Parameterized Matching: Solutions & Extensions

Juan Mendivelso¹ & Yoan Pinzón²

¹ Fundación Universitaria Konrad Lorenz
² Universidad Nacional de Colombia

Prague Stringology Conference 2015
Outline

- Background
- Motivation for Parameterized Matching
- Basic Problems
- Solutions
- Extensions
- Applications
- Conclusions
Background
String Comparison

- $X[1..m]$ and $Y[1..m]$ match if $X[i] = Y[i]$ for all $i$. 

\[
\begin{align*}
X & \quad a & b & a & c & a & b & a & c & a \\
Y & \quad a & b & a & c & a & b & a & c & a \\
\end{align*}
\]
String Pattern Matching

- Find the matches of a pattern $P[1..m]$ within a string $T[1..n]$.  

$T$: a b a c a b a c a c a b b

$P$: c a b
String Pattern Matching

- Find the matches of a pattern $P[1..m]$ within a string $T[1..n]$.

$T$:

```
abacabacabacab
```

$P$:

```
cab
```
String Pattern Matching

- Find the matches of a pattern $P[1..m]$ within a string $T[1..n]$.
Outline

- Background
- Motivation for Parameterized Matching
- Basic Problems
- Solutions
- Extensions
- Applications
- Conclusions
Motivation for Parameterized Matching

- Software Maintenance Application
- Definition of Parameterized-Match (p-match)
Motivation for Parameterized Matching

- Software Maintenance Application
- Definition of Parameterized-Match (p-match)
Software Maintenance Application

- Programmers introduce duplicate code in large software systems when they introduce new features or fix bugs.
- They copy and slightly modify the code to avoid the introduction of new bugs.
- The code can be seen as a sequence of tokens.
- Duplicate code can have tokens that remain the same and tokens that systematically change.
Finding duplicate code

- Baker developed interest in solving this problem.

```c
void copy_number(int *pmin, int *pmax, int *pfi, int *pfh) {
    *pmin = *pmax = 0;
    copy_number(&pmin, &pmax, pfi-1, pfh-1);
    *pmin = *pmax = 1;
    copy_number(&pmin, &pmax, pfi+1, pfh+1);
}
```

Figure: [Baker, 1992]
Importance of the problem

- Code gets larger, more complex and more difficult to maintain.
- Fixing a new issue in one of the copies does not fix it in the other (unmonitored) copies.
- Experiments show that 22% of code may be duplicate [Baker, 1992].
- Finding such code can help using better programming techniques to eliminate duplication.
Motivation for Parameterized Matching

- Software Maintenance Application
- Definition of Parameterized-Match (p-match)
Then, Baker defined...

- Constant Alphabet ($\Sigma$)
- Parameter Alphabet ($\Pi$)
- Parameterized-strings: defined over ($\Sigma \cup \Pi$)

\[
\Sigma = \{b\} \quad \Pi = \{x,y,z\}
\]

<table>
<thead>
<tr>
<th>X</th>
<th>x</th>
<th>b</th>
<th>y</th>
<th>y</th>
<th>x</th>
<th>b</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>z</td>
<td>b</td>
<td>x</td>
<td>x</td>
<td>z</td>
<td>b</td>
<td>z</td>
</tr>
</tbody>
</table>
Parameterized-match (p-match)

- P-strings $X[1..m]$ and $Y[1..m]$ are a p-match if one can be mapped into the other through a bijection such that the mapping is identity for the symbols in $\Sigma$.

<table>
<thead>
<tr>
<th>$\Sigma$</th>
<th>b</th>
<th>b</th>
<th>b</th>
<th>b</th>
<th>b</th>
<th>b</th>
<th>b</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\Pi$</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>z</td>
</tr>
<tr>
<td></td>
<td>y</td>
<td>y</td>
<td>z</td>
<td>x</td>
<td>z</td>
<td>x</td>
<td>y</td>
</tr>
<tr>
<td></td>
<td>z</td>
<td>z</td>
<td>y</td>
<td>z</td>
<td>x</td>
<td>y</td>
<td>x</td>
</tr>
</tbody>
</table>
Parameterized-match (p-match)

- There are $|\Pi|!$ possible bijections which makes parameterized matching an interesting combinatorial problem.
Similarity in structure

- Two p-strings that p-match...
  - ... have the same number of distinct symbols.
  - ... the occurrences of each distinct symbol take place in corresponding positions.
Outline

- Background
- Motivation for Parameterized Matching
- Basic Problems
- Solutions
- Extensions
- Applications
- Conclusions
Basic Problems

- Maximal p-matches over a Threshold Length
- Parameterized Pattern Matching
- Parameterized Fixed Multiple Pattern Matching
- Parameterized Dynamic Dictionary Matching
Basic Problems

- **Maximal p-matches over a threshold length:**
  - **Input:** \( T, k \)
  - **Output:** pairs \((u,v)\) of maximal parameterized matching substrings such that \(|u| \geq k\).

- **Complexity:** \( O(n+occ) \) [Baker, 1997]
Basic Problems

- Parameterized Fixed Pattern Matching:
  - **Input:** $T[1..n]$, $P[1..m]$
  - **Output:** substrings in $T$ that parameterized-match $P$
  - **Complexity:** $O(n \log \min\ (m, |\Pi|))$ [Amir, 1994]
Basic Problems

- **Parameterized Fixed Multiple Pattern Matching:**
  - **Input:** $T[1..n]$, set of $d$ patterns $P_i$
  - **Output:** substrings in $T$ that parameterized-match any $P_i$
  - **Complexity:** $O(n \log |\Sigma| + \text{occ})$ [Idury, 1996]
Basic Problems

- **Parameterized Dynamic Dictionary Matching:**
  - The same as Parameterized Fixed Pattern Matching, but new patterns can be inserted or removed from the set.
  - Complexity: $O((n+occ)(\log |\Sigma|+\log d))$ [Idury, 1996].

- Literature on parameterized matching includes solutions for all of these problems, as presented in next section.
Outline

- Background
- Motivation for Parameterized Matching
- Basic Problems
- Solutions
- Extensions
- Applications
- Conclusions
Solutions

Baker’s theory

Generalization of Exact Matching Algorithms
Solutions

Baker’s theory

Generalization of Exact Matching Algorithms
Baker’s Theory

DUP
Solution for String Comparison
prev
p-suffix trees
Pattern matching
Maximal p-matches over a threshold length
Baker’s Theory

DUP
Solution for String Comparison
prev
p-suffix trees
Pattern matching
Maximal p-matches over a threshold length
DUP Algorithm

- Proposed by [Baker, 1992].
- To find maximal p-matches over a threshold length.
- It works as follows:
  - Converts the parameters in a single symbol.
  - Looks for exact matches using a suffix tree.
  - Determines which of such matches are p-matches.
- Experimental results show that just few exact matches are p-matches.
Baker’s Theory

DUP
Solution for String Comparison
prev
p-suffix trees
Pattern matching
Maximal p-matches over a threshold length
Solution for String Comparison

- Straightforward solution [Baker, 1997].
- Construct a mapping table of the mapping while simultaneously traversing both strings until a mismatch is found.

$\Sigma = \{b\} \quad \Pi = \{x,y,z\}$

<table>
<thead>
<tr>
<th>$\alpha$</th>
<th>$f(\alpha)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>x b y y x b x</td>
</tr>
<tr>
<td>Y</td>
<td>z b x x z b z</td>
</tr>
</tbody>
</table>
Solution for String Comparison

- Straightforward solution [Baker, 1997].
- Construct a mapping table of the mapping while simultaneously traversing both strings until a mismatch is found.

\[ \Sigma = \{b\} \quad \Pi = \{x, y, z\} \]

<table>
<thead>
<tr>
<th>(\alpha)</th>
<th>(f(\alpha))</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>z</td>
</tr>
</tbody>
</table>

- \(X\) = \{x, b, y, y, x, b, x\}
- \(Y\) = \{z, b, x, x, z, b, z\}
Solution for String Comparison

- Straightforward solution [Baker, 1997].
- Construct a mapping table of the mapping while simultaneously traversing both strings until a mismatch is found.

\[ \Sigma = \{b\} \quad \Pi = \{x, y, z\} \]

\[
\begin{array}{ccccccc}
X & \text{x} & \text{b} & \text{y} & \text{y} & \text{x} & \text{b} & \text{x} \\
Y & \text{z} & \text{b} & \text{x} & \text{x} & \text{z} & \text{b} & \text{z} \\
\end{array}
\]
Solution for String Comparison

- Straightforward solution [Baker, 1997].
- Construct a mapping table of the mapping while simultaneously traversing both strings until a mismatch is found.

\[ \Sigma = \{b\} \quad \Pi = \{x,y,z\} \]

<table>
<thead>
<tr>
<th>( \alpha )</th>
<th>( f(\alpha) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>z</td>
</tr>
<tr>
<td>y</td>
<td>x</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>x</th>
<th>b</th>
<th>y</th>
<th>y</th>
<th>x</th>
<th>b</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>z</td>
<td>b</td>
<td>x</td>
<td>x</td>
<td>z</td>
<td>b</td>
<td>z</td>
</tr>
</tbody>
</table>
Solution for String Comparison

- Straightforward solution [Baker, 1997].
- Construct a mapping table of the mapping while simultaneously traversing both strings until a mismatch is found.

\[ \Sigma = \{b\} \quad \Pi = \{x, y, z\} \]

<table>
<thead>
<tr>
<th>(\alpha)</th>
<th>(f(\alpha))</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>z</td>
</tr>
<tr>
<td>y</td>
<td>x</td>
</tr>
</tbody>
</table>

\(X\)  
\begin{array}{cccccc}
  x & b & y & y & x & b & x \\
\end{array}

\(Y\)  
\begin{array}{cccccc}
  z & b & x & x & z & b & z \\
\end{array}
Solution for String Comparison

- Straightforward solution [Baker, 1997].
- Construct a mapping table of the mapping while simultaneously traversing both strings until a mismatch is found.

\[ \Sigma = \{b\} \quad \Pi = \{x, y, z\} \]

<table>
<thead>
<tr>
<th>(\alpha)</th>
<th>(f(\alpha))</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>z</td>
</tr>
<tr>
<td>y</td>
<td>x</td>
</tr>
</tbody>
</table>

\[ x \quad b \quad y \quad y \quad x \quad b \quad x \]

\[ z \quad b \quad x \quad x \quad z \quad b \quad z \]
Solution for String Comparison

- Straightforward solution [Baker, 1997].
- Construct a mapping table of the mapping while simultaneously traversing both strings until a mismatch is found.
- Time Complexity: $O(m)$.
- Space Complexity: $O(|\Pi|)$. 
Baker’s Theory

DUP
Solution for String Comparison

prev
p-suffix trees
Pattern matching
Maximal p-matches over a threshold length
Procedure \textit{prev}

- Proposed by [Baker, 1997].
- Array encoding of a p-string \(X[1..m]\) where:
  - Every symbol in \(\Sigma\) remains the same.
  - The first occurrence of each parameter becomes 0.
  - The other occurrences of each parameter becomes the distance to its previous occurrence (parameter pointers).
- It focuses on the string structure.
Procedure \textit{prev}

- Then, $X$ and $Y$ are a p-match iff $\text{prev}(X) = \text{prev}(Y)$:
Complexity of Computing $prev$

- Time complexity: $O(m)$.
- Space complexity: $O(|\Pi|)$.
- String comparison using prev: $O(m)$. 
Computing prev of a substring

- We can compute \( \text{prev}(X[i..j]) \) based on \( \text{prev}(X[1..m]) \).
- Specifically,

\[
\text{prev}(X[i..j])_k = \begin{cases} 
0 & \text{if } \text{prev}(X[1..m])_{i+k-1} > k-1 \\
\text{prev}(X[1..m])_{i+k-1} & \text{otherwise}
\end{cases}
\]

- Essentially, this means that a parameter pointer becomes zero when it points outside of the substring.
Computing prev of a substring

- We can compute $\text{prev}(X[i..j])$ based on $\text{prev}(X[1..m])$.

```
prev(X) = [0, b, 0, 1, 4, b, 2]
X   = [x, b, y, y, x, b, x]
X'  = [b, x, x, x, z, b]
prev(X') = [1, 2, 3, 4, 5]
```
Computing prev of a substring

- We can compute $\text{prev}(X[i..j])$ based on $\text{prev}(X[1..m])$. 

\[ \begin{align*}
\text{prev}(X) & \quad \text{prev}(X') \\
0 & \quad b \\
0 & \quad 0 \\
1 & \quad 1 \\
4 & \quad 4 \\
b & \quad b \\
2 & \quad 2 \\
\end{align*} \]
Computing prev of a substring

- We can compute $\text{prev}(X[i..j])$ based on $\text{prev}(X[1..m])$.
Computing prev of a substring

- We can compute $\text{prev}(X[i..j])$ based on $\text{prev}(X[1..m])$. 

<table>
<thead>
<tr>
<th>$\text{prev}(X)$</th>
<th>0</th>
<th>b</th>
<th>0</th>
<th>1</th>
<th>4</th>
<th>b</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>x</td>
<td>b</td>
<td>y</td>
<td>y</td>
<td>x</td>
<td>b</td>
<td>x</td>
</tr>
<tr>
<td>$X'$</td>
<td>b</td>
<td>x</td>
<td>x</td>
<td>x</td>
<td>z</td>
<td>b</td>
<td></td>
</tr>
<tr>
<td>$\text{prev}(X')$</td>
<td>b</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Computing prev of a substring

- We can compute $\text{prev}(X[i..j])$ based on $\text{prev}(X[1..m])$. 

<table>
<thead>
<tr>
<th>prev$(X)$</th>
<th>0</th>
<th>b</th>
<th>0</th>
<th>1</th>
<th>4</th>
<th>b</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X$</td>
<td>x</td>
<td>b</td>
<td>y</td>
<td>y</td>
<td>x</td>
<td>b</td>
<td>x</td>
</tr>
<tr>
<td>$X'$</td>
<td>b</td>
<td>x</td>
<td>x</td>
<td>z</td>
<td>b</td>
<td></td>
<td></td>
</tr>
<tr>
<td>prev$(X')$</td>
<td>b</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Computing prev of a substring

- We can compute $\text{prev}(X[i..j])$ based on $\text{prev}(X[1..m])$. 

\[
\begin{array}{ccccccc}
\text{prev}(X) & 0 & b & 0 & 1 & 4 & b & 2 \\
X & x & b & y & y & x & b & x \\
X' & b & x & x & z & b \\
\text{prev}(X') & b & 0 & 1 & 0 & b \\
\end{array}
\]
Baker’s Theory

DUP
Solution for String Comparison
prev
p-suffix trees
Pattern matching
Maximal p-matches over a threshold length
Parameterized-suffix (p-suffix)

- **P-suffixes** were also introduced by [Baker, 1997].
- $i$-th p-suffix of $X[1..m]$: $\text{prev}(X[i..m])$.
- Parameterized-suffix tree (**p-suffix tree**): compacted trie that stores all the p-suffixes of a p-string.
- Used as an aid to solve the parameterized pattern matching problem.
p-suffixes

- $\Sigma = \{b\}$, $\Pi = \{x, y\}$
- $T = xbyyxbx$
- $\text{prev}(T) = ob014b2$

<table>
<thead>
<tr>
<th>i</th>
<th>p-substring</th>
<th>p-suffix</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$xbyyxbx$</td>
<td>$ob014b2$</td>
</tr>
<tr>
<td>2</td>
<td>$byyxbx$</td>
<td>$b010b2$</td>
</tr>
<tr>
<td>3</td>
<td>$yyxbx$</td>
<td>$010b2$</td>
</tr>
<tr>
<td>4</td>
<td>$yxbx$</td>
<td>$00b2$</td>
</tr>
<tr>
<td>5</td>
<td>$bx$</td>
<td>$ob2$</td>
</tr>
<tr>
<td>6</td>
<td>$bx$</td>
<td>$bo$</td>
</tr>
<tr>
<td>7</td>
<td>$x$</td>
<td>$0$</td>
</tr>
</tbody>
</table>
p-suffix Tree

- $\Sigma = \{b\}$, $\Pi = \{x,y\}$
- $T = xbyyxbx$

Figure: [Baker, 1997]
# p-suffix Tree Construction

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Baker, 1997] : Lazy</td>
<td>$O(n</td>
</tr>
<tr>
<td>[Baker, 1993]: Eager</td>
<td>$O(n(</td>
</tr>
<tr>
<td>[Kosaraju, 1995]</td>
<td>$O(n \log (</td>
</tr>
<tr>
<td>[Lee, 2011]</td>
<td>Randomized $O(n)$</td>
</tr>
</tbody>
</table>
Baker’s Theory

DUP
Solution for String Comparison
prev
p-suffix trees
Pattern matching
Maximal p-matches over a threshold length
PatternMatching

- **Key idea:** if there is a p-match, $\text{prev}(P)$ exactly matches the first part of a p-suffix of $T$.

- **Algorithm:**
  - Construct a p-suffix tree of $T$.
  - Calculate $\text{prev}(P)$.
  - Follow the path established by $\text{prev}(P)$.
  - The leaves under the path indicate the matching positions.

- **Complexity (fixed alphabets):**
  - Time: $O(m+occ)$, Space: $O(n)$
Pattern Matching

- $\Sigma = \{b\}$, $\Pi = \{x, y\}$
- $T = xbyyxbx$
- $P = bxxyb$
- $\text{prev}(P) = bo10b$

Figure: [Baker, 1997]
Pattern Matching

- $\Sigma=\{b\}$, $\Pi=\{x,y\}$
- $T=xbyyxbx$
- $P=bxxyb$
- $\text{prev}(P) = b010b$

Figure: [Baker, 1997]
Baker’s Theory

DUP
Solution for String Comparison
prev
p-suffix trees
Pattern matching
Maximal p-matches over a threshold length
Maximal p-matches

- DUP was generalized to pDUP [Baker, 1997].
- Instead of a suffix tree, it uses a p-suffix tree.
- It augments the p-suffix tree with lists that provide useful information to determine left-extensibility.
- **Complexity:** $O(n+occ)$ even for variable alphabets.
Solutions

Baker’s theory

Generalization of Exact Matching Algorithms
Generalization of Exact Matching Algorithms

p-Suffix Arrays
p-KMP
p-TurboBM
p-AhoCorasick
PBTM
Parameterized Suffix Arrays

- Improve memory usage and access locality.
- Defined with respect to p-suffix trees in an analogous manner as suffix arrays are defined to suffix trees [Deguchi, 2008].
- \textbf{P-suffix arrays} and \textbf{p-LCP} (parameterized longest common prefix) can simulate the operation of p-suffix trees.
- Pattern matching can be solved with a binary search in $O(m+\log n+occ)$. 
Construction of p-suffix Arrays

- Algorithms to construct a p-suffix array without constructing its corresponding p-suffix tree.
  - [Deguchi, 2008] for binary alphabets.
p-suffix Sorting

- Problem of lexicographically sorting the p-suffixes of a p-string.
- The dynamic nature of p-strings becomes a challenge.
- p-suffix sorting has been considered:
  - [I, 2009]:
    - $O(n^3)$ based on QuickSort
    - $O(n^2)$ based on Raddix Sort.
  - [Beal, 2012]: uses fingerprints and arithmetic codes. Worst case: $o(n^2)$; expected time: $O(n)$. 
Other Insights on the Problem

- [Amir, 1994] defined an associated paradigm: **mapped matching** (where $\Sigma$ is empty).
- Notice that when $\Pi$ is empty, parameterized matching is equivalent to exact pattern matching.
- Based on a reduction to the element distinctness problem, they proved that $\log \min(m, |\Pi|)$ is inherent to any parameterized matching algorithm.
Parameterized KMP

- [Amir, 1994] also proposed a parameterized version of the KMP algorithm: p-KMP.
- It runs in $O(n \log \min(m, |\Pi|))$.
- It is the first optimal algorithm.
Parameterized Boyer-Moore

- Later, [Baker, 1995] explored the generalization of Boyer-Moore algorithm to parameterized matching, but its worst-case performance was poor.
- Then, she generalized one of its variants: TurboBM.
- The resulting algorithm takes
  - Searching phase: $O(n \log \min(m, |\Pi|))$ so it’s optimal.
  - Preprocessing phase: $O(m \log \min(m, |\Pi|))$
  - Space complexity: $O(n)$
  - Better for long patterns.
Parameterized Aho-Corasick

- [Idury, 1996] proposed multiple parameterized matching.
- They proposed an adaptation of the Aho Corasick algorithm that runs in $O(n \log (|\Sigma|+|\Pi|)+occ)$.
- A dynamic dictionary of patterns was also considered:
  - Searching for patterns: $O((n+occ)(\log (|\Sigma|+|\Pi|)+\log d))$
  - Inserting a pattern: $O(m \log (|\Sigma|+|\Pi|)+\log^2 d))$
  - Deleting a pattern: $O(m \log (|\Sigma|+|\Pi|)+\log d))$
Parameterized border arrays

- Parameterized version of traditional border arrays.
- The p-AhoCorasick algorithm led to their definition:
  - \texttt{pgoto}, \texttt{pfail} are the parameterized counterparts of \texttt{goto} and \texttt{fail} in traditional AhoCorasick.
  - When there is a single pattern, \texttt{pfail} can be implemented as a p-border array.
  - It can be computed in linear time [Idury, 1996].
Parameterized border arrays

- For binary alphabets [I, 2009a] proposed algorithms to:
  - Validate if an integer array is a valid p-border array. \textbf{Complexity:} $O(n)$.
  - Compute all the p-strings that share the same p-border array. \textbf{Complexity:} $O(n)$.
  - Compute all the border arrays shorter than a threshold length. \textbf{Complexity:} linear in the output reported.
Parameterized border arrays

- For unbounded alphabets, [I, 2009a] proposed an algorithm to verify if an integer array is valid p-border array. **Time:** $O(n^{1.5})$. **Space:** $O(n)$.
- Furthermore, they showed that the enumeration of all p-border arrays shorter than a threshold length can be done in $O(B^n n^{2.5})$. 
p-Shift-OR

- [Fredriksson, 2006] makes use of Baker’s theory to propose to algorithms: p-ShiftOR and PBTM.
- p-ShiftOR is a generalization of ShiftOR to p-strings.
- Time complexity:
  - Worst case: $O(n\lceil m/w \rceil)$
  - Average case: $O(n)$. 
Parameterized Backward Trie Matching (PBTM)

- It is based on the Backward DAWG Matching (BDM) Algorithm and makes use of tries.
- Its average time complexity is $O(n \log (m)/m)$.
- A variation that uses arrays instead of tries was also considered by [Fredriksson, 2006]; such variation is called PBAM.
Average Case Analysis

- P-ShiftOR and PBTM were the first parameterized matching algorithms for which the average-case analysis was made.
- An algorithm that has sublinear average-case expected time was proposed by [Salmela, 2006]. It is based on Boyer-Moore.
Solutions

Parameterized Matching Algorithms are based on Automata, which can be further divided into P-AhoCorasick, P-DDM, and P-KMP. Tries and Bit Parallelism lead to Sequential Traversing, which can be used for P-TurboBM.

The background color of each algorithm indicates the problem it solves:

- Maximal p-matches over a threshold length
- Parameterized Fixed Pattern Matching
- Parameterized Fixed and Multiple Pattern Matching
- Parameterized Dynamic Dictionary Matching
Outline

- Background
- Motivation for Parameterized Matching
- Basic Problems
- Solutions
- Extensions
- Applications
- Conclusions
Extensions

Some properties
Two-dimensional parameterized matching
Approximate Approaches
Parameterized Longest Previous Factor
Structural Matching
Function Matching
Extensions

Some properties
Two-dimensional parameterized matching
Approximate Approaches
Parameterized Longest Previous Factor
Structural Matching
Function Matching
Some properties

- **Relation with palindromes:** Two strings drawn from an alphabet of size 3 have the same set of maximal palindromes iff they are a p-match [I, 2010].

- **Periodicity and repetitions:** [Apostolico, 2008]
  - For binary alphabets, p-strings and strings behave in a similar manner.
  - For non-binary alphabets, there are significant differences between p-strings and strings.
Extensions

Some properties
Two-dimensional parameterized matching
Approximate Approaches
Parameterized Longest Previous Factor
Structural Matching
Function Matching
Two dimensional p-matching

Find all the 2-dimensional p-matches:

\[ P_{m \times m} \quad g_i \quad T'_{m \times m} \quad g_i \quad T_{n \times n} \]
Two-dimensional p-matching

- Deterministic solutions:
  - $O(n^2 + m^{2.5} \text{ polylog } m)$ by [Hazay, 2004].
  - $O(n^2 \log^2 m)$ by [Amir, 2003].

- Randomized Algorithm
  - $O(n^2 \log n)$ by [Amir, 2003] with error probability of $1/n^k$ (where $k$ is a constant).
Extensions

Some properties
Two-dimensional parameterized matching

Approximate Approaches
Parameterized Longest Previous Factor
Structural Matching
Function Matching
Approximate Approaches

P-Edit distance
P-matching under the hamming distance
$\delta\gamma$-Parameterized Matching
Longest Common Parameterized Subsequence
Parameterized edit distance

- **P-edit distance**: cost of a minimal script that transforms one p-strings into the other.

- Valid operations:
  - Insertions
  - Deletions
  - Parameterized replacements (replacement of a p-string with a p-string that matches it).

- $O(D(n+m))$-time algorithms proposed by [Baker, 1999].
  - Calculating the p-edit distance $D$.
  - Reporting the minimal p-edit script.
P-matching under the hamming distance

- For a given mapping $g$ between to equal-length $p$-strings $X$ and $Y$, the **$g$-match** is the number of matches between $X[i]$ and $g(Y[i])$, for all $i$. 
P-matching under the hamming distance

- **Approximate Parameterized Matching:** Find the maximal $g$-match between two equal-length p-strings.
- **Parameterized searching under the hamming distance:** For every length-$m$ text window in the text find the maximal $g$-match.
- An algorithm for a run-length encoded pattern and text, where one of them is a binary p-string, was proposed by [Apostolico, 2007].
P-matching under the hamming distance

- **Parameterized matching with $k$ mismatches:** Find all the text windows in the text that $p$-match the pattern with at most $k$ mismatches.
- **Algorithms proposed by [Hazay, 2007]:**

<table>
<thead>
<tr>
<th>Case</th>
<th>Time Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>String comparison</td>
<td>$O(m+k^{1.5})$</td>
</tr>
<tr>
<td>Pattern matching</td>
<td>$O(nk^{1.5}+mk \log m)$</td>
</tr>
<tr>
<td>2-Dimensional</td>
<td>$O(n^2mk^{1.5}+m^2k \log m)$</td>
</tr>
</tbody>
</table>
δγ-Parameterized Matching

- In traditional integer strings, $X[1..m]$ and $Y[1..m]$...
  - ... $\delta$-match iff $\max_i |X[i]-Y[i]| \leq \delta$.
  - ... $\gamma$-match iff $\sum_i |X[i]-Y[i]| \leq \gamma$.
- For example, the following strings $\delta\gamma$-match for $\delta=2$ and $\gamma=7$:

```
<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>1</th>
<th>3</th>
<th>6</th>
<th>3</th>
<th>3</th>
<th>4</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>1</td>
<td>3</td>
<td>1</td>
<td>3</td>
<td>6</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>X</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
</tbody>
</table>
```
δγ-Parameterized Matching

- Integer p-strings $X[1..m]$ and $Y[1..m]$ δγ–parameterized match iff $X$ can be transformed into $X'$ via a bijection $g$ such that $X'$ δγ–matches $Y$.

Example:
- $\delta=2$
- $\gamma=5$
δγ-Parameterized Matching

- A $O(nm)$ algorithm to find all the δγ-parameterized matches of a pattern in a text was proposed by [Mendivelso, 2010].
- It is based on a reduction to the Maximum Weight Perfect Matching problem in bipartite graphs.
Longest Common Parameterized Subsequence (LCPS)

- Given $X[1..n]$ and $Y[1..m]$, find a subsequence $I$ of $X$ and a subsequence $J$ of $Y$ of maximum length such that $I$ and $J$ are a p-match.
- It’s an NP-Hard problem.
- An approximate solution was proposed by [Keller, 2009].
Extensions

Some properties
Two-dimensional parameterized matching
Approximate Approaches
Parameterized Longest Previous Factor
Structural Matching
Function Matching
Parameterized Longest Previous Factor (p-LPF)

- For a p-string of, the p-LPF is calculated for each p-suffix starting at position $i$ as the longest factor between such p-suffix and a p-suffix starting before.
- Used to study duplication and compression in p-strings.
- [Beal, 2012] proposed an expected linear time algorithm to compute the p-LPF, LPF, p-LCP, LCP.
Variants of the p-LPF

- [Beal, 2012a] proposed a taxonomy of classes of LPF problems that show the relation between p-LPF and traditional data structures.
- It is shown that p-LCP can be used to linearly construct the p-border array and the border array.
- The concept of permuted LCP is extended to p-strings.
Variants of the p-LPF

- [Beal, 2012a] defined:
  - Parameterized Longest not-equal Factor (p-LneF)
  - Parameterized Longest reverse Factor (p-LrF)
  - Parameterized Longest Factor (p-LF)

- These structures can be calculated with the same framework of p-LPF by changing preprocessing and postprocessing.

- They have applications in clone detection, periodicity study and biological sequence compression.
Extensions

Some properties

Two-dimensional parameterized matching

Approximate Approaches

Parameterized Longest Previous Factor

**Structural Matching**

Function Matching
Structural Matching (s-matching)

- [Shibuya, 2004] defined it as parameterized matching but taking into account an injective complementary relation among a subset of the parameters.
- Additional constrain in the matching: if parameter $x$ is mapped to parameter $y$, then the complement of $x$ must be mapped to the complement of $y$.
- This is motivated by the application of RNA matching:
  - Adenine – Uracil
  - Cytosine – Guanine
Structural Suffix Trees

- Then, two s-strings that s-match have similar structures and, hence, similar functions.
- [Shibuya, 2004] proposed a solution based on structural suffix trees.
- He also proposed an $O(n(\log|\Sigma|+\log|\Pi|))$ online algorithm to construct a s-suffix tree.
- It is linear for RNA/DNA sequences.
Structural Suffix Arrays

- For better space utilization, [Beal, 2013 and 2015] defined:
  - S-suffix array
  - S-LCP
  - S-border array
Extensions

Some properties
Two-dimensional parameterized matching
Approximate Approaches
Parameterized Longest Previous Factor
Structural Matching
Function Matching
Function Matching

- Two equal-length strings function-match if one can be transformed into the other by means of a function.
- In pattern matching, many symbols in the pattern can be mapped to the same symbol in the text window.
- Solutions by [Amir, 2003]:
  - Deterministic Solution: $O(n |\Pi| \log m)$
  - Monte Carlo Algorithm: $O(n \log m)$ with $1/n^k$ failure probability.
Function Matching Extensions

- **2-dimensional Funcion Matching**: A $O(kn^2 \log n)$ randomized algorithm was proposed [Amir, 2003].

- **$\delta\gamma$-Function Matching**:
  - $X[1..m]$ and $Y[1..m]$ strings match if $X$ can be transformed into $X'$ by means of a function $g$ such that $X'$ $\delta\gamma$-matches $Y$.
  - A $O(nm)$ algorithm was proposed by [Mendivelso, 2012].
Generalized Function Matching with Don’t Cares

- The image of the mapping function any substring in $(\Sigma \cup \Pi)^*$. 
- The **don’t care** symbol $\phi$ can be present in strings. It matches:
  - Any substring in the text if it is in the pattern.
  - Any symbol in the pattern if it is in the text.
Generalized Function Matching with Don’t Cares

- A polynomial-time algorithm for finite alphabets was devised [Amir, 2007].
- It was shown that for infinite alphabets, the problem is NP-Hard.
- It is the first problem for which there is a polynomial solutions for finite alphabets but not for infinite alphabets.
Outline

- Background
- Motivation for Parameterized Matching
- Basic Problems
- Solutions
- Extensions
- Applications
- Conclusions
Applications

Image Processing
Databases
Graph Isomorphism Solution
Applications

Image Processing
Databases
Graph Isomorphism Solution
Image Processing

- The problem of searching an icon in the screen [Hazay, 2007].
- It can be solved with:
  - Exact matching
  - Parameterized matching
  - Approximate parameterized matching (hamming, p-edit, δγ distance)
  - Function matching
Applications

Image Processing
Databases
Graph Isomorphism Solution
Databases

- In a database of URL’s, parameterized queries can be used to improve the ergonomics of the site and finding the best places for advertisement ads.
- In computational biology, it can be used to find amino acid strings that follow a determined structure.
Applications

Image Processing
Databases
Graph Isomorphism Solution
Graph Isomorphism

- Is there a bijection $f$ that maps the nodes/edges of $G_1$ to the nodes/edges in $G_2$ so that the adjacency relation is preserved?
Graph Isomorphism

- Is there a bijection $f$ that maps the nodes/edges of $G_1$ to the nodes/edges in $G_2$ so that the adjacency relation is preserved?
Graph Isomorphism

Is there a bijection $f$ that maps the nodes/edges of $G_1$ to the nodes/edges in $G_2$ so that the adjacency relation is preserved?
Graph Linearization

- It represents the structure of a graph in a linear manner.

- Specifically, our linearization is a walk on the graph that contains all its nodes and edges at least once.

- Then, we evaluate graph isomorphism by comparing walks rather than graphs.
How do we linearize a graph?
How do we linearize a graph?
How do we linearize a graph?
How do we linearize a graph?
How do we linearize a graph?
How do we linearize a graph?
The parameters

$G_1$

A

B

e_1

e_2

e_3

C

D

A

B

C

D


B

A

B

C

D


How to use our linearizations to match graphs?

- $G_1$ and $G_2$ are isomorphic if there is a linearization of $G_2$ that parameterized-matches the linearization of $G_1$.
How to use our linearizations to match graphs?

- $G_1$ and $G_2$ are isomorphic if there is a linearization of $G_2$ that parameterized-matches the linearization of $G_1$. 
How to use our linearizations to match graphs?

- But what if we had calculated the following \( q \)?
- We need to check all the possible linearizations \( q \).
How to use our linearizations to match graphs?

- But there may be $\Omega(\max(n!,m!))$ linearizations of a graph.

$q$ can be any of:
Proposed Solution

- [Mendivelso, 2013] proposed a solution to determine if $G_1=(V_1, E_1)$ and $G_2=(V_2, E_2)$ are isomorphic. It consists of two steps:

  2. Determining whether there exists a walk $q$ in $G_2$ that parameterized-matches $p$. 
Proposed Solution

- The total time complexity is:
  \[ O(dm \log d + nd^{e/2}) = O(nd^{e/2}) \]
- Experimental results show that this solution is efficient especially for Miyazaki graphs which constitute a hard case for graph isomorphism algorithms [Mendivelso, 2015].
Outline

- Background
- Motivation for Parameterized Matching
- Basic Problems
- Solutions
- Applications
- Conclusions
Conclusions
Conclusions

- Parameterized matching allows to find strings with similar structure.
- It has important applications in different areas such as software maintenance, image processing, computational biology, to name some.
- There has been extensive research for the last decades.
- New insights include the definition of new data structures, the extension to RNA matching and its application to solve graph isomorphism.
Bibliography (I)

Bibliography (II)

Bibliography (III)

Thank you!

Any questions?