An Efficient Algorithm for Approximate Pattern Matching with Swaps

Matteo Campanelli\textsuperscript{2}  Domenico Cantone\textsuperscript{1}  Simone Faro\textsuperscript{1}  Emanuele Giaquinta\textsuperscript{1}

Department of Mathematics and Computer Science, University of Catania, Italy

Scuola Superiore di Catania, University of Catania, Italy
Pattern Matching with Swaps

A *swap permutation* for a string $P$ of length $m$ is a permutation $\pi : \{0, \ldots, m-1\} \rightarrow \{0, \ldots, m-1\}$ such that:

(a) if $\pi(i) = j$ then $\pi(j) = i$ (characters at positions $i$ and $j$ are swapped);
(b) for all $i$, $\pi(i) \in \{i-1, i, i+1\}$ (only adjacent characters are swapped);
(c) if $\pi(i) \neq i$ then $P[\pi(i)] \neq P[i]$ (identical characters can not be swapped).
A *swap permutation* for a string $P$ of length $m$ is a permutation
\( \pi : \{0, ..., m - 1\} \rightarrow \{0, ..., m - 1\} \) such that:

(a) if $\pi(i) = j$ then $\pi(j) = i$ (characters at positions $i$ and $j$ are swapped);
(b) for all $i$, $\pi(i) \in \{i - 1, i, i + 1\}$ (only adjacent characters are swapped);
(c) if $\pi(i) \neq i$ then $P[\pi(i)] \neq P[i]$ (identical characters can not be swapped).

$P$ has a swapped occurrence in $T$ at location $j$ with $k$ swaps - $P \propto_k T_j$ - if a swap permutation $\pi$ of $P$ exists such that $\pi(P)$ matches $T$ at location $j$ and $k = |\{i : P[i] \neq P[\pi(i)]\}|/2$.
Pattern Matching with Swaps

\[
\begin{align*}
\text{fate} & \quad \pi(1) = 2, \pi(2) = 1, \pi(3) = 3, \pi(4) = 4 \\
\text{afte} & \quad \pi(1) = 2, \pi(2) = 1, \pi(3) = 3, \pi(4) = 4 \\
\text{afet} & \quad \pi(1) = 2, \pi(2) = 1, \pi(3) = 4, \pi(4) = 3 \\
\text{faet} & \quad \pi(1) = 1, \pi(2) = 2, \pi(3) = 4, \pi(4) = 3 \\
\text{ftae} & \quad \pi(1) = 1, \pi(2) = 3, \pi(3) = 2, \pi(4) = 4
\end{align*}
\]
Approximate Pattern Matching with Swaps problem:

- Alphabet $\Sigma$
- Pattern $P$
- Text $T$

Find all the pairs $(j, k)$ such that $P$ has a swapped occurrence in $T$ at location $j$ with $k$ swaps
Previous work

- (Amir & Lewenstein & Porat, 2002): $O(n \log m \log \min(m, |\Sigma|))$
- (Cantone & Faro, 2009):
  - $O(mn)$ dynamic-programming algorithm
  - $O(\lceil (mn \log m)/w \rceil)$ bit-parallel algorithm; linear $O(n)$ if $m(\log(\lfloor m/2 \rfloor + 1) + 1) \leq w$
Approximate-BCS algorithm

- BDM-like algorithm:
  - right-to-left scans in windows of size $m$
  - window update by left-align with the longest prefix matched
- Find the longest prefix of the pattern which has a swapped occurrence in the current window and count the number of swap operations using dynamic-programming
The set $S_j^h$ includes all the values $i$ such that the $h$-substring of $P$ ending at position $i$ has a swapped occurrence ending at position $j$ in $T$. 

$$S_j^h = \{ h - 1 \leq i \leq m - 1 \mid P[i - h + 1 .. i] \propto T_j \}$$
\textbf{Approximate-BCS algorithm}

\[ \mathcal{W}_j^h = \{ h \leq i < m-1 \mid P[i-h+2..i] \propto T_j \text{ and } P[i-h+1] = T[j-h] \} \]

- The set \( \mathcal{W}_j^h \) includes all the values \( i \) such that the \( h - 1 \) substring of \( P \) ending at position \( i \) has a swapped occurrence at position \( j \) in \( T \) and the first part of the swap between characters \( P[i-h] \) and \( P[i-h+1] \) is recognized
The sets $S_j^h$ and $W_j^h$ can be computed using the following recurrences:

- $S_j^{h+1} = \{ h - 1 \leq i \leq m - 1 \mid (i \in S_j^h \text{ and } P[i - h] = T[j - h]) \text{ or } (i \in W_j^h \text{ and } P[i - h] = T[j - h + 1]) \}$
- $W_j^{h+1} = \{ h \leq i \leq m - 1 \mid i \in S_j^h \text{ and } P[i - h] = T[j - h - 1] \}$

Base cases:
- $S_j^0 = \{ i \mid 0 \leq i < m \}$
- $W_j^0 = \{ 0 \leq i < m - 1 \mid P[i + 1] = T[j] \}$
**Approximate-BCS algorithm**

- If $h - 1 \in S^h_j$ there is a swapped occurrence of the prefix of $P$ of length $h$
- The window is shifted by $m - l$, where $l = \max\{h : h - 1 \in S^h_j\}$
Approximate-BCS algorithm

- If $m - 1 \in S_j^m$ $P$ has a swapped occurrence at position $j$ in $T$

- $m - 1 \in S_j^m \iff m - 1 \in (S_j^h \cup W_j^h), 1 \leq h \leq m$

- Swap between characters $P[m - 1 - h]$ and $P[m - 1 - h + 1] \iff m - 1 \in S_j^{h+1} \land m - 1 \in W_j^h \land m - 1 \notin S_j^h$
$P = \text{ooze}, T = \text{ooez}$

$m - 1 \in W_j^1, m - 1 \notin S_j^1$

$m - 1 \notin W_j^2, m - 1 \in S_j^2$

$m - 1 \in W_j^3, m - 1 \in S_j^3$

$m - 1 \in W_j^4, m - 1 \in S_j^4$
Approximate-BCS algorithm

- The number of swaps for a match at position $j$ is given by
  \[ \left| \{ 1 \leq h < m : (m - 1) \in (S_{j+1}^h \setminus S_j^h) \} \right| \]

- The algorithm maintains a single counter per window

- At iteration $h$ the counter is incremented if $m - 1 \in S_{j+1}^h \setminus S_j^h$
Approximate-BPBCS algorithm

- Simulation of Approximate-BCS using bit-parallelism
- $S_j^h$ and $W_j^h$ represented as vector of $m$ bits
  - $S_j^h \rightarrow D_j^h$: the $i - h + 1$-th bit of $D_j^h$ is set to 1 if $i \in S_j^h$
  - $W_j^h \rightarrow C_j^h$: the $i - h + 1$-th bit of $C_j^h$ is set to 1 if $i \in W_j^h$
- Bit mask $M[c]$, $i$-th bit is set to 1 if $P[i] = c$, as in Shift-And
**Approximate-BPBCS algorithm**

(a) $S_{j}^{h+1} \leftarrow \{i : i \in S_{j}^{h} \text{ and } P[i - h] = T[j - h]\}$

(a') $D_{j}^{h+1} \leftarrow (D_{j}^{h} \ll 1) \& M[T[j - h]]$

(b) $S_{j}^{h+1} \leftarrow S_{j}^{h+1} \cup \{i : i \in W_{j}^{h} \text{ and } P[i - h] = T[j - h + 1]\}$

(b') $D_{j}^{h+1} \leftarrow D_{j}^{h+1} \mid ((C_{j}^{h} \ll 1) \& M[T[j - h + 1]])$
**Approximate-BPBCS algorithm**

(c) \( W_{j}^{h+1} \leftarrow \{ i : i \in S_{j}^{h} \text{ and } P[i - h] = T[j - h - 1] \} \)

(c') \( C_{j}^{h+1} \leftarrow (D_{j}^{h} \ll 1) \& M[T[j - h - 1]] \)

(d) \( m - 1 \in S_{j}^{h+1} \setminus S_{j}^{h} \)

(d') \( ((D_{j}^{h+1} \& \sim (D_{j}^{h} \ll 1)) \& (1 \ll h)) \neq 0 \)
**Complexity**

- **APPROXIMATE-BCS**: $O(nm^2)$ worst case time complexity, $O(m)$ space complexity

- **APPROXIMATE-BPBCS**: $O(\lceil nm^2 / w \rceil)$ worst case time complexity, $O(\sigma \lceil m / w \rceil + \log(\lceil m / 2 \rceil + 1))$ space complexity
Experimental results

- Implementation in C, compiled with gcc, run on AMD Turion X2 2GHz
- Randσ problems, protein and genome sequences, natural language text
- Set of 100 patterns of fixed length $m \in \{4, 8, 12, 16, 20, 24, 28, 32\}$, randomly extracted from the text
- Comparison between the following algorithms:
  - Approximate-Cross-Sampling (ACS)
  - BP-Approximate-Cross-Sampling (BPACS)
  - Approximate-BCS (ABCS)
  - Approximate-BPBCS (BPABCS)
  - Iliopoulos-Rahman algorithm with a naive check of the swaps (IR&C)
  - BP-Backward-Cross-Sampling algorithm with a naive check of the swaps (BPBCS&C)
### Experimental results

#### Running times for a Rand8 problem

<table>
<thead>
<tr>
<th>m</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>BPACS</td>
<td>0.832</td>
<td>0.830</td>
<td>0.828</td>
<td>0.831</td>
<td>0.830</td>
<td>0.829</td>
<td>0.827</td>
<td>0.827</td>
</tr>
<tr>
<td>BPABCS</td>
<td>0.413</td>
<td>0.229</td>
<td>0.175</td>
<td>0.145</td>
<td>0.127</td>
<td>0.114</td>
<td>0.104</td>
<td>0.096</td>
</tr>
<tr>
<td>IR&amp;C</td>
<td>0.282</td>
<td>0.279</td>
<td>0.279</td>
<td>0.277</td>
<td>0.280</td>
<td>0.279</td>
<td>0.283</td>
<td>0.285</td>
</tr>
<tr>
<td>BPBCS&amp;C</td>
<td>0.388</td>
<td>0.249</td>
<td>0.193</td>
<td>0.157</td>
<td>0.141</td>
<td>0.121</td>
<td>0.111</td>
<td>0.101</td>
</tr>
</tbody>
</table>

#### Running times for a natural language text ($\sigma = 93$)

<table>
<thead>
<tr>
<th>m</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACS</td>
<td>3.170</td>
<td>2.757</td>
<td>2.748</td>
<td>2.756</td>
<td>2.761</td>
<td>2.745</td>
<td>2.746</td>
<td>2.754</td>
</tr>
<tr>
<td>ABCS</td>
<td>6.175</td>
<td>4.054</td>
<td>3.164</td>
<td>2.705</td>
<td>2.306</td>
<td>2.288</td>
<td>2.042</td>
<td>1.866</td>
</tr>
<tr>
<td>BPACS</td>
<td>0.492</td>
<td>0.497</td>
<td>0.492</td>
<td>0.491</td>
<td>0.492</td>
<td>0.491</td>
<td>0.494</td>
<td>0.493</td>
</tr>
<tr>
<td>BPABCS</td>
<td>0.194</td>
<td>0.114</td>
<td>0.086</td>
<td>0.071</td>
<td>0.062</td>
<td>0.056</td>
<td>0.051</td>
<td>0.049</td>
</tr>
<tr>
<td>IR&amp;C</td>
<td>0.171</td>
<td>0.165</td>
<td>0.164</td>
<td>0.168</td>
<td>0.165</td>
<td>0.165</td>
<td>0.165</td>
<td>0.167</td>
</tr>
<tr>
<td>BPBCS&amp;C</td>
<td>0.164</td>
<td>0.126</td>
<td>0.094</td>
<td>0.076</td>
<td>0.070</td>
<td>0.059</td>
<td>0.056</td>
<td>0.055</td>
</tr>
</tbody>
</table>
# Experimental results

## Running times for a genome sequence ($\sigma = 4$)

<table>
<thead>
<tr>
<th>$m$</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACS</td>
<td>5.629</td>
<td>5.643</td>
<td>5.654</td>
<td>5.636</td>
<td>5.644</td>
<td>5.640</td>
<td>5.647</td>
<td>6.043</td>
</tr>
<tr>
<td>BPACS</td>
<td>0.950</td>
<td>0.914</td>
<td>0.917</td>
<td>0.766</td>
<td>0.874</td>
<td>0.934</td>
<td>0.935</td>
<td>0.843</td>
</tr>
<tr>
<td>BPABCS</td>
<td>0.647</td>
<td><strong>0.318</strong></td>
<td><strong>0.266</strong></td>
<td><strong>0.232</strong></td>
<td><strong>0.195</strong></td>
<td><strong>0.174</strong></td>
<td><strong>0.160</strong></td>
<td>0.147</td>
</tr>
<tr>
<td>IR&amp;C</td>
<td><strong>0.262</strong></td>
<td>0.287</td>
<td>0.314</td>
<td>0.311</td>
<td>0.311</td>
<td>0.311</td>
<td>0.310</td>
<td>0.311</td>
</tr>
<tr>
<td>BPBCS&amp;C</td>
<td>0.678</td>
<td>0.367</td>
<td>0.290</td>
<td>0.233</td>
<td>0.204</td>
<td>0.176</td>
<td><strong>0.160</strong></td>
<td><strong>0.146</strong></td>
</tr>
</tbody>
</table>

## Running times for a protein sequence ($\sigma = 22$)

<table>
<thead>
<tr>
<th>$m$</th>
<th>4</th>
<th>8</th>
<th>12</th>
<th>16</th>
<th>20</th>
<th>24</th>
<th>28</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABCS</td>
<td>7.045</td>
<td>4.557</td>
<td>3.734</td>
<td>3.162</td>
<td>2.806</td>
<td>2.661</td>
<td>2.600</td>
<td>2.351</td>
</tr>
<tr>
<td>BPACS</td>
<td>0.565</td>
<td>0.581</td>
<td>0.561</td>
<td>0.563</td>
<td>0.584</td>
<td>0.580</td>
<td>0.534</td>
<td>0.519</td>
</tr>
<tr>
<td>BPABCS</td>
<td>0.249</td>
<td><strong>0.142</strong></td>
<td><strong>0.103</strong></td>
<td><strong>0.084</strong></td>
<td><strong>0.074</strong></td>
<td><strong>0.066</strong></td>
<td><strong>0.061</strong></td>
<td><strong>0.058</strong></td>
</tr>
<tr>
<td>IR&amp;C</td>
<td>0.388</td>
<td>0.390</td>
<td>0.391</td>
<td>0.389</td>
<td>0.391</td>
<td>0.391</td>
<td>0.396</td>
<td>0.389</td>
</tr>
<tr>
<td>BPBCS&amp;C</td>
<td><strong>0.241</strong></td>
<td>0.145</td>
<td>0.107</td>
<td>0.087</td>
<td>0.075</td>
<td>0.068</td>
<td>0.062</td>
<td><strong>0.058</strong></td>
</tr>
</tbody>
</table>
Conclusions

- The **Approximate-BPBCS** algorithm is the fastest for $m \geq 8$
- The **Approximate-BPBCS** algorithm scales better than **BP-Approximate-Cross-Sampling**
  - **BP-Approximate-Cross-Sampling**: $m$ counters, linear if $m(\log(\lfloor m/2 \rfloor + 1) + 1) \leq w$
  - **Approximate-BPBCS**: one counter, linear if $m \leq w$