Efficient Online Abelian Pattern Matching in Strings by Simulating Reactive Multi-Automata

Domenico Cantone and Simone Faro

Department of Mathematics and Computer Science
University of Catania (Italy)

The Prague Stringology Conference 2014
Prague 1-4 August, 2014

The Abelian Pattern Matching Problem

Given a pattern p and a text t, the *abelian pattern matching* problem (also known as *jumbled matching*) consists in finding all substrings of the text t, whose characters have the same multiplicities as in p, so that they could be converted into the input pattern just by permuting their characters.

The Abelian Pattern Matching Problem

Given a pattern p and a text t, the *abelian pattern matching* problem (also known as *jumbled matching*) consists in finding all substrings of the text t, whose characters have the same multiplicities as in p, so that they could be converted into the input pattern just by permuting their characters.

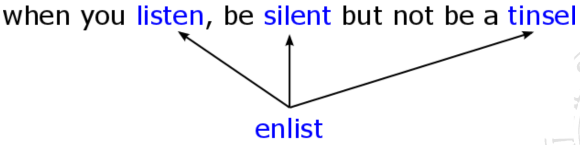
---

when you listen, be silent but not be a tinsel

enlist

The Abelian Pattern Matching Problem

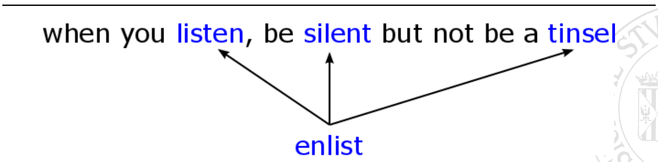
Given a pattern p and a text t, the *abelian pattern matching* problem (also known as *jumbled matching*) consists in finding all substrings of the text t, whose characters have the same multiplicities as in p, so that they could be converted into the input pattern just by permuting their characters.



The counting filter

In the field of text processing and in computational biology, algorithms for abelian pattern matching are used as a filtering technique [Baeza-Yates.Navarro.2002]:

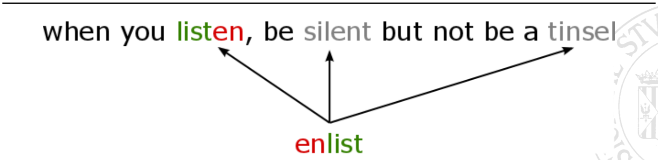
- *k*-mismatches [Grossi.Luccio.1989];
- *k*-differences [Jokinen.Tarhio.Ukkonen1996];
- inversions [Cantone.Cristofaro.Faro.2011];
- inversions and translocations [Cantone.Faro.Giaquinta.2011].



The counting filter

In the field of text processing and in computational biology, algorithms for abelian pattern matching are used as a filtering technique [Baeza-Yates.Navarro.2002]:

- *k*-mismatches [Grossi.Luccio.1989];
- *k*-differences [Jokinen.Tarhio.Ukkonen1996];
- inversions [Cantone.Cristofaro.Faro.2011];
- inversions and translocations [Cantone.Faro.Giaquinta.2011].



The Parikh vector

The *Parikh vector* of p is the vector of the multiplicities of the characters in p. More precisely, for each $c \in \Sigma$, we have

$$pv_{\mathrm{p}}[c] = |\{i : 0 \leq i < m \text{ and } \mathrm{p}[i] = c\}| \,.$$

The Parikh vector

The *Parikh vector* of p is the vector of the multiplicities of the characters in p. More precisely, for each $c \in \Sigma$, we have

$$pv_{\mathrm{p}}[c] = \left|\{i : 0 \leq i < m \text{ and } \mathrm{p}[i] = c\}\right| .$$

$$\mathrm{p} = \text{enlist}$$
$$pv_{\mathrm{p}}[e] = 1,\ pv_{\mathrm{p}}[a] = 0,\ pv_{\mathrm{p}}[t] = 1,\ pv_{\mathrm{p}}[c] = 0$$

$$\mathrm{p} = \text{stringology}$$
$$pv_{\mathrm{p}}[s] = 1,\ pv_{\mathrm{p}}[o] = 2,\ pv_{\mathrm{p}}[a] = 0,\ pv_{\mathrm{p}}[g] = 2,$$

Note. The Parikh vector of the substring $\mathrm{p}[i \mathinner{..} i+h-1]$ of p, of length h and starting at position i, will be denoted by $pv_{\mathrm{p}(i,h)}$.

The Parikh vector

In terms of Parikh vectors, the abelian pattern matching problem can be formally expressed as the problem of finding the set $\Gamma_{\mathsf{p},\mathsf{t}}$ of positions in t, defined as

$$\Gamma_{\mathsf{p},\mathsf{t}} = \{s : \ 0 \leq s \leq n - m \text{ and } pv_{\mathsf{t}(s,m)} = pv_{\mathsf{p}}\}.$$

The naïve solution

For a pattern p of length m and a text t of length n over an alphabet Σ of size σ, the *abelian pattern matching problem* can be solved in $O(n)$ time and $O(\sigma)$ space by using a naïve *prefix based approach* [Ejaz.2010].

For each position $s = 0, 1, \ldots, n-m-1$ and character $c \in \Sigma$, we have

$$pv_{\mathrm{t}(s+1,m)}[c] = pv_{\mathrm{t}(s,m)}[c] - \left|\{c\} \cap \{\mathrm{t}[s]\}\right| + \left|\{c\} \cap \{\mathrm{t}[s+m]\}\right| ,$$

---

when you listen, be silent but not be a tinsel

`b:0   e:1   i:1   l:1   n:1   s:1   t:1`

`b:0   e:1   i:1   l:1   n:1   s:1   t:1`

Another efficient prefix-based approach, which uses less branch conditions, has been proposed in [Grabowski.Faro.Giaquinta.2011]

The naïve solution

For a pattern p of length m and a text t of length n over an alphabet Σ of size σ, the *abelian pattern matching problem* can be solved in $O(n)$ time and $O(\sigma)$ space by using a naïve *prefix based approach* [Ejaz.2010].

For each position $s = 0, 1, \ldots, n-m-1$ and character $c \in \Sigma$, we have

$$pv_{\mathrm{t}(s+1,m)}[c] = pv_{\mathrm{t}(s,m)}[c] - \left|\{c\} \cap \{\mathrm{t}[s]\}\right| + \left|\{c\} \cap \{\mathrm{t}[s+m]\}\right| ,$$

when you listen, be silent but not be a tinsel

`b:1   e:1   i:1   l:0   n:1   s:1   t:1`

`b:0   e:1   i:1   l:1   n:1   s:1   t:1`

Another efficient prefix-based approach, which uses less branch conditions, has been proposed in [Grabowski.Faro.Giaquinta.2011]

The naïve solution

For a pattern p of length m and a text t of length n over an alphabet Σ of size σ, the *abelian pattern matching problem* can be solved in $O(n)$ time and $O(\sigma)$ space by using a naïve *prefix based approach* [Ejaz.2010].

For each position $s = 0, 1, \ldots, n - m - 1$ and character $c \in \Sigma$, we have

$$pv_{\mathrm{t}(s+1,m)}[c] = pv_{\mathrm{t}(s,m)}[c] - \left|\{c\} \cap \{\mathrm{t}[s]\}\right| + \left|\{c\} \cap \{\mathrm{t}[s+m]\}\right| ,$$

when you listen, be silent but not be a tinsel

`b:1   e:2   i:0   l:0   n:1   s:1   t:1`

`b:0   e:1   i:1   l:1   n:1   s:1   t:1`

Another efficient prefix-based approach, which uses less branch conditions, has been proposed in [Grabowski.Faro.Giaquinta.2011]

The naïve solution

For a pattern p of length m and a text t of length n over an alphabet Σ of size σ, the *abelian pattern matching problem* can be solved in $O(n)$ time and $O(\sigma)$ space by using a naïve *prefix based approach* [Ejaz.2010].

For each position $s = 0, 1, \dots, n - m - 1$ and character $c \in \Sigma$, we have

$$pv_{\mathrm{t}(s+1,m)}[c] = pv_{\mathrm{t}(s,m)}[c] - \left|\{c\} \cap \{\mathrm{t}[s]\}\right| + \left|\{c\} \cap \{\mathrm{t}[s+m]\}\right| ,$$

---

when you listen, be silent but not be a tinsel

b:1	e:2	i:0	l:0	n:1	s:1	t:1
b:0	e:1	i:1	l:1	n:1	s:1	t:1

Another efficient prefix-based approach, which uses less branch conditions, has been proposed in [Grabowski.Faro.Giaquinta.2011]

The naïve solution

For a pattern p of length m and a text t of length n over an alphabet Σ of size σ, the *abelian pattern matching problem* can be solved in $O(n)$ time and $O(\sigma)$ space by using a naïve *prefix based approach* [Ejaz.2010].

For each position $s = 0, 1, \ldots, n-m-1$ and character $c \in \Sigma$, we have

$$pv_{\mathrm{t}(s+1,m)}[c] = pv_{\mathrm{t}(s,m)}[c] - \big|\{c\} \cap \{\mathrm{t}[s]\}\big| + \big|\{c\} \cap \{\mathrm{t}[s+m]\}\big| \,,$$

when you listen, be silent but not be a tinsel

b:1	e:2	i:1	l:0	n:1	s:1	t:0
b:0	e:1	i:1	l:1	n:1	s:1	t:1

Another efficient prefix-based approach, which uses less branch conditions, has been proposed in [Grabowski.Faro.Giaquinta.2011]

The naïve solution

For a pattern p of length m and a text t of length n over an alphabet Σ of size σ, the *abelian pattern matching problem* can be solved in $O(n)$ time and $O(\sigma)$ space by using a naïve *prefix based approach* [Ejaz.2010].

For each position $s = 0, 1, \ldots, n - m - 1$ and character $c \in \Sigma$, we have

$$pv_{\mathrm{t}(s+1,m)}[c] = pv_{\mathrm{t}(s,m)}[c] - \left|\{c\} \cap \{\mathrm{t}[s]\}\right| + \left|\{c\} \cap \{\mathrm{t}[s+m]\}\right| ,$$

---

when you listen, be silent but not be a tinsel

b:1	e:1	i:1	l:1	n:1	s:1	t:0
b:0	e:1	i:1	l:1	n:1	s:1	t:1

Another efficient prefix-based approach, which uses less branch conditions, has been proposed in [Grabowski.Faro.Giaquinta.2011]

The naïve solution

For a pattern p of length m and a text t of length n over an alphabet Σ of size σ, the *abelian pattern matching problem* can be solved in $O(n)$ time and $O(\sigma)$ space by using a naïve *prefix based approach* [Ejaz.2010].

For each position $s = 0, 1, \ldots, n - m - 1$ and character $c \in \Sigma$, we have

$$pv_{\mathrm{t}(s+1,m)}[c] = pv_{\mathrm{t}(s,m)}[c] - \left|\{c\} \cap \{\mathrm{t}[s]\}\right| + \left|\{c\} \cap \{\mathrm{t}[s+m]\}\right| ,$$

---

when you listen, be silent but not be a tinsel

b:1 e:2 i:1 l:1 n:0 s:1 t:0

b:0 e:1 i:1 l:1 n:1 s:1 t:1

Another efficient prefix-based approach, which uses less branch conditions, has been proposed in [Grabowski.Faro.Giaquinta.2011]

The naïve solution

For a pattern p of length m and a text t of length n over an alphabet Σ of size σ, the *abelian pattern matching problem* can be solved in $O(n)$ time and $O(\sigma)$ space by using a naïve *prefix based approach* [Ejaz.2010].

For each position $s = 0, 1, \ldots, n-m-1$ and character $c \in \Sigma$, we have

$$pv_{\mathrm{t}(s+1,m)}[c] = pv_{\mathrm{t}(s,m)}[c] - \left|\{c\} \cap \{\mathrm{t}[s]\}\right| + \left|\{c\} \cap \{\mathrm{t}[s+m]\}\right| ,$$

---

when you listen, be silent but not be a tinsel

```
b:0   e:2   i:1   l:1   n:1   s:1   t:0
b:0   e:1   i:1   l:1   n:1   s:1   t:1
```

Another efficient prefix-based approach, which uses less branch conditions, has been proposed in [Grabowski.Faro.Giaquinta.2011]

The naïve solution

For a pattern p of length m and a text t of length n over an alphabet Σ of size σ, the *abelian pattern matching problem* can be solved in $O(n)$ time and $O(\sigma)$ space by using a naïve *prefix based approach* [Ejaz.2010].

For each position $s = 0, 1, \ldots, n - m - 1$ and character $c \in \Sigma$, we have

$$pv_{\mathrm{t}(s+1,m)}[c] = pv_{\mathrm{t}(s,m)}[c] - \left|\{c\} \cap \{\mathrm{t}[s]\}\right| + \left|\{c\} \cap \{\mathrm{t}[s+m]\}\right| ,$$

---

when you listen, be silent but not be a tinsel

```
b:0   e:1   i:1   l:1   n:1   s:1   t:1
b:0   e:1   i:1   l:1   n:1   s:1   t:1
```

Another efficient prefix-based approach, which uses less branch conditions, has been proposed in [Grabowski.Faro.Giaquinta.2011]

A suffix based solution

A *suffix-based approach* to the problem has been proposed as an adaptation of the Horspool strategy [Ejaz.2010].

Characters are read from right to left. As soon as a frequency overflow occurs, the reading phase is stopped and a new alignment is attempted by sliding the window to the right. The resulting algorithm has an $O(nm)$ worst-case time complexity but performs well in practical cases.

---

when you listen, be silent but not be a tinsel

```
b:0   e:0   i:0   l:0   n:0   s:0   t:0
b:0   e:1   i:1   l:1   n:1   s:1   t:1
```

A suffix based solution

A *suffix-based approach* to the problem has been proposed as an adaptation of the Horspool strategy [Ejaz.2010].

Characters are read from right to left. As soon as a frequency overflow occurs, the reading phase is stopped and a new alignment is attempted by sliding the window to the right. The resulting algorithm has an $O(nm)$ worst-case time complexity but performs well in practical cases.

when you listen, be silent but not be a tinsel

```
b:1  e:0  i:0  l:0  n:0  s:0  t:0
b:0  e:1  i:1  l:1  n:1  s:1  t:1
```

A suffix based solution

A *suffix-based approach* to the problem has been proposed as an adaptation of the Horspool strategy [Ejaz.2010].

Characters are read from right to left. As soon as a frequency overflow occurs, the reading phase is stopped and a new alignment is attempted by sliding the window to the right. The resulting algorithm has an $O(nm)$ worst-case time complexity but performs well in practical cases.

when you listen, be silent but not be a tinsel

b:0 e:0 i:0 l:0 n:0 s:0 t:0

b:0 e:1 i:1 l:1 n:1 s:1 t:1

A suffix based solution

A *suffix-based approach* to the problem has been proposed as an adaptation of the Horspool strategy [Ejaz.2010].

Characters are read from right to left. As soon as a frequency overflow occurs, the reading phase is stopped and a new alignment is attempted by sliding the window to the right. The resulting algorithm has an $O(nm)$ worst-case time complexity but performs well in practical cases.

when you listen, be silent but not be a tinsel

```
b:1   e:0   i:0   l:0   n:0   s:0   t:0
b:0   e:1   i:1   l:1   n:1   s:1   t:1
```

A suffix based solution

A *suffix-based approach* to the problem has been proposed as an adaptation of the Horspool strategy [Ejaz.2010].

Characters are read from right to left. As soon as a frequency overflow occurs, the reading phase is stopped and a new alignment is attempted by sliding the window to the right. The resulting algorithm has an $O(nm)$ worst-case time complexity but performs well in practical cases.

---

when you listen, be silent but not be a tinsel

```
b:0  e:0  i:0  l:0  n:0  s:0  t:0
b:0  e:1  i:1  l:1  n:1  s:1  t:1
```

A suffix based solution

A *suffix-based approach* to the problem has been proposed as an adaptation of the Horspool strategy [Ejaz.2010].

Characters are read from right to left. As soon as a frequency overflow occurs, the reading phase is stopped and a new alignment is attempted by sliding the window to the right. The resulting algorithm has an $O(nm)$ worst-case time complexity but performs well in practical cases.

when you listen, be silent but not be a tinsel

```
b:0  e:0  i:0  l:0  n:0  s:1  t:0
b:0  e:1  i:1  l:1  n:1  s:1  t:1
```

A suffix based solution

A *suffix-based approach* to the problem has been proposed as an adaptation of the Horspool strategy [Ejaz.2010].

Characters are read from right to left. As soon as a frequency overflow occurs, the reading phase is stopped and a new alignment is attempted by sliding the window to the right. The resulting algorithm has an $O(nm)$ worst-case time complexity but performs well in practical cases.

when you listen, be silent but not be a tinsel

```
b:0  e:0  i:0  l:0  n:1  s:1  t:0
b:0  e:1  i:1  l:1  n:1  s:1  t:1
```

A suffix based solution

A *suffix-based approach* to the problem has been proposed as an adaptation of the Horspool strategy [Ejaz.2010].

Characters are read from right to left. As soon as a frequency overflow occurs, the reading phase is stopped and a new alignment is attempted by sliding the window to the right. The resulting algorithm has an $O(nm)$ worst-case time complexity but performs well in practical cases.

---

when you listen, be silent but not be a tinsel

```
b:0   e:0   i:1   l:0   n:1   s:1   t:0
b:0   e:1   i:1   l:1   n:1   s:1   t:1
```

A suffix based solution

A *suffix-based approach* to the problem has been proposed as an adaptation of the Horspool strategy [Ejaz.2010].

Characters are read from right to left. As soon as a frequency overflow occurs, the reading phase is stopped and a new alignment is attempted by sliding the window to the right. The resulting algorithm has an $O(nm)$ worst-case time complexity but performs well in practical cases.

---

when you listen, be silent but not be a tinsel

`b:0 e:0 i:1 l:0 n:1 s:1 t:1`

`b:0 e:1 i:1 l:1 n:1 s:1 t:1`

For more details ...

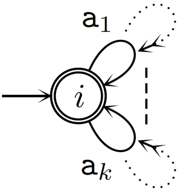
A detailed analysis of the abelian pattern matching problem and of its solutions is presented in [Ejaz.2010].

Reactive Automata

A reactive automaton is an ordinary automaton extended with *reactive links* between its (ordinary) links. These can be of two types

- *activation* reactive links;
- *deactivation* reactive links.

At any step of the computation of a reactive automaton on a given input string S, states and links are distinguished as active and non-active.

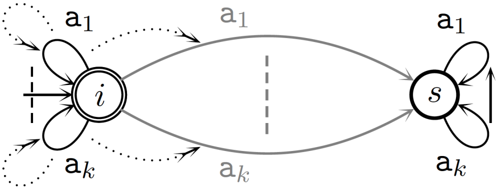


Reactive Automata

A reactive automaton is an ordinary automaton extended with *reactive links* between its (ordinary) links. These can be of two types

- *activation* reactive links;
- *deactivation* reactive links.

At any step of the computation of a reactive automaton on a given input string S, states and links are distinguished as active and non-active.

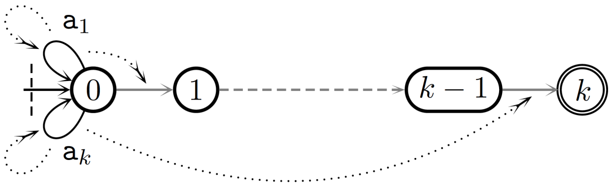


Reactive Automata

A reactive automaton is an ordinary automaton extended with *reactive links* between its (ordinary) links. These can be of two types

- *activation* reactive links;
- *deactivation* reactive links.

At any step of the computation of a reactive automaton on a given input string S, states and links are distinguished as active and non-active.

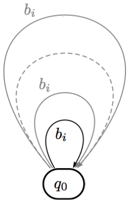


Reactive Multi-Automata

Reactive multi-automata extend reactive automata in that they allow the presence of multiple links labeled by a same character between any two states.

Reactive Multi-Automata

Reactive multi-automata extend reactive automata in that they allow the presence of multiple links labeled by a same character between any two states.



Definition (Reactive multi-automata)

Let Q, Σ, L be finite sets of states, of characters, and of labels, respectively.

A *reactive multi-automaton* is a nonuple $\mathcal{R} = \big(Q, \Sigma, L, q_0, \delta, \overline{\delta}, T^+, T^-, F\big)$, where

- $(Q, \Sigma, L, q_0, \delta, F)$ is a *multi-automaton* (called the *multi-automaton underlying* $\mathcal{R}$), with $q_0 \in Q$ (initial state), $F \subseteq Q$ (set of final states), and $\delta \subseteq Q \times \Sigma \times L \times Q$ (transition relation);
- $T^+, T^- \subseteq \delta \times \delta$ are the sets of activation and deactivation reactive links;
- $\overline{\delta} \subseteq \delta$ is the set of initially active links (initial transition relation).

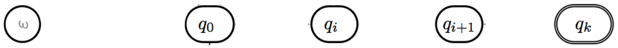
The Abelian Reactive Multi-Automaton

Let p be a pattern of length m over an alphabet Σ and let $\langle b_0, b_1, \ldots, b_{k-1} \rangle$ be the sequence of the distinct characters occurring in p, ordered by their first occurrence. The *abelian reactive multi-automaton* (ARMA) for p is the reactive multi-automaton with ε-transitions

$$\mathcal{R} = \left(Q, \Sigma, L, q_0, \delta, \overline{\delta}, T^+, T^-, F\right)$$

The Set of States of the Automaton

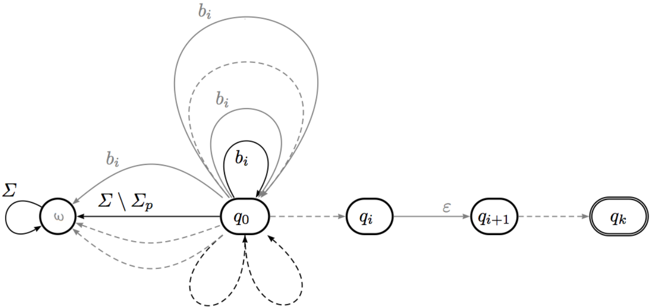
- $Q = \{q_0, q_1, \ldots, q_k, \omega\}$ is the set of states;
- q_0 is the initial state; ω is a special state called the *overflow state*;
- $F = \{q_k\}$ is the set of final states;



The Full Transition Relation

the transition relation δ of $\mathcal{R}$ and its subset $\overline{\delta} \subseteq \delta$ of the links initially active (initial transition relation) are defined as follows

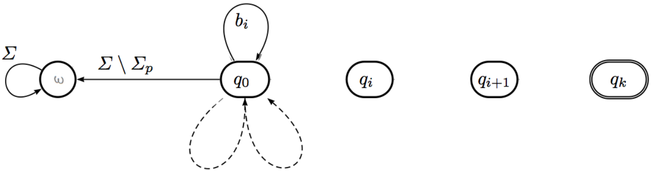
$$\begin{aligned}\delta = \quad & \{(q_i, \varepsilon, \ell_0, q_{i+1}) \mid 0 \leq i < k\} && (\varepsilon\text{-transitions}) \\ & \cup \{(q_0, p[i], \ell_i, q_0) \mid 0 \leq i < m\} && (\text{self-loops}) \\ & \cup \{(q_0, c, \ell_0, \omega) \mid c \in \Sigma\} && (\text{overflow transitions})\end{aligned}$$



The Initial Transition Relation

the transition relation δ of $\mathcal{R}$ and its subset $\overline{\delta} \subseteq \delta$ of the links initially active (initial transition relation) are defined as follows

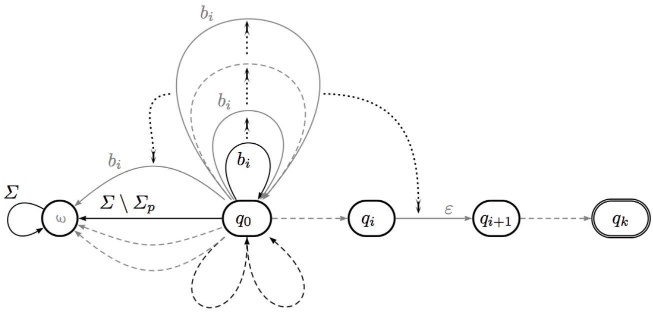
$$\begin{aligned}\overline{\delta} = \quad & \{(q_0, c, \ell_{\lambda(c)}, q_0) \mid c \in \Sigma_p\} \\ & \cup \{(q_0, c, \ell_0, \omega) \mid c \in \Sigma \setminus \Sigma_p\} \\ & \cup \{(\omega, c, \ell_0, \omega) \mid c \in \Sigma\}\end{aligned}$$



The Set of Activation Reactive Links

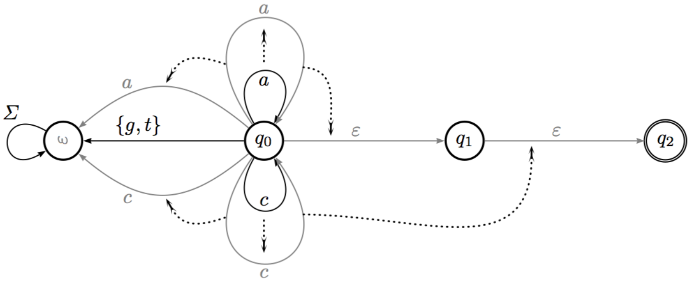
The set T^{+} of activation reactive links are defined as follows

$$
\begin{aligned}
T^{+} = \quad & \{((q_0, \mathrm{p}[\rho(b_i)], \ell_{\rho(b_i)}, q_0), (q_i, \varepsilon, \ell_0, q_{i+1})) \mid 0 \leq i < k\} \\
& \cup \{((q_0, \mathrm{p}[\rho(b_i)], \ell_{\rho(b_i)}, q_0), (q_0, \mathrm{p}[\rho(b_i)], \ell_{\rho(b_i)}, \omega)) \mid 0 \leq i < k\} \\
& \cup \{((q_0, \mathrm{p}[i], \ell_i, q_0), (q_0, \mathrm{p}[\nu(i)], \ell_{\nu(i)}, q_0)) \mid 0 \leq i < m \text{ and } i \neq \rho(p_i)\}
\end{aligned}
$$



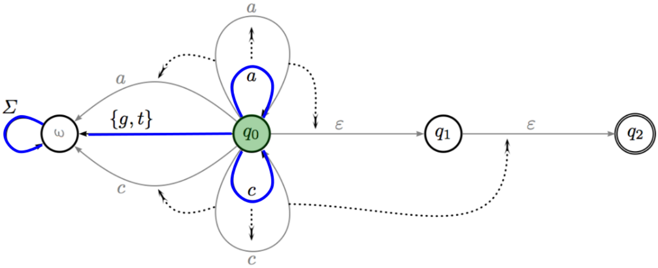
The abelian reactive automaton for *acca*

The complete abelian reactive automaton for the pattern $P = acca$ over the DNA alphabet $\Sigma = \{a, c, g, t\}$. Standard transitions are represented with solid lines while reactive links in T^+ are represented with dashed lines. Reactive links in T^- are not represented. Non active transitions are represented in gray color.



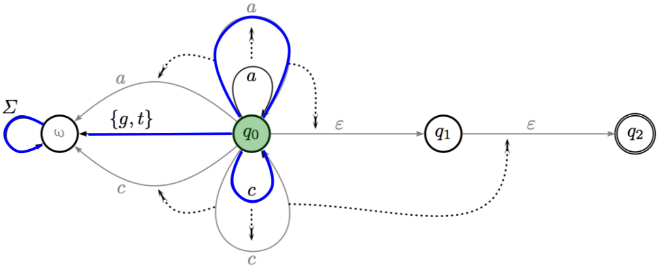
An example

p = acca
t = gtcaaaccgtacgagtacgat...



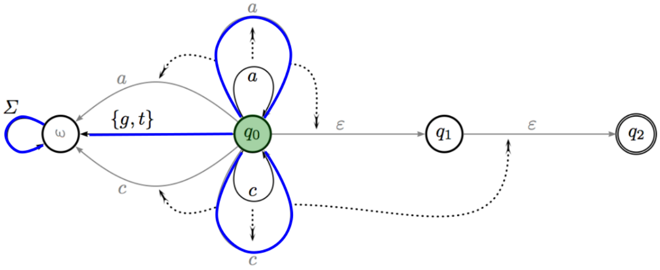
An example

p = acca
t = gtcaaaccgtacgagtacgat...



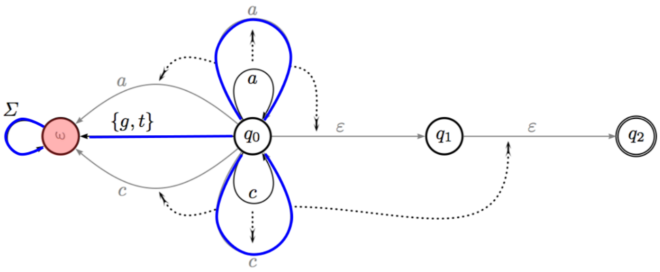
An example

p = acca
t = gtcaaaccgtacgagtacgat...



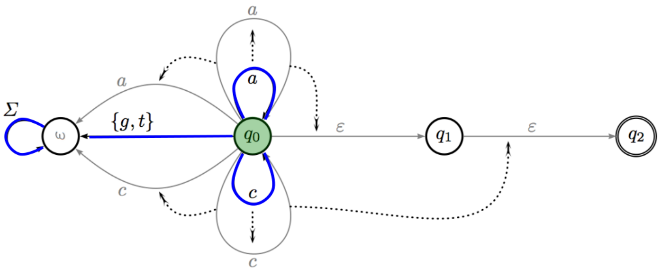
An example

p = acca
t = gtcaaaccgtacgagtacgat...



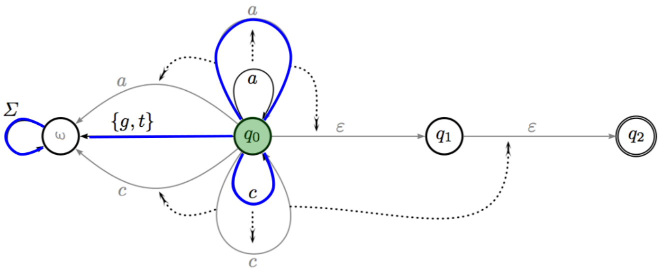
An example

p = acca
t = gtcaaaccgtacgagtacgat...



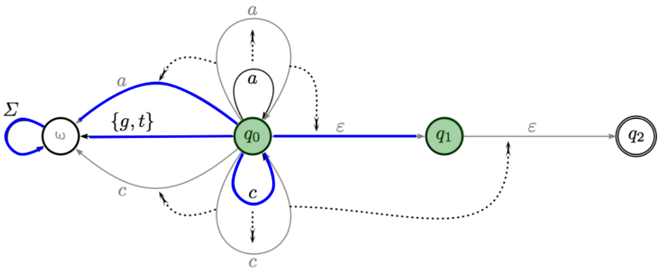
An example

p = acca
t = gtcaaaccgtacgagtacgat...



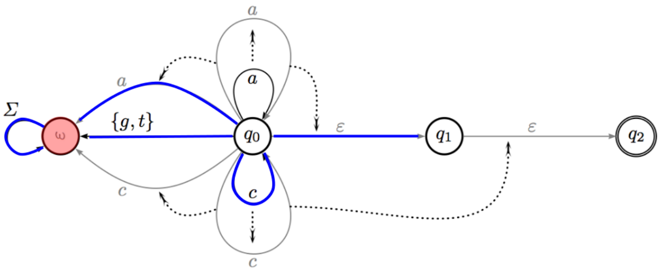
An example

p = acca
t = gtcaaaccgtacgagtacgat...



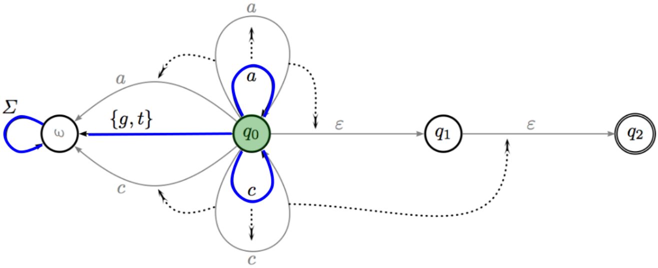
An example

p = acca
t = gtcaaaccgtacgagtacgat...



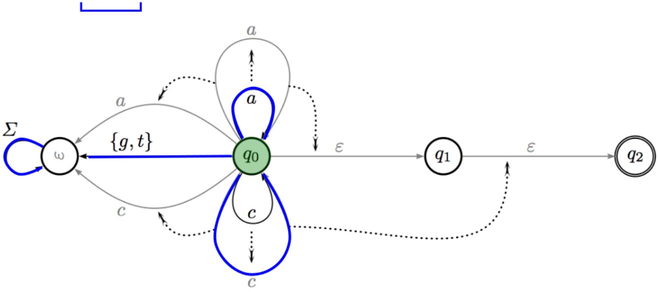
An example

p = acca
t = gtcaaaccgtacgagtacgat...



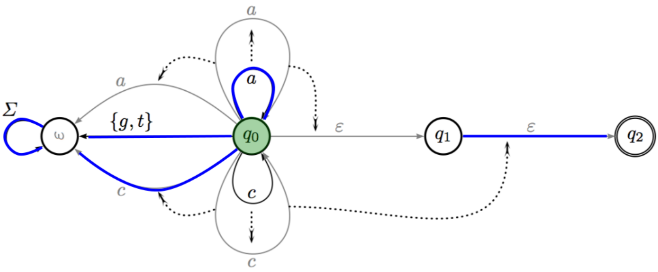
An example

p = acca
t = gtcaaaccgtacgagtacgat...



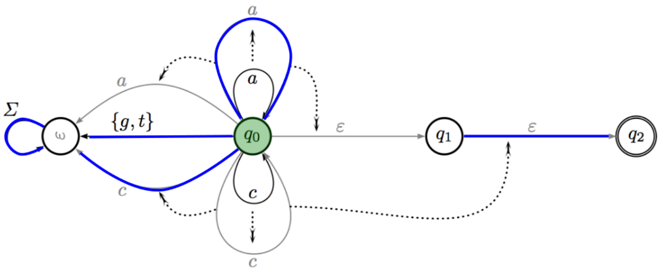
An example

p = acca
t = gtcaaaccgtacgagtacgat...



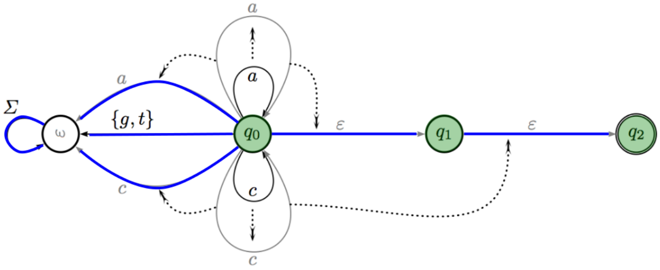
An example

p = acca
t = gtcaaaccgtacgagtacgat...



An example

p = acca
t = gtcaaaccgtacgagtacgat...



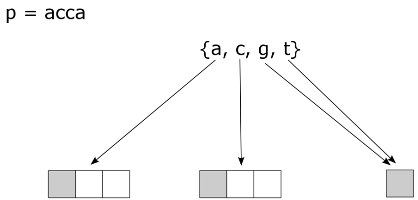
A Bit-Parallel Simulation

The underlying idea is to associate a counter to each distinct character in p, plus a single 1-bit counter for the remaining characters of the alphabet which do not occur in p, maintaining them in the same computer word.

A Bit-Parallel Simulation

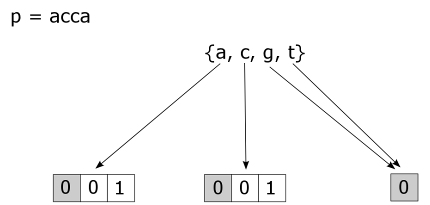
The underlying idea is to associate a counter to each distinct character in p, plus a single 1-bit counter for the remaining characters of the alphabet which do not occur in p, maintaining them in the same computer word.

The counter associated to the character b_i in p is represented by a group of l_i bits, where $l_i = \lceil \log(pv_{\mathrm{p}}[b_i]) + 1 \rceil + 1$.



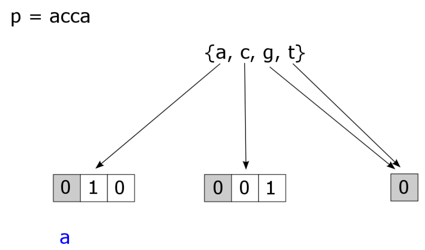
A Bit-Parallel Simulation

- Initially, the counter for b_i is set to the value $2^{l_i} - pv_{\mathrm{p}}[b_i] - 1$, so that its overflow bit is 0 and it remains so for up to $pv_{\mathrm{p}}[b_i]$ increments.
- The overflow bit gets set only when the $(pv_{\mathrm{p}}[b_i] + 1)$-st occurrence of b_i is encountered in the text window.



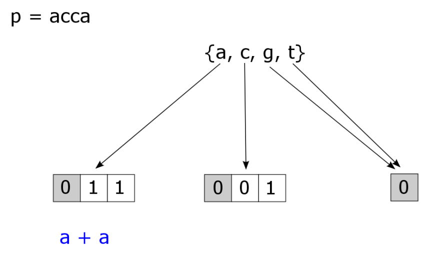
A Bit-Parallel Simulation

- Initially, the counter for b_i is set to the value $2^{l_i} - pv_{\mathrm{p}}[b_i] - 1$, so that its overflow bit is 0 and it remains so for up to $pv_{\mathrm{p}}[b_i]$ increments.
- The overflow bit gets set only when the $(pv_{\mathrm{p}}[b_i] + 1)$-st occurrence of b_i is encountered in the text window.



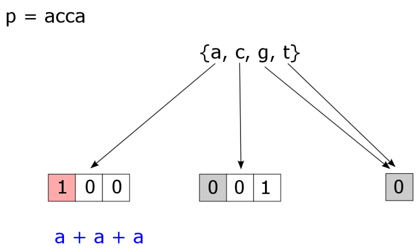
A Bit-Parallel Simulation

- Initially, the counter for b_i is set to the value $2^{l_i} - pv_{\mathrm{p}}[b_i] - 1$, so that its overflow bit is 0 and it remains so for up to $pv_{\mathrm{p}}[b_i]$ increments.
- The overflow bit gets set only when the $(pv_{\mathrm{p}}[b_i] + 1)$-st occurrence of b_i is encountered in the text window.



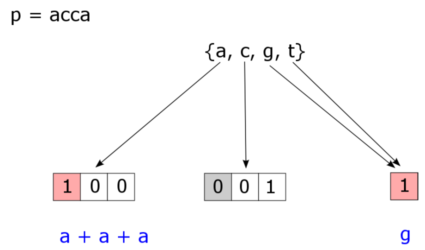
A Bit-Parallel Simulation

- Initially, the counter for b_i is set to the value $2^{l_i} - pv_{\mathrm{p}}[b_i] - 1$, so that its overflow bit is 0 and it remains so for up to $pv_{\mathrm{p}}[b_i]$ increments.
- The overflow bit gets set only when the $(pv_{\mathrm{p}}[b_i] + 1)$-st occurrence of b_i is encountered in the text window.



A Bit-Parallel Simulation

- Likewise, the 1-bit counter reserved for all the characters not occurring in p is initially null and it gets set as soon as any character not in p is encountered in the text window.



The Preprocessing Phase

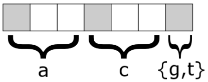
for each distinct character b_i occurring in p we compute, a bit mask $M[b_i]$ of $l+1$ bits is computed, where

$$l = \sum_{i=0}^{k-1} l_i \qquad \text{and} \qquad M[b_i] = 1 \ll \left(\sum_{j=0}^{i-1} l_j \right).$$

The bit mask $M[b_i]$ is then used to increment the counter in D associated to the character b_i.

---

p = acca



The Preprocessing Phase

for each distinct character b_i occurring in p we compute, a bit mask $M[b_i]$ of $l+1$ bits is computed, where

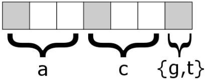
$$l = \sum_{i=0}^{k-1} l_i \qquad \text{and} \qquad M[b_i] = 1 \ll \left(\sum_{j=0}^{i-1} l_j \right).$$

The bit mask $M[b_i]$ is then used to increment the counter in D associated to the character b_i.

---

p = acca

M[a]	0	0	1	0	0	0	0
M[c]	0	0	0	0	0	1	0
M[g]	0	0	0	0	0	0	1



The Preprocessing Phase

Two additional bit masks are used:

- the bit mask I, which contains the initial values for each counter

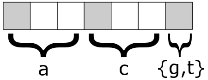
$$I = \sum_{i=0}^{k-1} \left[\left(2^{l_i} - pv_{\mathrm{p}}[b_i] - 1 \right) \ll \sum_{j=0}^{i-1} l_j \right]$$

---

p = acca

M[a]	0	0	1	0	0	0	0
M[c]	0	0	0	0	0	1	0
M[g]	0	0	0	0	0	0	1

I	0	0	1	0	0	1	0



The Preprocessing Phase

Two additional bit masks are used:

- the bit mask F, whose bits set are exactly the overflow bits.

$$F = \sum_{i=0}^{k-1} \left[1 \ll \left(\sum_{j=0}^{i} l_j - 1 \right) \right]$$

---

p = acca

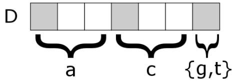
M[a]	0	0	1	0	0	0	0
M[c]	0	0	0	0	0	1	0
M[g]	0	0	0	0	0	0	1
I	0	0	1	0	0	1	0
F	1	0	0	1	0	0	1
	a			c			{g,t}

The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to I.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...

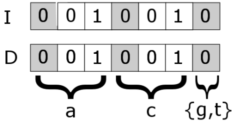


The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to I.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...



The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to l.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...

M[a]	0	0	1	0	0	0	0
D	0	0	1	0	0	1	0

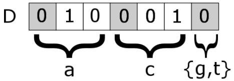
$\underbrace{0\ 0\ 1}_{a}\ \underbrace{0\ 0\ 1}_{c}\ \underbrace{0}_{\{g,t\}}$

The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to l.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...

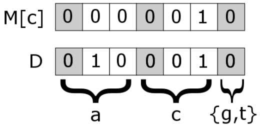


The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to I.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...

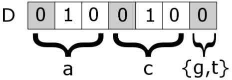


The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to I.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...



The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to l.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...

M[t]	0	0	0	0	0	0	1
D	0	1	0	0	1	0	0
		a			c		{g,t}

The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to I.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...

$$D \quad \underbrace{\boxed{0}\boxed{1}\boxed{0}}_{a}\ \underbrace{\boxed{0}\boxed{1}\boxed{0}}_{c}\ \underbrace{\boxed{1}}_{\{g,t\}}$$

The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to I.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character $\mathrm{t}[j]$ of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...

I	0	0	1	0	0	1	0
D	0	0	1	0	0	1	0

$$D = \underbrace{0\ 0\ 1}_{a}\ \underbrace{0\ 0\ 1}_{c}\ \underbrace{0}_{\{g,t\}}$$

The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to l.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...

M[a]	0	0	1	0	0	0	0
D	0	0	1	0	0	1	0

$\underbrace{0\ 0\ 1}_{a}\ \underbrace{0\ 0\ 1}_{c}\ \underbrace{0}_{\{g,t\}}$

The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to I.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...

D	0	1	0	0	0	1	0
	a			c			{g,t}

The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to I.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M$[t[j]].

---

p = acca
t = gtcaaaccgtacgagtacgat...

M[a]	0	0	1	0	0	0	0
D	0	1	0	0	0	1	0
	a			c			{g,t}

The Searching Phase

- At the beginning of each attempt, a bit mask D of $l + 1$ bits is initialized to I.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...

$$D \quad \underbrace{\boxed{0}\boxed{1}\boxed{1}}_{a}\ \underbrace{\boxed{0}\boxed{0}\boxed{1}}_{c}\ \underbrace{\boxed{0}}_{\{g,t\}}$$

The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to I.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...

M[a]	0	0	1	0	0	0	0
D	0	1	1	0	0	1	0
		a			c		{g,t}

The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to I.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...

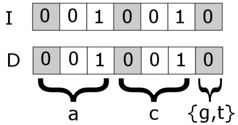
$$D \quad \underbrace{1\ 0\ 0}_{a}\ \underbrace{0\ 0\ 1}_{c}\ \underbrace{0}_{\{g,t\}}$$

The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to I.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...



The Searching Phase

- At the beginning of each attempt, a bit mask D of $l + 1$ bits is initialized to I.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...

$$M[c] = \boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{0}\boxed{1}\boxed{0}$$

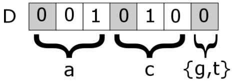
$$D = \underbrace{\boxed{0}\boxed{0}\boxed{1}}_{a}\underbrace{\boxed{0}\boxed{0}\boxed{1}}_{c}\underbrace{\boxed{0}}_{\{g,t\}}$$

The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to I.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...

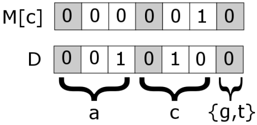


The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to I.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...



The Searching Phase

- At the beginning of each attempt, a bit mask D of $l + 1$ bits is initialized to I.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...

$$D\quad \underbrace{\boxed{0}\boxed{0}\boxed{1}}_{a}\ \underbrace{\boxed{0}\boxed{1}\boxed{1}}_{c}\ \underbrace{\boxed{0}}_{\{g,t\}}$$

The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to l.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...

M[a]	0	0	1	0	0	0	0
D	0	0	1	0	1	1	0

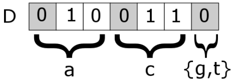
$$D = \underbrace{0\;0\;1}_{a}\ \underbrace{0\;1\;1}_{c}\ \underbrace{0}_{\{g,t\}}$$

The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to I.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character $t[j]$ of the text, the bit mask D is updated accordingly by setting it to $D + M[t[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...



The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to l.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\mathrm{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...

$$M[a] = \boxed{0}\boxed{0}\boxed{1}\boxed{0}\boxed{0}\boxed{0}\boxed{0}$$

$$D = \underbrace{\boxed{0}\boxed{1}\boxed{0}}_{a}\underbrace{\boxed{0}\boxed{1}\boxed{1}}_{c}\underbrace{\boxed{0}}_{\{g,t\}}$$

The Searching Phase

- At the beginning of each attempt, a bit mask D of $l+1$ bits is initialized to I.
- Then, during the attempt, the window is read character by character, proceeding from right to left.
- When reading the character t[j] of the text, the bit mask D is updated accordingly by setting it to $D + M[\text{t}[j]]$.

---

p = acca
t = gtcaaaccgtacgagtacgat...

$$D \quad \underbrace{\boxed{0}\,\boxed{1}\,\boxed{1}}_{a}\;\underbrace{\boxed{0}\,\boxed{1}\,\boxed{1}}_{c}\;\underbrace{\boxed{0}}_{\{g,t\}}$$

The Searching Phase

When $l + 1 > w$, we must content ourselves to maintain the counters only for a proper selection Σ'_p of the set of characters occurring in p. In this case, when a match relative to the characters in Σ'_p is reported, an additional verification phase must be run, in order to discard possible *false positives*.

```
BAM(p, m, t, n, Σ)
    for each c ∈ Σ do M[c] ← pv_p[c] ← 0
    I ← F ← sh ← 0
    for i ← 0 to m − 1 do pv_p[p[i]] ← pv_p[p[i]] + 1
    for each c ∈ Σ do
        if pv_p[c] > 0 then
            M[c] ← M[c] | (1 ≪ sh)
            I ← I | (((1 ≪ log m) − pv_p[c] − 1) ≪ sh)
            F ← F | (1 ≪ (sh + log m))
            sh ← sh + log m + 1
    F ← F | (1 ≪ sh)
    for each c ∈ Σ do
        if pv_p[c] = 0 then M[c] ← M[c] | (1 ≪ sh)
    s ← 0
    while s ≤ n − m do
        D ← I; j ← s + m − 1
        while j ≥ s do
            D ← D + M[t[j]]
            if (D & F) then break
            j ← j − 1
        if j < s then
            OUTPUT(s)
            s ← s + 1
        else s ← j + 1
```

Experimental Results

In this section we evaluate the performance of the bit-parallel simulation BAM described in the previous section and compare it with some standard solutions known in literature.

We compare the performances of the following algorithms:

- The prefix based algorithm due to Grabowsky *et al.* (GFG);
- The algorithm using the suffix based approach (SBA),
- The Bit-parallel Abelian Matcher (BAM) described in this paper.

Experimental Results

In this section we evaluate the performance of the bit-parallel simulation BAM described in the previous section and compare it with some standard solutions known in literature.

We compare the performances of the following algorithms:

- The prefix based algorithm due to Grabowsky *et al.* (GFG);
- The algorithm using the suffix based approach (SBA),
- The Bit-parallel Abelian Matcher (BAM) described in this paper.

We use the following text buffers

- a genome sequence (alphabet of 4 characters);
- a protein sequence (alphabet of 20 characters);

Experimental Results

m	GFG	SBA	BAM
2	**23.56**	39.20	27.03
4	**23.56**	33.27	23.17
8	23.54	27.54	**19.01**
16	23.49	24.05	**16.21**
32	23.52	23.78	**15.63**
64	23.50	25.33	**16.12**
128	23.57	28.74	**17.69**
256	23.53	33.14	**19.63**

m	GFG	SBA	BAM
2	23.08	**18.07**	12.51
4	23.00	**15.39**	10.36
8	22.96	13.67	**9.40**
16	23.03	11.91	**8.44**
32	23.04	9.58	**7.16**
64	23.01	8.46	**6.64**
128	22.97	7.82	**6.49***
256	22.96	7.84	**7.69***

Tabella: Experimental results on a genome sequence (on the left) and a on a protein sequence (on the right). An asterisk symbol (*) indicates those runs where false positives have been detected. All best results have been boldfaced.