UNIVERSITÄT DES SAARLANDES

ZBI ZENTRUM FÜR BIOINFORMATIK

Exact Pattern Matching with Bit-Parallel Algorithms

Algorithms for Sequence Analysis

Sven Rahmann

Summer 2021

Overview

Previous Lecture

Exact Pattern Matching (for single patterns without index)

- Reminder: NFAs and DFAs
- DFA-based Knuth-Morris-Pratt algorithm (lps table)
- Bit-parallel simulation of NFA: Shift-And algorithm

Overview

Previous Lecture

Exact Pattern Matching (for single patterns without index)

- Reminder: NFAs and DFAs
- DFA-based Knuth-Morris-Pratt algorithm (lps table)
- Bit-parallel simulation of NFA: Shift-And algorithm

Today's Lecture

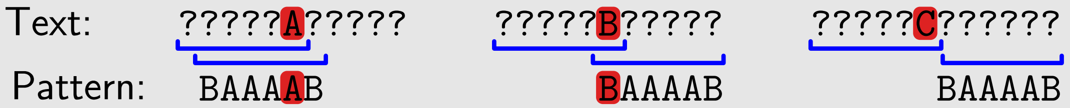
More on bit-parallel algorithms:

- How to get longer shifts than Horspool's algorithm?
  $\rightarrow$ **BNDM algorithm** (backward non-deterministic DAWG matching)
- Bit-parallel algorithms for more general patterns

A Substring-based Algorithm: BNDM

Reminder: Horspool Algorithm

Horspool shift function



Approach

- Compare characters **from right to left** in current window
- Shift window based on last character

Problem

Small alphabet (most likely) leads to short shifts (especially bad for long patterns).

Substring-based Shift Function

Ideas

- Read from right to left (like Horspool)
- **Read on** after mismatch to achieve longer shifts
- When **substring** of window is not **substring** of pattern, window can be shifted beyond that **substring**.
- Keeping track of **suffixes of window** that are **prefixes of pattern** can further increase shifts

Sought: Data Structure

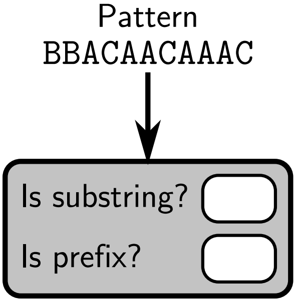
Requirements / supported queries

- Add characters from right to left
- Is part read so far a **substring** of the pattern?
- Is part read so far a **prefix** of the pattern?

Sought: Data Structure

Requirements / supported queries

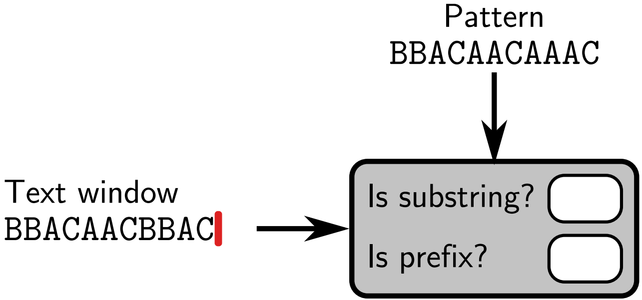
- Add characters from right to left
- Is part read so far a **substring** of the pattern?
- Is part read so far a **prefix** of the pattern?



Sought: Data Structure

Requirements / supported queries

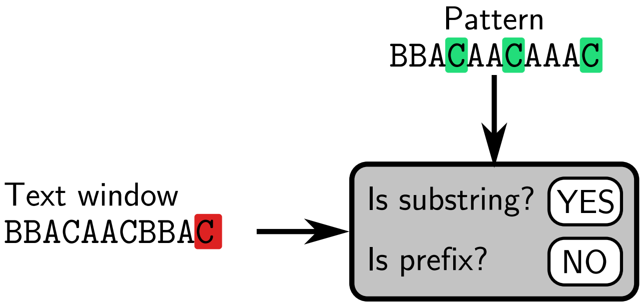
- Add characters from right to left
- Is part read so far a **substring** of the pattern?
- Is part read so far a **prefix** of the pattern?



Sought: Data Structure

Requirements / supported queries

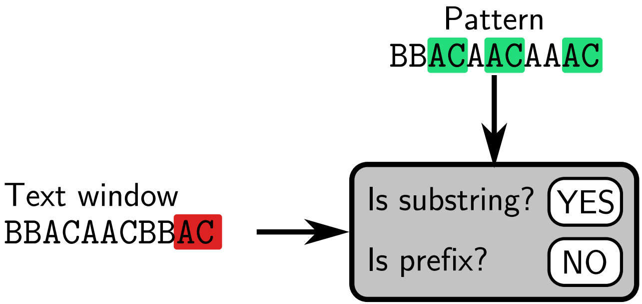
- Add characters from right to left
- Is part read so far a **substring** of the pattern?
- Is part read so far a **prefix** of the pattern?



Sought: Data Structure

Requirements / supported queries

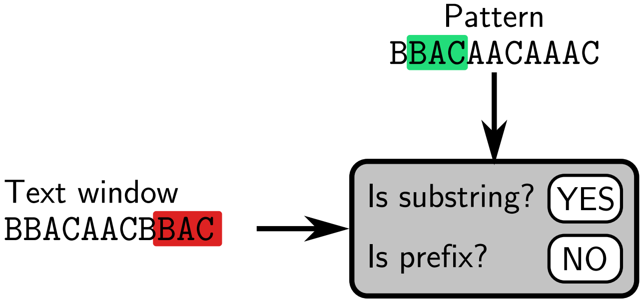
- Add characters from right to left
- Is part read so far a **substring** of the pattern?
- Is part read so far a **prefix** of the pattern?



Sought: Data Structure

Requirements / supported queries

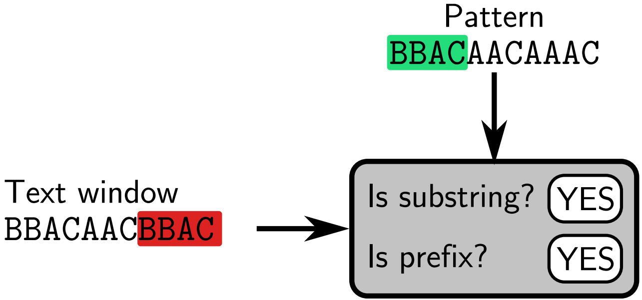
- Add characters from right to left
- Is part read so far a **substring** of the pattern?
- Is part read so far a **prefix** of the pattern?



Sought: Data Structure

Requirements / supported queries

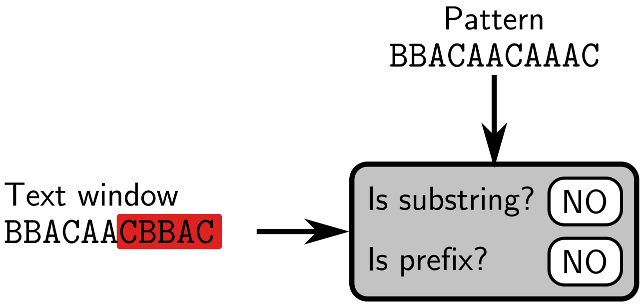
- Add characters from right to left
- Is part read so far a **substring** of the pattern?
- Is part read so far a **prefix** of the pattern?



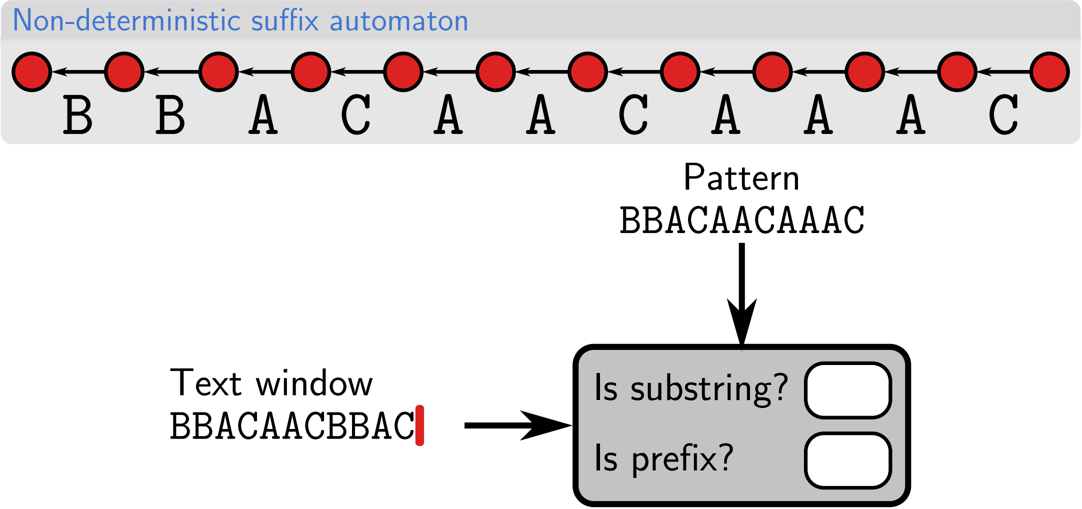
Sought: Data Structure

Requirements / supported queries

- Add characters from right to left
- Is part read so far a **substring** of the pattern?
- Is part read so far a **prefix** of the pattern?



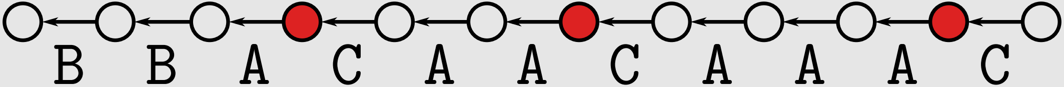
Solution

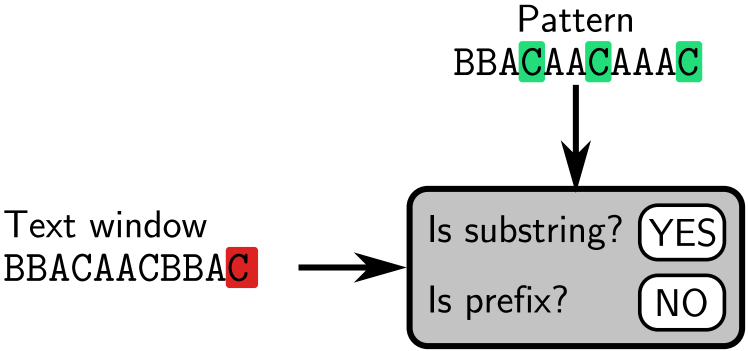


Non-deterministic suffix automaton
B B A C A A C A A A C
Pattern
BBACAACAAAC
Text window
BBACAACBBAC
Is substring?
Is prefix?

Solution

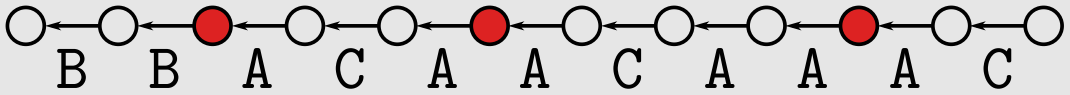
Non-deterministic suffix automaton

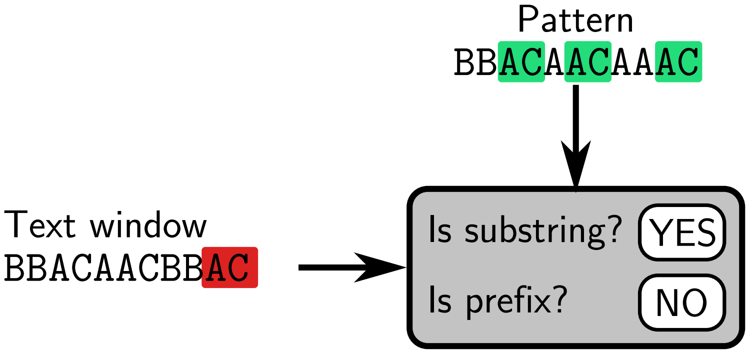




Solution

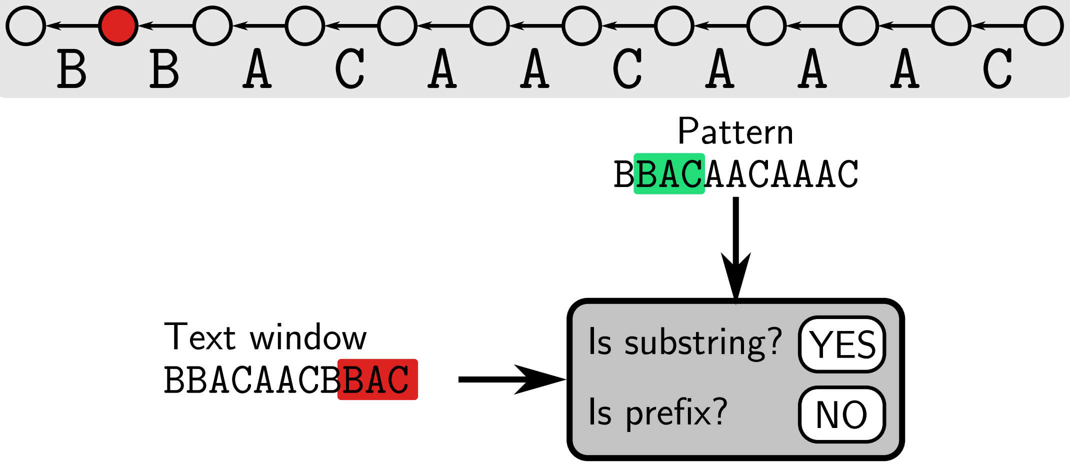
Non-deterministic suffix automaton





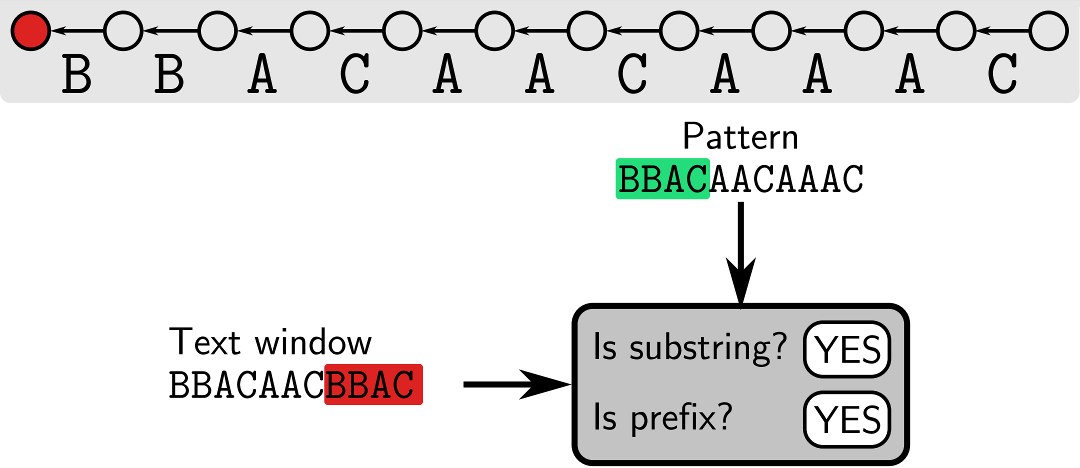
Solution

Non-deterministic suffix automaton



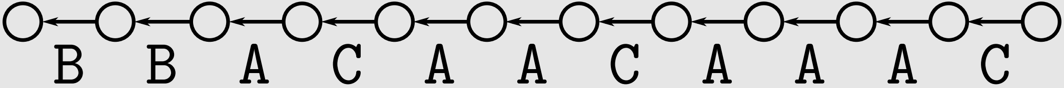
Solution

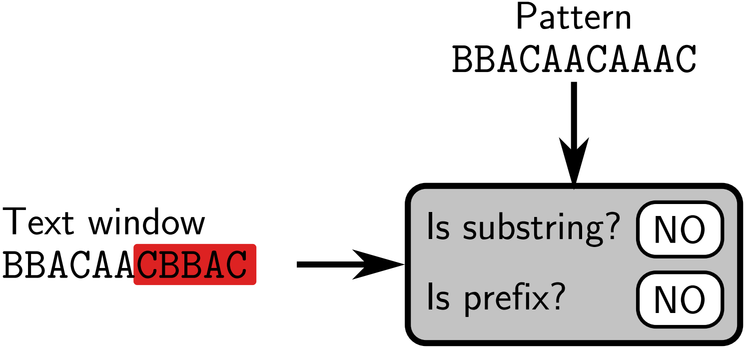
Non-deterministic suffix automaton



Solution

Non-deterministic suffix automaton





Non-Deterministic Suffix Automaton

Construction

- Construct pattern matching NFA of **reverse pattern**
- **All** states are **start states**

Usage

- Use Shift-And approach to maintain set of active states
- **Any** state active $\Leftrightarrow$ substring occurs in pattern
- **Accept** state active $\Leftrightarrow$ found prefix

BNDM Algorithm

BNDM Algorithm Outline

For **each window**:

1. **Initialize** suffix automaton (all states active)
2. Read window from **right to left** until no states active or full window read.
3. Keep track of **longest window suffix** that is **pattern prefix**
4. **Shift** window to **align** this suffix with pattern prefix

BNDM Algorithm: Code

```
def BNDM(P, T):
  masks, accept_state = compute_masks(P[::-1])
  n, m, pos = len(T), len(P), len(P)
  while pos <= n:
    A = (1 << m) - 1  # initialize: all bits on
    j, lastsuffix = 1, 0
    while A != 0:
      A &= masks[T[pos-j]]       # update (AND)
      if A & accept_state != 0: # accept state?
        if j == m:              # full pattern found?
          yield (pos - m, pos)
          break
        else:                   # found proper prefix
          lastsuffix = j        # store suffix
      j += 1
      A = A << 1                # update (shift)
    pos += m - lastsuffix # shift window
```

Deterministic Counterpart: BDM

BDM Algorithm

- As before, we could turn NFA into DFA
  $\rightarrow$ **deterministic suffix automaton** (=DAWG)
- Either use subset construction (can be inefficient) or use complicated techniques (or keep using BNDM)

Names

- **BDM** = Backward deterministic DAWG Matching,
- **BNDM** = Backward Non-deterministic DAWG Matching,
- **DAWG** = Directed Acyclic Word Graph.

Bit-Parallel Algorithms for Extended Patterns

Overview

So far, patterns were simple strings, $P \in \Sigma^*$.

For several applications (e.g., transcription factor binding sites on DNA), it is necessary to consider patterns that allow

- different characters (some subset of Σ) at some positions,
- variable-length runs of arbitrary characters,
- optional characters at some positions.

Overview

So far, patterns were simple strings, $P \in \Sigma^*$.

For several applications (e.g., transcription factor binding sites on DNA), it is necessary to consider patterns that allow

- different characters (some subset of Σ) at some positions,
- variable-length runs of arbitrary characters,
- optional characters at some positions.

All of these patterns are subsets of **regular expressions**, which are recognized by DFAs.

Overview

So far, patterns were simple strings, $P \in \Sigma^*$.

For several applications (e.g., transcription factor binding sites on DNA), it is necessary to consider patterns that allow

- different characters (some subset of Σ) at some positions,
- variable-length runs of arbitrary characters,
- optional characters at some positions.

All of these patterns are subsets of **regular expressions**, which are recognized by DFAs.

However, specialized bit-parallel implementations for each pattern class are more efficient.
All of the above patterns can be recognized by variations of the Shift-And algorithm.

Generalized Strings

- A **generalized string** over Σ is a string over $2^{\Sigma} \setminus \{\emptyset\}$, i.e., a string whose characters are non-empty subsets of Σ.

Generalized Strings

- A **generalized string** over Σ is a string over $2^{\Sigma} \setminus \{ \emptyset \}$, i.e., a string whose characters are non-empty subsets of Σ.
- **Example:** Consider the set { Maier, Meier, Meier, Meyer }. It can be written as a single generalized string: { M }{ a,e }{ i,y }{ e }{ r }. Shorthand: M[ae][iy]er

Generalized Strings

- A **generalized string** over Σ is a string over $2^{\Sigma} \setminus \{\emptyset\}$, i.e., a string whose characters are non-empty subsets of Σ.
- **Example:** Consider the set { Maier, Meier, Meier, Meyer }. It can be written as a single generalized string: { M }{ a,e }{ i,y }{ e }{ r }. Shorthand: M[ae][iy]er
- **Notation:** Singleton sets are represented by their unique element. Larger sets are represented by square brackets: [ae] for { a,e }. We write # for $\Sigma \in 2^{\Sigma}$ ("any charachter").

Generalized Strings

- A **generalized string** over Σ is a string over $2^\Sigma \setminus \{\emptyset\}$,
  i.e., a string whose characters are non-empty subsets of Σ.
- **Example:** Consider the set { Maier, Meier, Meier, Meyer }.
  It can be written as a single generalized string: { M } { a,e } { i,y } { e } { r }.
  Shorthand: M[ae][iy]er
- **Notation:** Singleton sets are represented by their unique element.
  Larger sets are represented by square brackets: [ae] for { a,e }.
  We write # for $\Sigma \in 2^\Sigma$ ("any charachter").
- In DNA sequences, the IUPAC code specifies a one-letter code for each subset:
  size 1: `ACGT`; size 2: `SWRYKM`; size 3: `BDHV`; size 4: `N`.

The Shift-And Algorithm for Generalized Strings

- Recall the Shift-And algorithm with active state bits D:
  $D \leftarrow \big((D \ll 1) \mid 1\big) \,\&\, \text{mask}(c)$
- The Shift-And algorithm can process generalized strings without modifications.
- The bit masks simply tell which characters are allowed at which position.
  It is no problem that more than one bit is set at some positions.

The Shift-And Algorithm for Generalized Strings

- Recall the Shift-And algorithm with active state bits D:
  $D \leftarrow \big((D \ll 1) \mid 1\big) \ \& \ \mathrm{mask}(c)$
- The Shift-And algorithm can process generalized strings without modifications.
- The bit masks simply tell which characters are allowed at which position.
  It is no problem that more than one bit is set at some positions.
- **Example:** $P = \texttt{abba\#b}$ over $\Sigma = \{\texttt{a}, \texttt{b}\}$.

	b#abba (reversed because of bit numbers)
mask[a]	011001
mask[b]	110110

The Shift-And Algorithm for Generalized Strings

- Recall the Shift-And algorithm with active state bits D:
  $D \leftarrow \big((D \ll 1) \mid 1\big) \;\&\; \mathrm{mask}(c)$
- The Shift-And algorithm can process generalized strings without modifications.
- The bit masks simply tell which characters are allowed at which position.
  It is no problem that more than one bit is set at some positions.
- **Example:** $P =$ `abba#b` over $\Sigma = \{\mathtt{a}, \mathtt{b}\}$.

	`b#abba` (reversed because of bit numbers)
mask[a]	`011001`
mask[b]	`110110`

(That was too easy, so let's try more complex patterns...)

Bounded-length Runs of Arbitrary Characters

- A **run of arbitrary characters** is a sequence of Σs (written as $\#$s) in a generalized string.
  We allow **variable run lengths**, but with fixed **lower and upper bounds**.
- **Notation:** `#`(L, U) with lower bound L and upper bound U
- **Example:** $P =$ `bba#(1,3)a`:
  After `bba`, we have one to three arbitrary characters, followed by `a`.

Bounded-length Runs of Arbitrary Characters

- A **run of arbitrary characters** is a sequence of Σs (written as $\#$s) in a generalized string.
  We allow **variable run lengths**, but with fixed **lower and upper bounds**.
- **Notation:** $\#(L, U)$ with lower bound L and upper bound U
- **Example:** $P =$ `bba#(1,3)a`:
  After `bba`, we have one to three arbitrary characters, followed by `a`.
- Three restrictions:
  1. An element $\#(L, U)$ does not appear first or last in the pattern.
     (We could remove them without substantially changing the pattern.)
  2. No two such elements appear next to each other.
     (No problem, just add them: $\#(L, U)\#(L', U') \mathrel{\widehat{=}} \#(L + L', U + U')$.)
  3. We require $1 \leq L \leq U$.
     (Allowing $L = 0$ is technically more challenging!)

An NFA for Bounded-length Runs of Arbitrary Characters

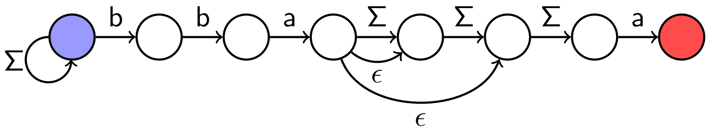
- Before considering a bit-parallel implementation, we design an NFA.
- We need ϵ-transitions, an extension of the standard NFA definition: ϵ-transitions happen instantaneously, without consuming a character.

An NFA for Bounded-length Runs of Arbitrary Characters

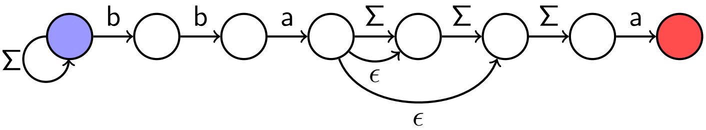
- Before considering a bit-parallel implementation, we design an NFA.
- We need ϵ-transitions, an extension of the standard NFA definition: ϵ-transitions happen instantaneously, without consuming a character.
- The ϵ-transitions allow us to skip the optional characters. For technical reasons, they **exit the initial state** of the run; the **first** #s in each run are optional. (One could do it differently, but that would be harder to implement!)

An NFA for Bounded-length Runs of Arbitrary Characters

- Before considering a bit-parallel implementation, we design an NFA.
- We need ϵ-transitions, an extension of the standard NFA definition: ϵ-transitions happen instantaneously, without consuming a character.
- The ϵ-transitions allow us to skip the optional characters. For technical reasons, they **exit the initial state** of the run; the **first** #s in each run are optional. (One could do it differently, but that would be harder to implement!)
- **Example:** $P =$ `bba#(1,3)a`:

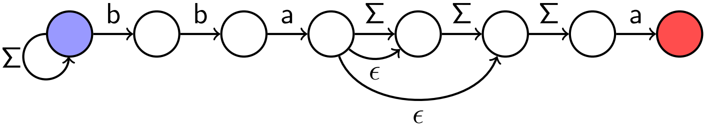


Bit-parallel Implementation



- We use the Shift-And algorithm on the maximal-length pattern as a basis. Then we additionally need to implement the ϵ-transitions.
- Masks are constructed as before (for #: 1-bits for each character).

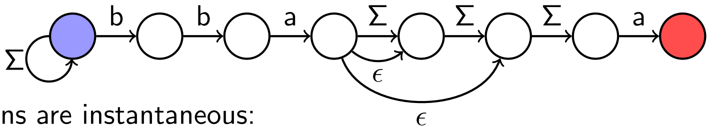
Bit-parallel Implementation



- We use the Shift-And algorithm on the maximal-length pattern as a basis. Then we additionally need to implement the ϵ-transitions.
- Masks are constructed as before (for #: 1-bits for each character).
- **Example:** $P =$ `bba#(1,3)a` with $\Sigma = \{a, b, c\}$:

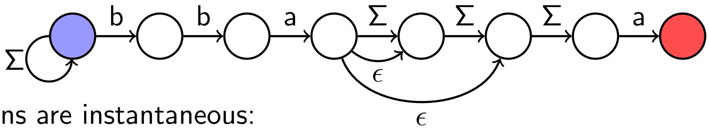
	a###abb
mask[a]	1111100
mask[b]	0111011
mask[c]	0111000

Implementation of ϵ-Transitions



- ϵ-transitions are instantaneous:
  Whenever a state with outgoing ϵ-transitions becomes active (1-bit), this is immediately propagated to the targets of the outgoing ϵ-edges; these are by construction adjacent to the source state.

Implementation of ϵ-Transitions



- ϵ-transitions are instantaneous:
  Whenever a state with outgoing ϵ-transitions becomes active (1-bit), this is immediately propagated to the targets of the outgoing ϵ-edges; these are by construction adjacent to the source state.
- The actual propagation of 1-bits will be achieved by subtraction (next slide).
- We use two additional bit masks:
  - Bit mask I marks states with outgoing ϵ-transitions.
  - Bit mask F marks the state after the target of the last ϵ-transition of each run.

	a###abb
F	0100000
I	0000100

Propagation of Ones

- Let A be the bit mask of active states. Then $A \,\&\, I$ selects active I-states.
- Subtraction $F - (A \,\&\, I)$ propagates 1-bits, zeroes F-bit

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

Propagation of Ones

- Let A be the bit mask of active states. Then $A \,\&\, I$ selects active I-states.
- Subtraction $F - (A \,\&\, I)$ propagates 1-bits, zeroes F-bit

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

- Problem: Inactive I-states keep corresponding F-bit set:

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

Propagation of Ones (Continued)

- Solution: Zero out F-bits by a bitwise `and` with the negation of F:

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

Propagation of Ones (Continued)

- Solution: Zero out F-bits by a bitwise `and` with the negation of F:

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

Propagation of Ones (Continued)

- Solution: Zero out F-bits by a bitwise `and` with the negation of F:

F	010000100000
$A \mathbin{\&} I$	000000000100
$F - (A \mathbin{\&} I)$	010000011100
$\sim F$	101111011111
$(F - (A \mathbin{\&} I)) \mathbin{\&} \sim F$	000000011100

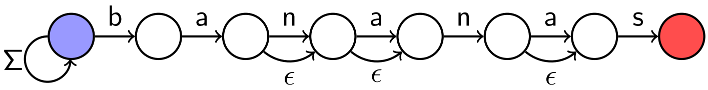
- The resulting modified Shift-And algorithm is thus:
  1. Apply standard Shift-And update:
     ```
     A = ((A << 1) | 1) & mask[c]
     ```
  2. Propagate active I-states along ϵ-transitions:
     ```
     A = A | ((F - (A & I)) & ~F)
     ```

Patterns with Optional Characters

- Another modification of the Shift-And algorithm allows optional characters.
- **Notation:** Write `?` after the optional character.
- **Example:** The set $\{$color, colour$\}$ becomes $P =$ `colou?r`.
- Consecutive ϵ-transitions ("blocks") are allowed.

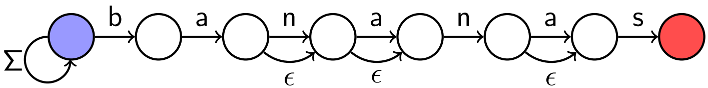
Patterns with Optional Characters

- Another modification of the Shift-And algorithm allows optional characters.
- **Notation:** Write `?` after the optional character.
- **Example:** The set {color, colour} becomes $P =$ `colou?r`.
- Consecutive ϵ-transitions ("blocks") are allowed.
- **Larger example:** $P =$ `ban?a?na?s` and $T =$ `banabanns`



Bit-Parallel Implementation of Optional Characters

- Three bit masks:
  I: block start; O: tagets of ϵ-transitions; F: block end

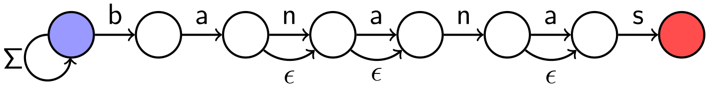


I :	0	1	0	0	1	0	0
F :	0	0	0	1	0	1	0
O :	0	0	1	1	0	1	0

- Note: actual bit patterns are reversed (bit numbering vs. state numbering)!

Bit-Parallel Implementation of Optional Characters

- Three bit masks:
  I: block start; O: tagets of ϵ-transitions; F: block end



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

- Note: actual bit patterns are reversed (bit numbering vs. state numbering)!
- Activity of any state within a block must be propagated to the block's end.

Bit-Parallel Implementation of Optional Characters (Continued)

- Activity of any state within a block must be propagated to the block's end: Propagate the lowest active bit within a block up to the F-bit.

Bit-Parallel Implementation of Optional Characters (Continued)

- Activity of any state within a block must be propagated to the block's end: Propagate the lowest active bit within a block up to the F-bit.
- Consider how 1-bit propagation via subtraction works:

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

- Bits to the left (green) and to the right (black) are unchanged; only bits between the rightmost ones in the current block change (red).

Bit-Parallel Implementation of Optional Characters (Continued)

- Activity of any state within a block must be propagated to the block's end: Propagate the lowest active bit within a block up to the *F*-bit.
- Consider how 1-bit propagation via subtraction works:

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

- Bits to the left (green) and to the right (black) are unchanged; only bits between the rightmost ones in the current block change (red).
- We develop the machinery by example:

```
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.
```

Bit-Parallel Implementation of Optional Characters (Conclusion)

- **Note:** Bitwise equality $X = Y$ can be implemented as $\sim(X \oplus Y)$.
- Full implementation:
  1. Create masks for all characters;
     treat optional characters as regular characters.
  2. Standard Shift-And update of active states A:
     ```
     A = ((A << 1) | 1) & mask[c]
     ```
  3. Propagate active states over optional characters:
     ```
     A_f = A | F
     A = A | (O & (~(A_f - I) ^ A_f))
     ```
     (Here ˆ denotes the xor-operation.)

Summary I

Topic

Bit-parallel methods for exact pattern matching of single patterns without text indexing

Properties of bit-parallel algorithms

- Typically only applicable if an "almost linear" NFA recognizes the pattern, and if this NFA has at most 64 (register width) states
- Shift-And approach is simple and very flexible, extends to general patterns; running time is always $O(n)$ for constant $|P| < 64$.
- BNDM approach is also simple and flexible; may pathologically use $O(mn)$ time even for constant $m = |P| < 64$, but has best-case running time of $O(m + n/m)$.

Summary II

Topic

Exact pattern matching of single patterns without text indexing

Strengths of different algorithms

- **Shift-And:** simple, applicable if $|P| < 64$
- **B(N)DM:** for $|P| < 64$; best case of $O(m + n/m)$; long shifts even for small alphabet + long pattern
- **Horspool:** best case of $O(m + n/m)$ for large alphabet + long pattern
- **Knuth-Morris-Pratt:** best asymptotic time of $O(m + n)$

- Automata theory was actually very useful
- Next topic: index structures (i.e. preprocessing the text)

Exam Questions

- Explain the idea of bit-parallel simulation of NFAs.
- Explain the suffix automaton and the BNDM algorithm.
- What are the advantages of BNDM over Horspool's algorithm?
- What are the advantages of BNDM over the Shift-And algorithm?
- What is a generalized string?
- How does the Shift-And algorithm change when you allow generalized strings?
- Why would you want to use the Shift-And algorithm for runs with bounded length, when the algorithms for optional characters is more general ($\#(3,5) = \#?\#?\#\#\#$)?
- How do you implement bit-parallel propagation of an active state?