operations with the following
time complexity: (i) \(O((n+\text{occ})(\log(|\Sigma_C|+|\Sigma_P|)+\log d))\) for searching the p-string patterns of the dictionary in a p-string text \(T = T_1...n\); (ii) \(O(m(\log(|\Sigma_C|+|\Sigma_P|)) + \log^2 d)\) for inserting a new pattern \(P = P_1...m\) into the dictionary; and (iii) \(O(m(\log(|\Sigma_C| + |\Sigma_P|)) + \log d)\) for deleting a pattern \(P = P_1...m\) from the dictionary, where \(d\) is the total size of all the patterns.

The adaptation of the Aho–Corasick algorithm to p-strings proposed by Idury and Schäffer leads to the definition of parameterized border arrays, or p–border arrays, which constitute the parameterized version of traditional border arrays. In particular, this adaptation modifies the goto and fail functions with their respective parameterized versions: \(pgoto\) and \(pfail\). When only a single pattern is considered, the \(pfail\) function can be implemented by a p–border array. A p-border array can be computed in linear time, as presented in [54]. In more recent works, I et.al. proposed three related algorithms for the binary alphabet case [53,52]: (i) a linear time algorithm to verify if an integer array is a valid p–border array; (ii) a linear time algorithm to compute all the p–strings that share a given p–border array; and (iii) an algorithm that computes all the p–border arrays shorter than a given threshold length. The latter is linear in the number of p–border arrays reported.

The same authors proposed an algorithm to verify if a length-\(n\) integer array is a valid p–border array for the case of unbounded alphabets [53,52]. Its time complexity is \(O(n^{1.5})\) and its space complexity is \(O(n)\). This algorithm is more efficient than the previous solution, which takes time proportional to the \(n\)–th Bell number \(\frac{1}{e} \sum_{k=0}^{\infty} \frac{k^n}{k!}\). Furthermore, it is shown that the enumeration of all p–border arrays shorter than a threshold length \(n\) can be performed in \(O(B^* n^{2.5})\) where \(B^*\) denotes the number of length-\(n\) p–border arrays [53].

On the other hand, Fredriksson and Mozgovoy proposed two new solutions for both the single and multiple parameterized matching problems [44]. Both of them make use of Baker’s lemma to compute the \(\text{prev}\) of a text substring through the \(\text{prev}\) of the container p–string [17]. One of them is a bit–parallelism based algorithm called P–\text{SHIFT}–\text{OR}. It is a generalization of the \text{SHIFT}–\text{OR} algorithm [11] to p–strings and
runs in \(O(n \lceil m/w \rceil)\) worst-case time and \(O(n)\) average time. The other solution, called Parameterized Backward Trie Matching (PBTM) \([44]\), is based on the Backward DAWG Matching (BDM) algorithm \([32,37]\). The average time complexity of PBTM is \(O(n \log(m)/m)\). This algorithm could also make use of a suffix array \([64]\) instead of a trie, in which case it is called Parameterized Backward Array Matching (PBAM). These parameterized matching algorithms are the first for which an average time complexity analysis has been made. They have optimal average-case running time as confirmed by experimental results. Other algorithms with sublinear average-case complexity were proposed in \([72,71]\); they are based on the BOYERMOORE algorithm.

The diagram in Figure 1 shows the algorithms for solving the different parameterized matching problems organized by the nature of their approaches.

4 Extensions

Parameterized matching has been studied in many directions. For instance, it has a close relation with palindromes. It was shown that two strings drawn from an alphabet of size at most 3 have the same set of maximal palindromes if and only if they are a p–match \([77]\). On the other hand, an investigation about the periodicity of parameterized strings was done \([10]\). They attempted to generalize to p–strings two of the periodicity lemmas of strings: the Lyndon and Schitzenberger lemma (referred as Weak Version) \([62]\), and the Fine and Wilf lemma \([43]\). They found out that only the Weak Version holds for p–strings only when the two mappings inducing the periodicity commute. These results and some other studies about the repetitions in p–strings showed considerable differences between p–strings and ordinary strings. Nevertheless, binary p–strings behave in a very similar way as ordinary strings with respect to periodicity and repetitions.

Furthermore, parameterized matching was extended to the two dimensional case by considering matrices of symbols instead of p–strings. Two–dimensional parameterized matching consists of finding all the p–matches of a pattern of size \(m \times m\) in a text of size \(n \times n\). An algorithm for the problem that runs in \(O(n^2 + m^{2.5} \text{polylog } m)\) time was proposed \([18]\). Other solutions include a \(O(n^2 \log^2 m)\) deterministic algorithm and a \(O(n^2 \log n)\) randomized algorithm that reports all the p–matches \([4]\). Nevertheless, it may report a mismatch as match with probability of \(1/n^k\), where \(k\) is a given constant.

Other topic that arose as a matter of interest was the calculation of similarity between two p–strings. In particular, Baker defined the parameterized edit distance or p–edit distance of two p–strings as the cost of a minimal edit script, called p–edit script, that transforms one p–string into the other \([18]\). The valid operations are insertions, deletions and parameterized replacements (the replacement of a substring with a p–string that p–matches it). Moreover, Baker proposed an algorithm \([18]\) for calculating the p–edit distance \(D\) of two prev–encoded p–strings, \(X = X_{1...m}\) and \(Y = Y_{1...m}\), by generalizing Myers’s algorithm for finding the LCS of two strings \([70]\). The algorithm runs in \(O(D(n+m))\) time and \(O(n+m)\) space. Furthermore, a divide-and-conquer based algorithm for reporting the minimal p–edit script was proposed \([18]\). It also runs in \(O(D(n+m))\) and \(O(n+m)\) space.

There have been some works about approximate parameterized problem under hamming distance. In particular, the \(\pi\)–match between two p–strings \(X = X_{1...m}\) and \(Y = Y_{1...m}\) was defined as the number of matches between \(\pi(Y_i)\) and \(X_i\), for \(1 \leq i \leq m\) \([8]\). For two equal–length p–strings, the approximate parameterized matching
problem, also called parameterized matching with mismatches, consists of finding a \( \pi \) of maximal \( \pi \)-match. Given a \( p \)-string pattern \( P = P_{1...m} \) and a \( p \)-string text \( T = T_{1...n} \), the approximate parameterized searching problem under hamming distance consists of computing the approximate parameterized matching between \( P \) and every length-\( m \) \( p \)-substring of \( T \). It is not necessary to choose the same \( \pi \) for every text window. Furthermore, a linear algorithm to solve this problem, for the case where both \( P \) and \( T \) are run–length encoded and one of them is a binary \( p \)-string, was devised [8].

Further studies about parameterized matching and hamming distance have been developed [19,18]. Specifically, a related problem, called parameterized matching with a threshold of \( k \) mismatches, was proposed. Its goal is finding all the \( p \)-matches of a pattern \( P = P_{1...m} \) in a text \( T = T_{1...n} \) with at most \( k \) mismatches. Furthermore, for two \( p \)-strings \( X = X_{1...m} \) and \( Y = Y_{1...m} \), they proposed a \( O(m + k^{1.5}) \) time algorithm and a \( O(m^{1.5}) \) time algorithm for the cases when \( k \) is and is not considered, respectively. These solutions are based on maximum matching algorithms; furthermore, it was demonstrated that the maximum matching problem is reducible to the approximate parameterized matching problem. For a \( p \)-string pattern \( P = P_{1...m} \), a \( p \)-string text \( T = T_{1...n} \) and a given \( k \), a \( O(nk^{1.5} + mk \log m) \) time algorithm for the parameterized matching with \( k \) mismatches problem was also proposed. It is shown that this could be extended to the two dimensional case in \( O(n^2mk^{1.5} + m^2k \log m) \) time.

Another approximate version of parameterized matching is based on \( \delta \)- and \( \gamma \)-distances. Two equal-length integer strings are said to \( \delta \gamma \)-match if (i) the difference between their corresponding symbols is at most \( \delta \); and (ii) the sum of such differences is at most \( \gamma \). Note that constants \( \delta \) and \( \gamma \) are bounds for the local and global errors, respectively, on the difference between the corresponding symbols of the strings. Thus, these distances are used to search for all similar but not necessarily identical occurrences of a given pattern [59]. Then, the \( \delta \gamma \)-approximate parameterized matching problem was defined [59]. Specifically, given two equal-length integer strings \( X = X_{1...m} \) and \( Y = Y_{1...m} \), string \( X \) is said to \( \delta \gamma \)-parameterized match string \( Y \) if \( X \) can be transformed into a string \( X' \), via a bijection \( \pi \) (i.e., \( X'_i = \pi(X_i) \) for \( 1 \leq i \leq m \)), such that \( X' \) \( \delta \gamma \)-matches \( Y \). Moreover, a \( O(nm) \) algorithm to report the \( \delta \gamma \)-parameterized matches of a pattern \( P = P_{1...m} \) in a text \( T = T_{1...n} \) was proposed [59]. In particular, this algorithm is based on a reduction to the Maximum Weight Perfect Matching problem in bipartite graphs [66].

The parameterized matching problem under the LCS distance problem has also been considered. The longest common parameterized subsequence (LCPS) for two \( p \)-strings \( X = X_{1...m} \) and \( Y = Y_{1...n} \) was defined as the pair of sequences \( I \) and \( J \) of maximum length, such that \( I \) is a subsequence of the \( p \)-string \( X \), \( J \) is a subsequence of the \( p \)-string \( Y \), and \( I \) and \( J \) are a \( p \)-match [59]. It is important to remark that it is not required that the symbols in \( I \) and \( J \) are consecutive in \( X \) and \( Y \). The LCPS could be useful as a similarity measure between code sections; nevertheless, this problem has been proven to be NP–hard. Then, an approximate algorithm was proposed [59].

The longest previous factor (LPF) on traditional strings [39] has important applications in string compression [79] and for detecting runs [63]. It was extended to \( p \)-strings as the parameterized longest previous factor, or \( p \)-LPF, by Beal and Adjeroh [20,23]. In particular, given a \( p \)-string \( T = T_{1...n} \), the \( p \)-LPF of a \( p \)-suffix in \( T \), starting at position \( i \), is the longest \( p \)-suffix starting at position \( h \) such that \( 1 \leq h < i \).
The p–LPF is useful to detect study duplication and compression in p–strings. An algorithm to compute the p–LPF in expected linear time was also proposed in [26]; it utilizes p–suffix arrays. This solution can be used to calculate the p–LCP, LCP and LPF due to the general definition of parameterized matching.

This generality was further exploited by the same authors in [27]. They proposed a taxonomy of classes for longest previous factor problems that allows them to show the relation between p–LPF and traditional data structures. Specifically, they show that the p–LCP can be used to linearly construct the p–border array and the border array, which are quite relevant for pattern matching. Moreover, the concept of permuted LCP is extended to p–strings. Also, motivated by the variants of the traditional LPF problem [39,40], they defined the counterpart versions for p–strings: parameterized longest not-equal factor (p–LneF), parameterized longest reverse factor (p–LrF) and parameterized longest factor (p-LF). The same framework of the p-LPF solution can be used to compute all these structures by changing the preprocessing and postprocessing phases. Applications of these data structures include clone detection, pattern substitution, LZ decomposition, periodicity study and biological sequence compression and analysis.

Another parameterized paradigm called parameterized pattern queries, that is closely related to the theory developed by Baker, was proposed [42]. They use a set of symbols and a set of variables that correspond to Baker’s constant alphabet and parameter alphabet. They also defined a concept of valuation that could be associated with the mapping bijection and the p–match definition. This paradigm was conceived as an extension of traditional pattern expressions to enhance the querying and clustering operations over sequence databases. Thus, the definition of a set of predicates on the variables (constraints) is also permitted under this new model. Furthermore, a KMP–based algorithm for this problem is also proposed. Experimental results showed that it notably decreases the query evaluation time compared to a naive approach.

One of the most important extensions of parameterized matching is structural matching (or s–matching for short). Shibuya defined it as parameterized matching but taking into account an injective complementary relation among a subset of the parameters. This relation is used to establish an additional constrain in the matching: if parameter \( x \) is mapped to \( y \), then the complement of \( x \) is also mapped to the complement of \( y \) in the bijection [74]. The motivation for this definition is the application for matching RNA and single-stranded DNA sequences as they contain complementary bases: adenine with uracil or thymine, and cytosine with guanine. Then, two sequences that s–match are likely to have similar structure and, therefore, similar functions [20]. The solution proposed to solve this problem involves the utilization of a structural suffix tree, also called s–suffix tree, which is a generalization of Baker’s p–suffix trees. An on-line algorithm to construct the s–suffix tree in \( O(n(\log |\Sigma_C| + \log |\Sigma_P|)) \) is also presented in [74]. This is the first on-line algorithm that constructs p–suffix trees. It performs in linear time for RNA and DNA sequences. Moreover, it is important to remark that this was a novel approach to the problem of comparing RNA and DNA sequences; other solutions include [50,12,3,45,55,78].

Given that the practical space requirement for s–suffix trees is high, Beal and Adjeroj recently defined the structural suffix array, or s–suffix array for short, and the structural longest common prefix array, denoted as s–LCP, to solve the s–matching problem [21,31]. They exploit the flexibility of these data structures to address diverse variants of the RNA matching problem with slight modifications in the solution.
same authors also proposed another data structure to solve the s–matching problem: the *structural border array* (*s–border array*) [30,24]. A linear time algorithm to construct the s–border array is also presented in [30]; it is based on special properties of the s–border data structure. Furthermore, it is shown how to modify the alphabets so that the algorithm constructs the p–border and the traditional border as well. Due to the recent interest on parameterized matching in compressed strings [8,7,28], the authors also show how to tackle parameterized matching on run-length encoded strings. Another data structure with applications in RNA matching is the *forward stem matrix* (FSM) [29]. This structure efficiently represents the length-\(k\) options, for \(k \in K\), within a length-\(n\) RNA sequence; its size is \(O(n|K|)\).

In order to support other applications, parameterized matching was generalized to *function matching* by allowing the mapping function to be of any type, and not just bijections as in parameterized matching [4]. In other words, many symbols of the pattern can be mapped to the same text symbol. A deterministic solution for the function matching problem, that runs in \(O(n|\Sigma_P| \log m)\) time, was devised [4]. Furthermore, they proposed a Monte Carlo algorithm that runs in \(O(n \log m)\) time with failure probability of \(1/n^k\), where \(k\) is a given constant. Function matching was also extended for the two–dimensional case and a randomized algorithm that runs in \(O(kn^2 \log n)\) time was proposed [4]. This algorithm has a \(1/n^k\) probability of reporting a false positive. An approximate version of function matching based on the \(\delta\gamma\)-distances was developed [69]. Given two integer strings, \(X = X_{1...m}\) and \(Y = Y_{1...m}\), and two given constants, \(\delta\) and \(\gamma\), there is a match from \(X\) to \(Y\) if \(X\) can be transformed into a string \(X'\), by means of a function \(f\), such that \(X'\) is \(\delta\gamma\)-equal to \(Y\). A \(O(nm)\) algorithm to find the \(\delta\gamma\)-function matches of a pattern \(P = P_{1...m}\) in a text \(T = T_{1...n}\) was proposed [69].

To support even a much wider range of applications, function matching was extended to the *generalized function matching with don’t cares* problem [6]. In this problem, the image of the mapping function can be any substring in \((\Sigma_C \cup \Sigma_P)^*\) and not just a single symbol as in function matching. Furthermore, an extra symbol \(\phi\), called the *don’t care* character, can be present in the strings. A \(\phi\) in the text matches any pattern symbol; a \(\phi\) in the pattern matches any text substring. This problem represents many pattern searching types but, as a consequence, it is much more complex. A polynomial algorithm for the finite alphabet case was presented; for the case of infinite alphabets, it was demonstrated that the problem is *NP–hard* [6]. This is the first problem, so far, for which there is a polynomial solution for the finite alphabet case and there is not one for the infinite alphabet case.

5 Applications

Besides its applications in software maintenance, parameterized matching is useful in image processing [49,4]. The Human–Computer Interaction Lab at the University of Maryland tackled the problem of searching for an icon in the screen. If the colors are fixed, the problem can be solved with an exact two-dimensional pattern matching algorithm. Nevertheless, sometimes the pattern image appears in other ranges of colors within the text, which makes impossible for exact–matching algorithms to find these occurrences. In this kind of cases it is proper to use two dimensional *parameterized matching* algorithms. However, images often have some errors resulting from distortion and loss of resolution, so such occurrences of a pattern image could not be reported by parameterized matching algorithms either. But occurrences with
these errors can indeed be found by taking either a function matching approach \[4,69\] or an approximate parameterized matching approach under the hamming, p–edit, or \(\delta\gamma\) distance \[18,48,49,59\].

On the other hand, parameterized matching has applications in databases. For instance, in a database that contains URLs of the pages visited by different users, parameterized pattern queries can be used to retrieve useful information for improving the ergonomy of the site and finding the best places for advertisement ads \[42\]. For example, given the symbol \(a\) and the variable \(x\) where both represent URLs, the query of the parameterized pattern expression \(axa\) would retrieve the set of URLs that the users have visited before coming back to the previously visited page represented by \(a\). In a similar fashion, this idea can be used in computational biology to retrieve all the amino acids substrings that follow a determined structure where the presence of determined amino acids at certain positions are a constraint. This is also applicable to databases of any type, where the analysis over the sequential occurrence of elements is a matter of interest.

In recent works, parameterized matching has been utilized as a mechanism to solve the graph isomorphism problem \[68\]. Graph isomorphism is the problem of determining if the topology of two graphs is the same. More formally, \(G_1 = (V_1, E_1)\) and \(G_2 = (V_2, E_2)\) are isomorphic if there exists a bijection \(f : V_1 \rightarrow V_2\) such that \((u, v) \in E_1\), for \(u, v \in V_1\), if and only if \((f(u), f(v)) \in E_2\). Given that parameterized matching is defined for linear structures, the concept of graph linearization was defined to represent the topology of a graph as a walk that traverses all its nodes and edges. Then, two graphs are isomorphic if and only if there exists a walk in one of the graphs that parameterized-matches a linearization of the other graph.

Specifically, the solution for graph isomorphism under this approach has two main steps: (i) representing \(G_1\) by means of a linearization \(p\); and (ii) determining if there exists a walk in \(G_2\) that parameterized-matches the linearization \(p\) \[68\]. For the former, an efficient linearization algorithm that generates short linearizations with an approximation guarantee was proposed; it requires \(O(|E_1| + d|V_1| \lg d)\) time and \(O(|E_1|)\) space, where \(d\) is the maximum node degree. For the latter, a DFS-based algorithm that prunes the search space by using vertex degrees and previous assignments was developed; it requires \(O(|V_2|d^{\lfloor \ell/2 \rfloor})\) time and \(O(|V_1| + |E_1|)\) space, where \(\ell\) is the length of the linearization \(p\). This solution was experimentally evaluated on graphs of different types and sizes. It was compared to the performance of VF2, which is a prominent algorithm for graph isomorphism. Empirical measurements show that graph linearization finds a matching graph faster than VF2 in numerous cases, especially in Miyazaki-constructed graphs which are known to be one of the hardest cases for graph isomorphism algorithms \[67\].

6 Conclusions

Parameterized matching is a string searching variant that allows to find strings with the same structure. Thus, it is useful in any area where patterns are defined in terms of structural correlation across the positions. Its applications in areas like software maintenance, plagiarism detection and image processing have motivated extensive research for more than two decades. In particular, different problems, solutions, extensions and properties have been studied. New insights on parameterized matching in recent research works include: (i) the definition of new data structures to yield more
efficient solutions; (ii) the generalization to s–matching as a mechanism to match RNA sequences; and (iii) its use to solve the graph isomorphism problem.

References