UA RUHR

Algorithmen auf Sequenzen

Erweiterte Muster

Sven Rahmann

Genominformatik
Universitätsklinikum Essen
Universität Duisburg-Essen
Universitätsallianz Ruhr

Übersicht

UA RUHR

- Bisher wurden nur einfache Muster (Strings) behandelt
- In vielen Anwendungen ist es interessant, Muster zu betrachten, bei denen an bestimmten Stellen verschiedene Zeichen (oder beliebige Zeichen) stehen dürfen,
- oder Folgen beliebiger Zeichen mit variabler (beschränkter) Länge,
- oder Muster mit optionalen Zeichen.

Übersicht

UA RUHR

- Bisher wurden nur einfache Muster (Strings) behandelt
- In vielen Anwendungen ist es interessant, Muster zu betrachten, bei denen an bestimmten Stellen verschiedene Zeichen (oder beliebige Zeichen) stehen dürfen,
- oder Folgen beliebiger Zeichen mit variabler (beschränkter) Länge,
- oder Muster mit optionalen Zeichen.
- Dies sind Teilmengen von regulären Ausdrücken, so dass man hierauf aufbauend effiziente Algorithmen zum Suchen nach regulären Ausdrücken (grep-Tool) entwickeln kann.

Übersicht

UA RUHR

- Bisher wurden nur einfache Muster (Strings) behandelt
- In vielen Anwendungen ist es interessant, Muster zu betrachten, bei denen an bestimmten Stellen verschiedene Zeichen (oder beliebige Zeichen) stehen dürfen,
- oder Folgen beliebiger Zeichen mit variabler (beschränkter) Länge,
- oder Muster mit optionalen Zeichen.
- Dies sind Teilmengen von regulären Ausdrücken, so dass man hierauf aufbauend effiziente Algorithmen zum Suchen nach regulären Ausdrücken (grep-Tool) entwickeln kann.
- Alle hier genannten Erweiterungen lassen sich durch kleine Modifikationen des Shift-And-Algorithmus realisieren.

Verallgemeinerte Strings

- Verallgemeinerte Strings über Σ sind Strings, die *Teilmengen* von Σ als Zeichen besitzen, also Strings über 2^{Σ}.
- Mustermenge $\{\text{Meier, Meyer}\} \mapsto$ Me[iy]er;
  M Abkürzung für Menge {M}; [iy] für {i,y}
- Wir schreiben # für $\Sigma \in 2^{\Sigma}$ (beliebiges Zeichen aus Σ)

Verallgemeinerte Strings

- Verallgemeinerte Strings über Σ sind Strings, die *Teilmengen* von Σ als Zeichen besitzen, also Strings über 2^{Σ}.
- Mustermenge $\{\text{Meier, Meyer}\} \mapsto$ Me[iy]er;
  M Abkürzung für Menge {M}; [iy] für {i,y}
- Wir schreiben # für $\Sigma \in 2^{\Sigma}$ (beliebiges Zeichen aus Σ)
- Erweiterung des Shift-And-Algorithmus durch Anpassung der Bitmasken

Verallgemeinerte Strings

- Verallgemeinerte Strings über Σ sind Strings, die *Teilmengen* von Σ als Zeichen besitzen, also Strings über 2^{Σ}.
- Mustermenge {Meier, Meyer} $\mapsto$ Me[iy]er;
  M Abkürzung für Menge {M}; [iy] für {i,y}
- Wir schreiben # für $\Sigma \in 2^{\Sigma}$ (beliebiges Zeichen aus Σ)
- Erweiterung des Shift-And-Algorithmus durch Anpassung der Bitmasken
- Beispiel: $P = \texttt{abba\#b}$ mit $\Sigma = \{\texttt{a}, \texttt{b}\}$.

	b#abba
$mask^a$	011001
$mask^b$	110110

Beliebige Zeichenfolgen beschränkter Länge

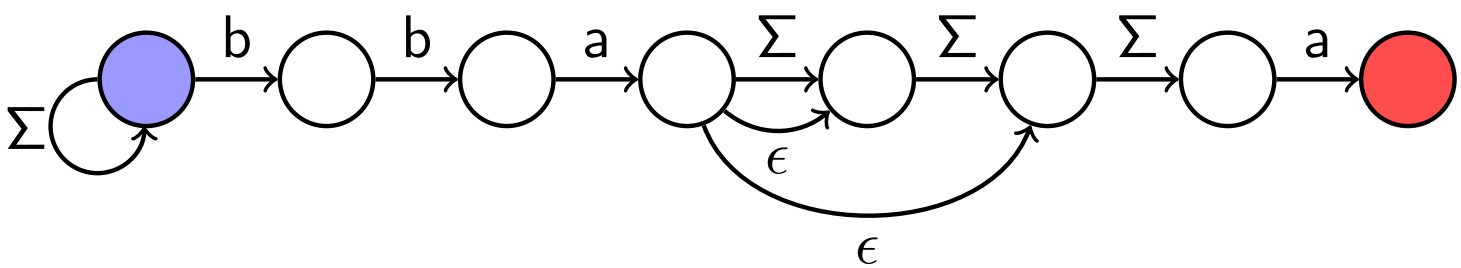
UA RUHR

- Unter beliebigen Zeichenfolgen beschränkter Länge verstehen wir Folgen von konsekutiven Σ (als # geschrieben) mit fest definierter unterer und oberer Schranke für die Länge.
- $P = \texttt{bba\#(1,3)a}$:
  1 bis 3 beliebige Zeichen; 1 notwendig, 2 optional

Beliebige Zeichenfolgen beschränkter Länge

UA RUHR

- Unter beliebigen Zeichenfolgen beschränkter Länge verstehen wir Folgen von konsekutiven Σ (als # geschrieben) mit fest definierter unterer und oberer Schranke für die Länge.
- $P = \texttt{bba\#(1,3)a}$:
  1 bis 3 beliebige Zeichen; 1 notwendig, 2 optional
- Realisierung mit ϵ-Übergängen im NFA,
  ϵ-Kanten gehen vom **Beginn** des Elements aus!



Beliebige Zeichenfolgen beschränkter Länge

Einschränkungen:

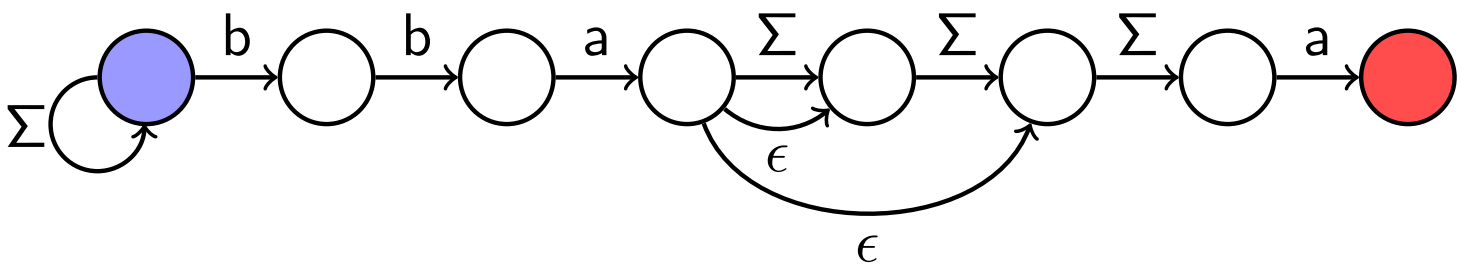
1. Elemente $\#(,)$ nicht an erster oder letzter Stelle im Muster
2. Nicht zwei solche Elemente hintereinander
   $[\#(u,v)\#(u',v') \mathrel{\widehat{=}} \#(u+u',v+v')]$.
3. Für $\#(u,v)$ muss gelten: $1 \leq u \leq v$ (Fall $u = 0$ aufwändiger!)

Implementierung

Für die Zeichenmasken gilt: Für das Element $\#(u, v)$ werden für jedes Zeichen aus dem Alphabet genau v `1`en hinzugefügt.

Beispiel mit $P = \texttt{bba\#(1,3)a}$ und $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$:

	`a###abb`
$mask^a$	`1111100`
$mask^b$	`0111011`
$mask^c$	`0111000`

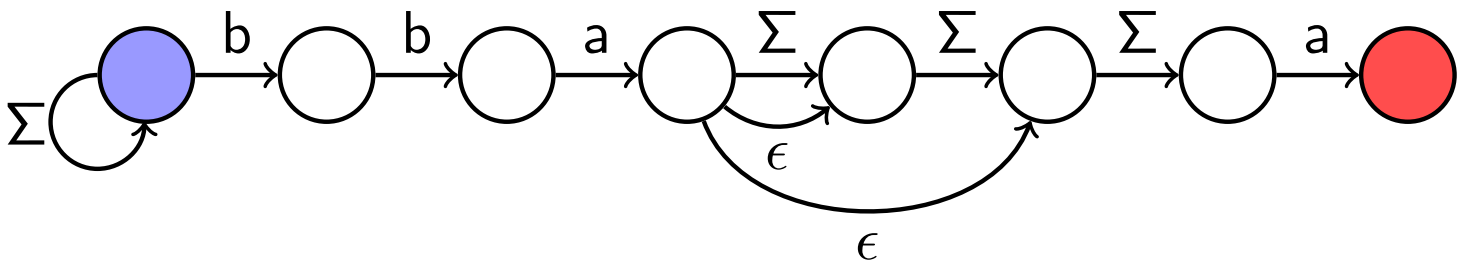


Implementierung

- Bitmaske I enthält Zustände, von denen ϵ-Kanten ausgehen.
- Bitmaske F enthält Zustände **hinter** dem Ziel der letzten ϵ-Kante jedes Elements

	a###abb
F	0100000
I	0000100

	a###abb
$mask^a$	1111100
$mask^b$	0111011
$mask^c$	0111000



Implementierung

- Um festzustellen, ob I-Zustand aktiv ist befindet, wird I mit der aktive Zustandsmenge verundet: $A \,\&\, I$.
- Durch Subtraktion $F - (A \,\&\, I)$ werden alle Zustände, die mit ϵ-Kanten von aktiven I-Zuständen erreicht werden, aktiviert.

F	0100000
$A \,\&\, I$	0000100
$-$	0011100

Implementierung

UA RUHR

- Um festzustellen, ob I-Zustand aktiv ist befindet, wird I mit der aktive Zustandsmenge verundet: $A \,\&\, I$.
- Durch Subtraktion $F - (A \,\&\, I)$ werden alle Zustände, die mit ϵ-Kanten von aktiven I-Zuständen erreicht werden, aktiviert.

F	0100000
$A \,\&\, I$	0000100
$-$	0011100

- Problem, wenn ein I-Zustand nicht aktiv ist:
  Der zugehörige F-Zustand bleibt gesetzt.

F	010000100000
$A \,\&\, I$	000000000100
$-$	010000011100

Implementierung

- Lösung: Durch Negation von F wird eine Maske erzeugt, mit der wir unerwünschte F-Bits löschen

F	010000100000
$A \,\&\, I$	000000000100
$F - (A \,\&\, I)$	010000011100

Implementierung

- Lösung: Durch Negation von F wird eine Maske erzeugt, mit der wir unerwünschte F-Bits löschen

F	010000100000
$A \,\&\, I$	000000000100
$F - (A \,\&\, I)$	010000011100
$\sim F$	101111011111
$(F - (A \,\&\, I)) \,\&\, \sim F$	000000011100

Implementierung

- Lösung: Durch Negation von F wird eine Maske erzeugt, mit der wir unerwünschte F-Bits löschen

F	010000100000
$A \,\&\, I$	000000000100
$F - (A \,\&\, I)$	010000011100
$\sim F$	101111011111
$(F - (A \,\&\, I)) \,\&\, \sim F$	000000011100

- Wegen der Einschränkung, dass zwei Elemente nicht unmittelbar aufeinander folgen, ist kein F-Zustand gleichzeitig I-Zustand des nächsten Elements.

Implementierung

UA RUHR

Erweiterte Zustandsaktualisierung des Shift-And-Algorithmus:

1. Herkömmliche Shift-And-Update-Funktion anwenden:

```
A = ((A << 1) | 1) & mask[c]
```

2. Zustandsvektor um neue aktive Zustände erweitern:

```
A = A | ((F - (A & I)) & ~F)
```

Beispiel

Sei $P = \texttt{bba\#(1,3)a}$ und $T = \texttt{bbacca}$:

	a###abb
$mask^a$	1111100
$mask^b$	0111011
$mask^c$	0111000

F	0100000
$(A \,\&\, I)$	0000000
–	0000000

$$A = 0000000$$

Beispiel

Sei $P = \texttt{bba\#(1,3)a}$ und $T = \texttt{bbacca}$:

	a###abb
$mask^a$	1111100
$mask^b$	0111011
$mask^c$	0111000

F	0100000
$(A \,\&\, I)$	0000000
–	0000000

Update 1 mit b: $A = 0000001$

Beispiel

Sei $P = \texttt{bba\#(1,3)a}$ und $T = \texttt{bbacca}$:

	a###abb
$mask^a$	1111100
$mask^b$	0111011
$mask^c$	0111000

F	0100000
$(A \,\&\, I)$	0000000
$-$	0000000

Update 2: $A = \texttt{0000001}$

Beispiel

Sei $P = \texttt{bba\#(1,3)a}$ und $T = \texttt{bbacca}$:

	a###abb
$mask^a$	1111100
$mask^b$	0111011
$mask^c$	0111000

F	0100000
$(A \,\&\, I)$	0000000
–	0000000

Update 1 mit b: $A = 0000011$

Beispiel

Sei $P = \texttt{bba\#(1,3)a}$ und $T = \texttt{bbacca}$:

	a###abb
$mask^a$	1111100
$mask^b$	0111011
$mask^c$	0111000

F	0100000
$(A \,\&\, I)$	0000000
–	0000000

Update 2: $A = \texttt{0000011}$

Beispiel

Sei $P = \texttt{bba\#(1,3)a}$ und $T = \texttt{bbacca}$:

	a###abb
$mask^a$	1111100
$mask^b$	0111011
$mask^c$	0111000

F	0100000
$(A \,\&\, I)$	0000000
–	0000000

Update 1 mit a: $A = 0000100$

Beispiel

Sei $P = \texttt{bba\#(1,3)a}$ und $T = \texttt{bbacca}$:

	a###abb
$mask^a$	1111100
$mask^b$	0111011
$mask^c$	0111000

F	0100000
$(A \,\&\, I)$	0000100
–	0011100

Update 2: $A = \texttt{0011100}$

Beispiel

Sei $P = \texttt{bba\#(1,3)a}$ und $T = \texttt{bbacca}$:

	a###abb
$mask^a$	1111100
$mask^b$	0111011
$mask^c$	0111000

F	0100000
$(A \,\&\, I)$	0000000
–	0000000

Update 1 mit c: $A = 0111000$

Beispiel

Sei $P = \texttt{bba\#(1,3)a}$ und $T = \texttt{bbacca}$:

	a###abb
$mask^a$	1111100
$mask^b$	0111011
$mask^c$	0111000

F	0100000
$(A \,\&\, I)$	0000000
–	0000000

Update 2: $A = \texttt{0111000}$

Beispiel

Sei $P = \texttt{bba\#(1,3)a}$ und $T = \texttt{bbacca}$:

	a###abb
$mask^a$	1111100
$mask^b$	0111011
$mask^c$	0111000

F	0100000
$(A \,\&\, I)$	0000000
–	0000000

Update 1 mit c: $A = 0110000$

Beispiel

Sei $P = \texttt{bba\#(1,3)a}$ und $T = \texttt{bbacca}$:

	a###abb
$mask^a$	1111100
$mask^b$	0111011
$mask^c$	0111000

F	0100000
$(A \,\&\, I)$	0000000
–	0000000

Update 2: $A = \texttt{0110000}$

Beispiel

Sei $P =$ `bba#(1,3)a` und $T =$ `bbacca`:

	`a###abb`
$mask^a$	`1111100`
$mask^b$	`0111011`
$mask^c$	`0111000`

F	`0100000`
$(A \,\&\, I)$	`0000000`
–	`0000000`

Update 1 mit `a`: $A =$ `1100000`

Beispiel

Sei $P = \texttt{bba\#(1,3)a}$ und $T = \texttt{bbacca}$:

	a###abb
$mask^a$	1111100
$mask^b$	0111011
$mask^c$	0111000

F	0100000
$(A \,\&\, I)$	0000000
–	0000000

Update 2: $A = \texttt{1100000}$ Treffer

Optionale Zeichen

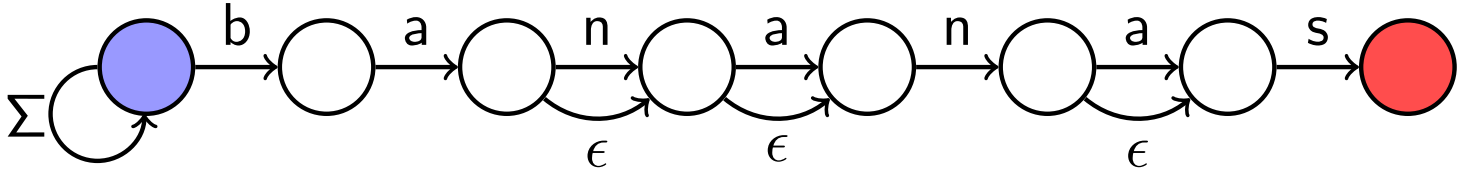
UA RUHR

- Mit einer zusätzlichen Erweiterung können optionale Zeichen verarbeitet werden, entspricht `#(0,1);`
  (bisher waren 0 Vorkommen nicht möglich!)
- Notation: `?` hineter dem optionalen Zeichen
- $P = \{\text{color, colour}\} \mapsto P = \texttt{colou?r}$.
- Konsekutive ϵ-Transitionen (Blöcke) sind erlaubt.
- Auch diese Erweiterung lässt sich mit dem Shift-And-Algorithmus realisieren.

Optionale Zeichen

Implementierung: drei zusätzliche Bitasken
I: Blockbeginn; F: Blockende; O: Ziele von ϵ-Kanten

Beispiel: $P =$ `ban?a?na?s`

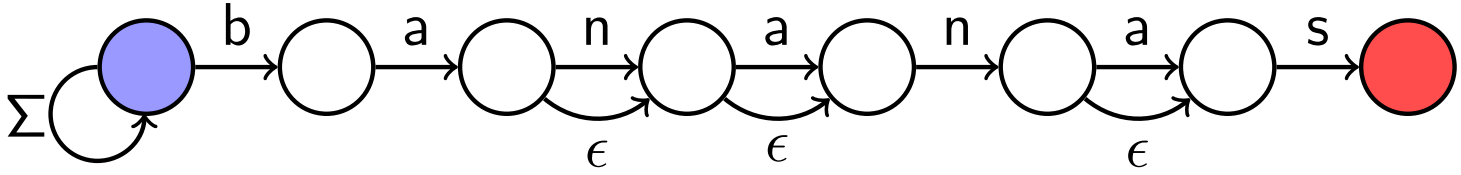


I :	0	1	0	0	1	0	0
F :	0	0	0	1	0	1	0
O :	0	0	1	1	0	1	0

Optionale Zeichen

Implementierung: drei zusätzliche Bitasken
I: Blockbeginn; F: Blockende; O: Ziele von ϵ-Kanten

Beispiel: $P =$ `ban?a?na?s`



I :	0	1	0	0	1	0	0
F :	0	0	0	1	0	1	0
O :	0	0	1	1	0	1	0

Sobald ein Zustand innerhalb eines Blocks aktiv ist,
müssen seine Folgezustände im Block aktiviert werden.

Implementierung optionaler Zeichen

- Propagierung mit Subtraktion

```
  1101010000
-            1
------------
  1101001111
```

```
  1101011000
-          100
------------
  1101010100
```

Implementierung optionaler Zeichen

- Propagierung mit Subtraktion

```
    1101010000          1101011000
 -          1       -        100
------------------  ------------------
    1101001111          1101010100
```

- Propagierung vom niederwertigsten aktiven Bit innerhalb eines Blocks bis zum *F*-Bit

```
A   .0010100.                    A   .0010100.
I   .0000001.                  A|F   .1010100.
O   .1111110.               (A|F)-I   .1010011.
F   .1000000.        (A|F-I)=(A|F)   .1111000.
                   O&(A|F-I)=(A|F)   .1111000.
>>  .1111100.  A|(O&(A|F-I)=(A|F))   .1111100.
```

Implementierung optionaler Zeichen

- Propagierung mit Subtraktion

```
    1101010000          1101011000
  -          1        -        100
------------------  ------------------
    1101001111          1101010100
```

- Propagierung vom niederwertigsten aktiven Bit innerhalb eines Blocks bis zum F-Bit

```
A   .0010100.                    A   .0010100.
I   .0000001.                  A|F   .1010100.
O   .1111110.               (A|F)-I   .1010011.
F   .1000000.       (A|F-I)=(A|F)    .1111000.
                  O&(A|F-I)=(A|F)    .1111000.
>>  .1111100.  A|(O&(A|F-I)=(A|F))   .1111100.
```

- Bitweise Gleichheit $X = Y$ implementiert als $\sim(X \oplus Y)$.

Implementierung optionaler Zeichen

UA RUHR

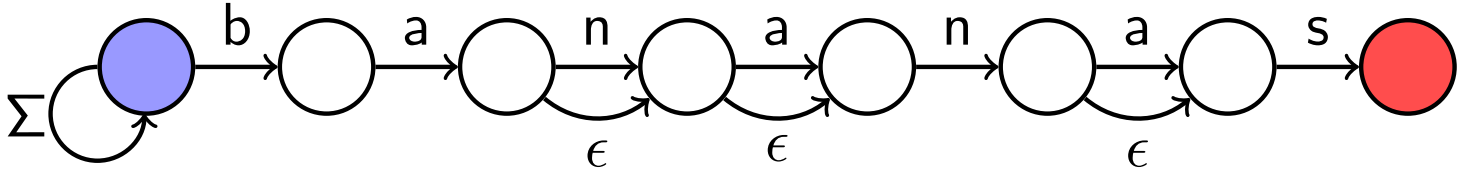
Implementierung:

1. Masken für alle Zeichen aus Σ erstellen, dabei die optionalen Zeichen wie reguläre Zeichen behandeln.
2. Herkömmliche Shift-And-Update-Funktion anwenden:
   ```
   A = ((A << 1) | 1) & mask[c]
   ```
3. Zustandsvektor um neue aktive Zustände erweitern:

   ```
   A_f = A | F
   A = A | (O & (~(A_f - I) ^ A_f))
   ```

   (Hier ist ^ die xor-Operation.)

Beispiel



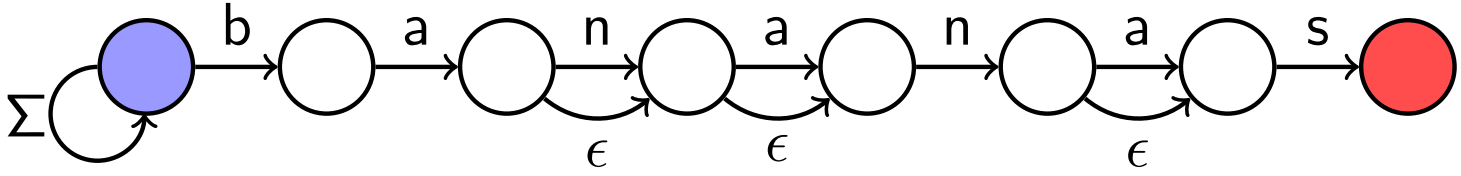
Sei $P = \texttt{ban?a?na?s}$ und $T = \texttt{banns}$:

	sananab
$mask^a$	0101010
$mask^b$	0000001
$mask^n$	0010100
$mask^s$	1000000

I	0010010
F	0101000
O	0101100
$o \,\&\, (\sim(A_f - I) \oplus A_f)$	0000000

$$A = 0000000$$

Beispiel



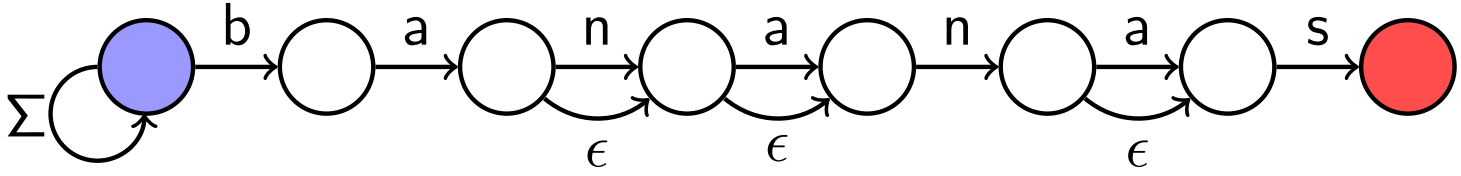
Sei $P =$ `ban?a?na?s` und $T =$ `banns`:

	`sananab`
$mask^a$	`0101010`
$mask^b$	`0000001`
$mask^n$	`0010100`
$mask^s$	`1000000`

I	`0010010`
F	`0101000`
O	`0101100`
$o \,\&\, (\sim(A_f - I) \oplus A_f)$	`0000000`

Update 1 mit `b`: $A =$ `0000001`

Beispiel



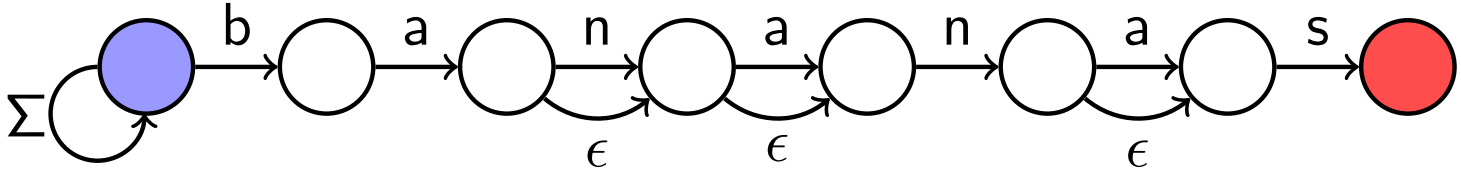
Sei $P =$ `ban?a?na?s` und $T =$ `banns`:

	`sananab`
$mask^a$	`0101010`
$mask^b$	`0000001`
$mask^n$	`0010100`
$mask^s$	`1000000`

I	`0010010`
F	`0101000`
O	`0101100`
$o \,\&\, (\sim(A_f - I) \oplus A_f)$	`0000000`

Update 2: $A =$ `0000001`

Beispiel



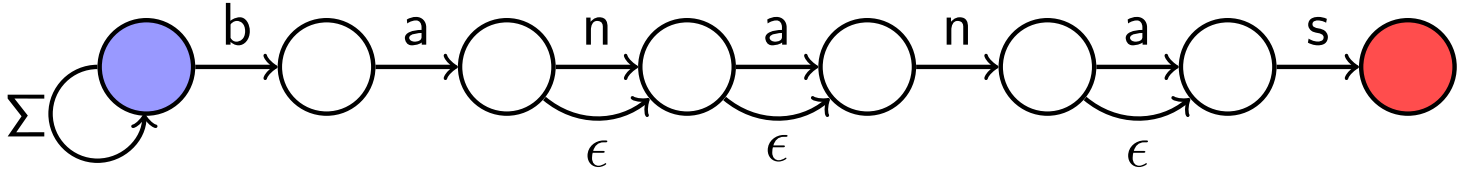
Sei $P =$ `ban?a?na?s` und $T =$ `banns`:

	`sananab`
$mask^a$	`0101010`
$mask^b$	`0000001`
$mask^n$	`0010100`
$mask^s$	`1000000`

I	`0010010`
F	`0101000`
O	`0101100`
$o \,\&\, (\sim(A_f - I) \oplus A_f)$	`0000000`

Update 1 mit `a`: $A =$ `0000010`

Beispiel



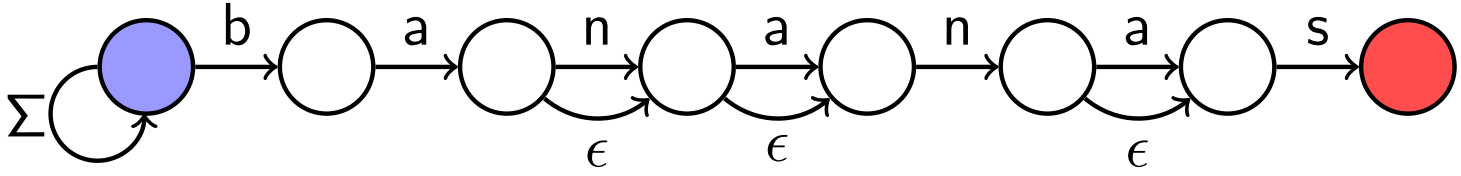
Sei $P =$ `ban?a?na?s` und $T =$ `banns`:

	`sananab`
$mask^a$	`0101010`
$mask^b$	`0000001`
$mask^n$	`0010100`
$mask^s$	`1000000`

I	`0010010`
F	`0101000`
O	`0101100`
$o \,\&\, (\sim(A_f - I) \oplus A_f)$	`0001100`

Update 2: $A =$ `0001110`

Beispiel



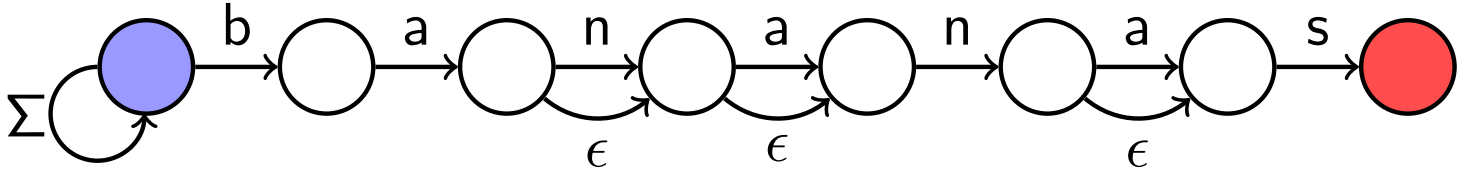
Sei $P =$ `ban?a?na?s` und $T =$ `banns`:

	`sananab`
$mask^a$	`0101010`
$mask^b$	`0000001`
$mask^n$	`0010100`
$mask^s$	`1000000`

I	`0010010`
F	`0101000`
O	`0101100`
$o \,\&\, (\sim(A_f - I) \oplus A_f)$	`0000000`

Update 1 mit `n`: $A =$ `0010100`

Beispiel



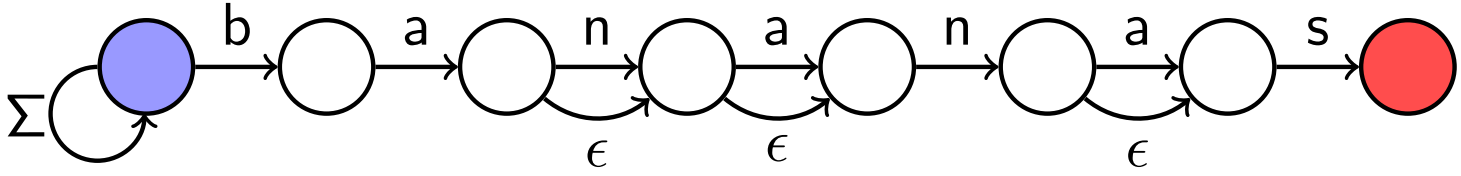
Sei $P =$ `ban?a?na?s` und $T =$ `banns`:

	`sananab`
$mask^a$	`0101010`
$mask^b$	`0000001`
$mask^n$	`0010100`
$mask^s$	`1000000`

I	`0010010`
F	`0101000`
O	`0101100`
$o \,\&\, (\sim(A_f - I) \oplus A_f)$	`0101100`

Update 2: $A =$ `0111100`

Beispiel



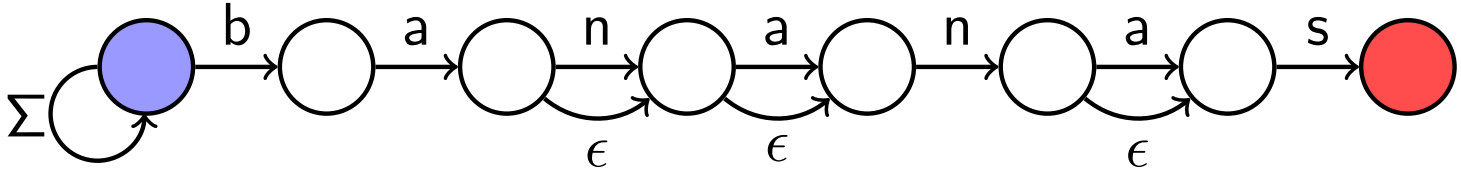
Sei $P =$ `ban?a?na?s` und $T =$ `bannns`:

	`sananab`
$mask^a$	`0101010`
$mask^b$	`0000001`
$mask^n$	`0010100`
$mask^s$	`1000000`

I	`0010010`
F	`0101000`
O	`0101100`
$o \,\&\, (\sim(A_f - I) \oplus A_f)$	`0000000`

Update 1 mit `n`: $A =$ `0010000`

Beispiel



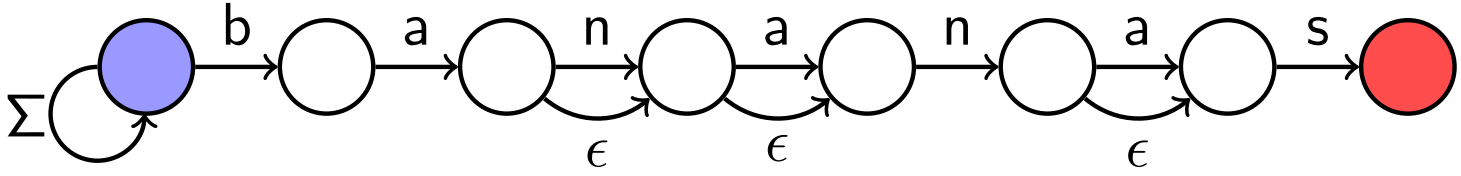
Sei $P =$ `ban?a?na?s` und $T =$ `bannns`:

	`sananab`
$mask^a$	`0101010`
$mask^b$	`0000001`
$mask^n$	`0010100`
$mask^s$	`1000000`

I	`0010010`
F	`0101000`
O	`0101100`
$o \,\&\, (\sim(A_f - I) \oplus A_f)$	`0100000`

Update 2: $A =$ `0110000`

Beispiel



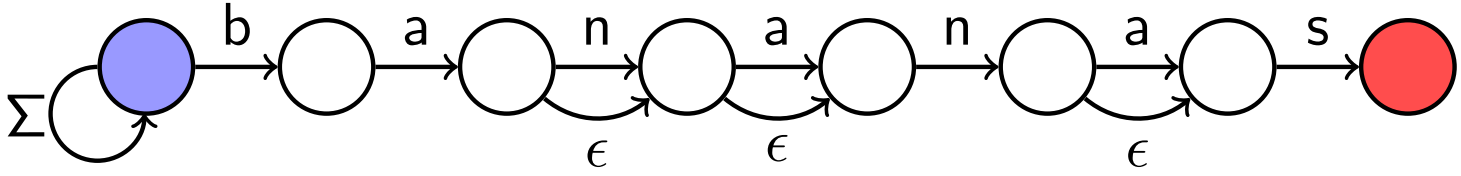
Sei $P =$ `ban?a?na?s` und $T =$ `bannns`:

	`sananab`
$mask^a$	`0101010`
$mask^b$	`0000001`
$mask^n$	`0010100`
$mask^s$	`1000000`

I	`0010010`
F	`0101000`
O	`0101100`
$o \,\&\, (\sim(A_f - I) \oplus A_f)$	`0000000`

Update 1 mit `s`: $A =$ `1000000`

Beispiel



Sei $P =$ `ban?a?na?s` und $T =$ `bannns`:

	`sananab`
$mask^a$	`0101010`
$mask^b$	`0000001`
$mask^n$	`0010100`
$mask^s$	`1000000`

I	`0010010`
F	`0101000`
O	`0101100`
$o \,\&\, (\sim(A_f - I) \oplus A_f)$	`0000000`

Update 2: $A =$ `1000000` Treffer

Zusammenfassung

UA RUHR

- Durch Modifikationen lässt sich der Shift-And-Algorithmus auf flexiblere Muster erweitern.
- Verallgemeinerte Strings, beliebige Zeichenfolgen beschränkter Länge und optionale Zeichen sind möglich.
- Laufzeiten ändern sich nicht.