Part II: DNA Motif Search

Just-in-time Compiled Python for Bioinformatics Research

Johanna Elena Schmitz, Jens Zentgraf and Sven Rahmann

ISMB 2024, Montréal, Canada (July 12, 2024)

DNA Motif Search

DNA Motifs with IUPAC codes

All 15 non-empty subsets of the DNA alphabet {A, C, G T}
can be represented by a single letter code (IUPAC code).

`A` means {A},
`C` means {C},
`G` means {G},
`T` means {T},

`B` means {C, G, T} (not A),
`D` means {A, G, T} (not C),
`H` means {A, C, T} (not G),
`V` means {A, C, G} (not T),

`R` means {A, G},
`Y` means {C, T},
`S` means {C, G},
`W` means {A, T},
`K` means {G, T},
`M` means {A, C},

`N` means {A, C, G, T}.

DNA Motifs with IUPAC codes

All 15 non-empty subsets of the DNA alphabet $\{A, C, G\ T\}$
can be represented by a single letter code (IUPAC code).

`A` means $\{A\}$,
`C` means $\{C\}$,
`G` means $\{G\}$,
`T` means $\{T\}$,

`B` means $\{C, G, T\}$ (not A),
`D` means $\{A, G, T\}$ (not C),
`H` means $\{A, C, T\}$ (not G),
`V` means $\{A, C, G\}$ (not T),

`R` means $\{A, G\}$,
`Y` means $\{C, T\}$,
`S` means $\{C, G\}$,
`W` means $\{A, T\}$,
`K` means $\{G, T\}$,
`M` means $\{A, C\}$,

`N` means $\{A, C, G, T\}$.

These symbols can be concatenated to describe DNA motifs.

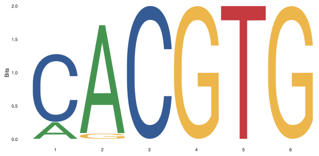
Transcription Factor Binding Sites

Transcription factor binding sites (TFBSs) can be described with IUPAC motifs.

Example from the JASPAR database:

Arnt is a nuclear basic helix-loop-helix (bHLH) transcription factor.

In Mus musculus, it binds to the DNA sequence motif `[AC][AG]CGTG` or `MRCGTG`:



JASPAR is an open-access database of curated, non-redundant transcription factor (TF) binding profiles stored as position frequency matrices (PFMs) and TF flexible models (TFFMs) for TFs across multiple species in six taxonomic groups [`https://jaspar.uio.no/`]

Motifs with Variable-Length Spacers

Some binding motifs consist of 3 parts:

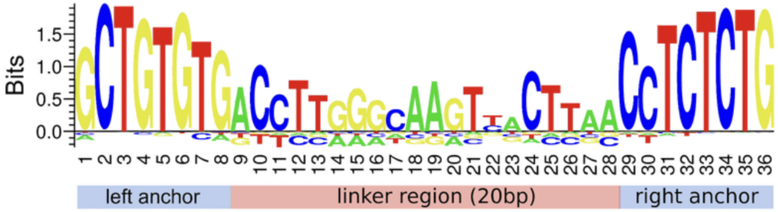
1. a specific 5' motif ("left anchor")
2. a variable middle part (flexible sequence and flexible length), described as $\mathtt{N}(u, v)$, meaning u to v arbitrary nucleotides
3. a specific 3' motif ("right anchor")

Motifs with Variable-Length Spacers

Some binding motifs consist of 3 parts:

1. a specific 5' motif ("left anchor")
2. a variable middle part (flexible sequence and flexible length), described as `N(`u, v`)`, meaning u to v arbitrary nucleotides
3. a specific 3' motif ("right anchor")

Example: ZNF768 binding pattern `RCTGTGYRN(17,23)CYTCTCTG`
[Rohrmoser et al.: "MIR sequences recruit zinc finger protein ZNF768 to expressed genes", Nucl. Acid Res. 47(2): p. 707, 2019]



Goal

Write a simple DNA motif search tool (in jit-compiled Python) that finds IUPAC motifs with variable-length spacers in FASTA sequences.

Goal

Write a simple DNA motif search tool (in jit-compiled Python) that finds IUPAC motifs with variable-length spacers in FASTA sequences.

Motif mini-language

DNA motifs are described as a sequence of **motif elements**.

A motif element is either

1. a single IUPAC character from `ACGTBDHVRYSWMKN`, or
2. a variable-length spacer of the form `N(`u`,`v`)`,

Goal

Write a simple DNA motif search tool (in jit-compiled Python) that finds IUPAC motifs with variable-length spacers in FASTA sequences.

Motif mini-language

DNA motifs are described as a sequence of **motif elements**.

A motif element is either

1. a single IUPAC character from `ACGTBDHVRYSWMKN`, or
2. a variable-length spacer of the form $\mathtt{N}(u, v)$,

such that variable-length spacers satisfy the following restrictions:

- they do not occur as first or last element of the motif,
- they do not occur next to each other,
- in $\mathtt{N}(u, v)$, we have $1 \leq u \leq v$.

Goal

Write a simple DNA motif search tool (in jit-compiled Python) that finds IUPAC motifs with variable-length spacers in FASTA sequences.

Motif mini-language

DNA motifs are described as a sequence of **motif elements**.

A motif element is either

1. a single IUPAC character from `ACGTBDHVRYSWMKN`, or
2. a variable-length spacer of the form `N(`u, v`)`,

such that variable-length spacers satisfy the following restrictions:

- they do not occur as first or last element of the motif,
- they do not occur next to each other,
- in `N(`u, v`)`, we have $1 \leq u \leq v$.

The only real restriction is $u \geq 1$. Dealing with $u = 0$ is more complex.

Goal

Write a simple DNA motif search tool (in jit-compiled Python)
that finds IUPAC motifs with variable-length spacers in FASTA sequences.

Example:

```
python motifmatcher.py --fasta genome.fa \
       --motif 'RCTGTGYRN(17,23)CYTCTCTG'
```

Goal

Write a simple DNA motif search tool (in jit-compiled Python)
that finds IUPAC motifs with variable-length spacers in FASTA sequences.

Example:

```
python motifmatcher.py --fasta genome.fa \
       --motif 'RCTGTGYRN(17,23)CYTCTCTG'
```

Desired output

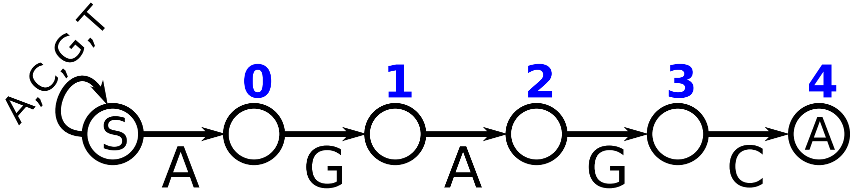
- list of (chromosome, position) intervals where the motif occurs
- simpler: list of positions per chromosome where an instance of the motif **ends**

Strategy

1. represent the motif as a non-deterministic finite automation (NFA), a concept from theoretical computer science,
2. specifically understand the challenges presented by variable-length spacers,
3. simulate the NFA efficiently (using one bit per state),
4. implement a CLI application stub,
5. implement the transformation from the given motif to an NFA,
6. develop a Python implementation of the core NFA simulation algorithm

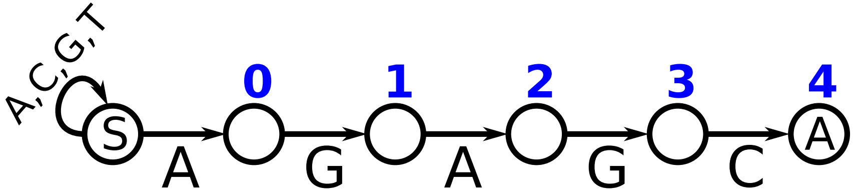
Nondeterministic Finite Automata for DNA Motifs

Consider motif $P = \texttt{AGAGC}$



Nondeterministic Finite Automata for DNA Motifs

Consider motif $P =$ `AGAGC`

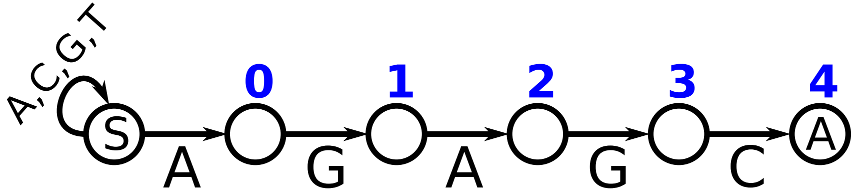


We have

- states (circles), some have numbers
- a start state (without number, marked `S`)
- an accepting (sometimes "final") state (marked `F`)
- an alphabet for the text (DNA) to be processed: `A`, `C`, `G`, `T`

Nondeterministic Finite Automata for DNA Motifs

Consider motif $P =$ `AGAGC`



We have

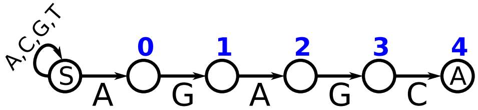
- states (circles), some have numbers
- a start state (without number, marked `S`)
- an accepting (sometimes "final") state (marked `F`)
- an alphabet for the text (DNA) to be processed: `A`, `C`, `G`, `T`
- a **transition function**:
  how to move from state to state when reading text characters

General Definition: Non-Deterministic Finite Automaton (NFA)

An **NFA** is a tuple $(Q, Q_0, F, \Sigma, \Delta)$, where

- Q is a finite set of **states**,
- $Q_0 \subset Q$ is a set of **start states**,
- $F \subset Q$ is a set of **accepting states**,
- Σ is an input **alphabet**, and
- $\Delta\colon Q \times \Sigma \to 2^Q$ is a **non-deterministic transition function**.

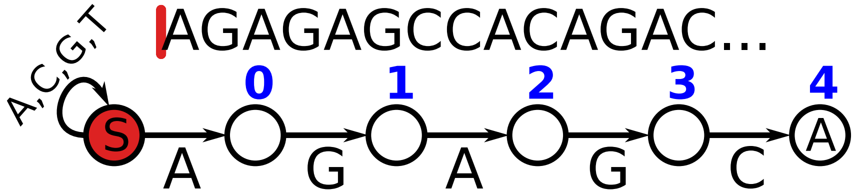
Example: motif $P = \texttt{AGAGC}$



- Motif is represented by a linear chain of states
- A state represents our progress in matching the motif
- Start state always remains active

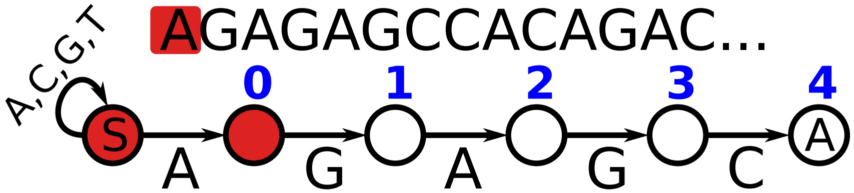
Searching for a Motif within a Text with an NFA

- motif $P = \texttt{AGAGC}$
- text $T = \texttt{AGAGAGCCACAGAC}$



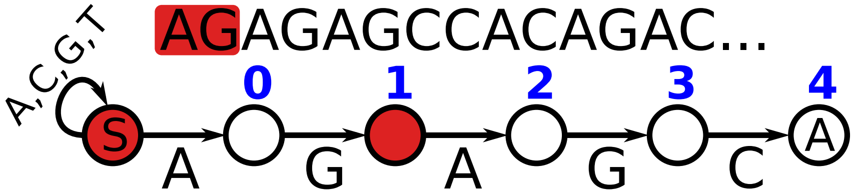
Searching for a Motif within a Text with an NFA

- motif $P =$ `AGAGC`
- text $T =$ `AGAGAGCCACAGAC`



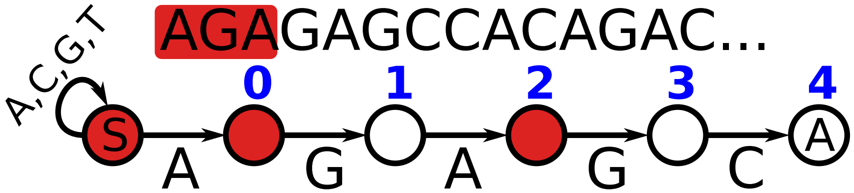
Searching for a Motif within a Text with an NFA

- motif $P = \texttt{AGAGC}$
- text $T = \texttt{AGAGAGCCACAGAC}$



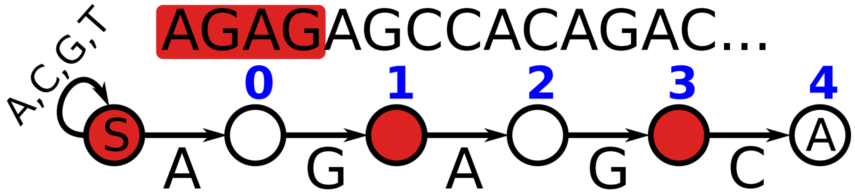
Searching for a Motif within a Text with an NFA

- motif $P = \texttt{AGAGC}$
- text $T = \texttt{AGAGAGCCACAGAC}$



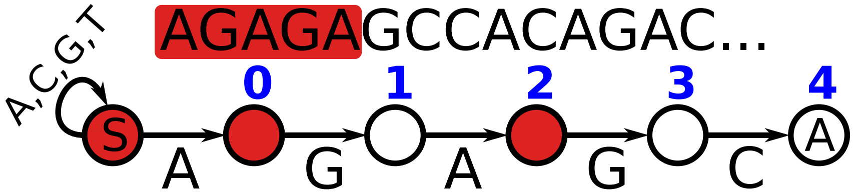
Searching for a Motif within a Text with an NFA

- motif $P = \texttt{AGAGC}$
- text $T = \texttt{AGAGAGCCACAGAC}$



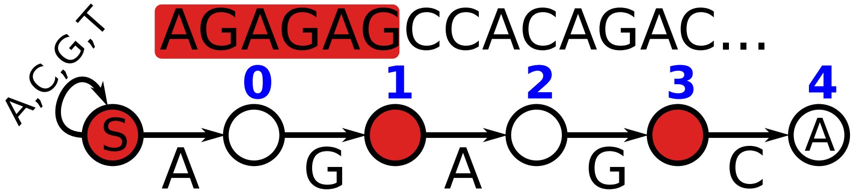
Searching for a Motif within a Text with an NFA

- motif $P = \texttt{AGAGC}$
- text $T = \texttt{AGAGAGCCACAGAC}$



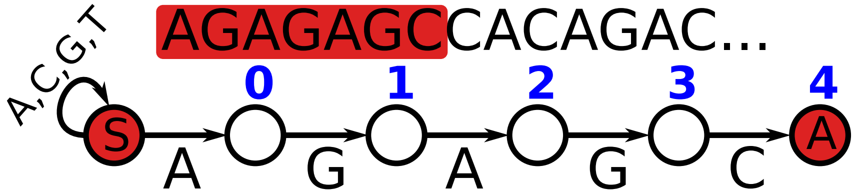
Searching for a Motif within a Text with an NFA

- motif $P = \texttt{AGAGC}$
- text $T = \texttt{AGAGAGCCACAGAC}$



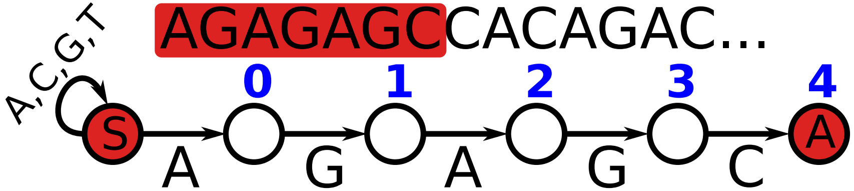
Searching for a Motif within a Text with an NFA

- motif P = `AGAGC`
- text T = `AGAGAGCCACAGAC`



Searching for a Motif within a Text with an NFA

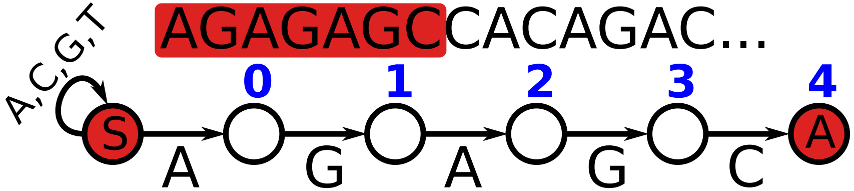
- motif $P = \texttt{AGAGC}$
- text $T = \texttt{AGAGAGCCACAGAC}$



and so on ...

Searching for a Motif within a Text with an NFA

- motif $P = \texttt{AGAGC}$
- text $T = \texttt{AGAGAGCCACAGAC}$

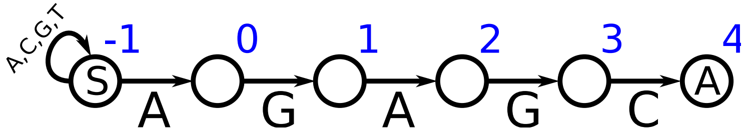


and so on ...

Things left to do

- Formally define this automaton.
- Give an efficient implementation.

DNA Motif NFA (Formally)



Pattern Search NFA for pattern $P \in \Sigma^m$

- state set $Q = \{-1, 0, \ldots, m-1\}$, where $m = |P|$
- start states $Q_0 = \{-1\}$
- accepting states $F = \{m-1\}$
- transition function Δ:

For $q = -1$: $$\Delta(-1, c) = \begin{cases} \{-1, 0\} & \text{if } c = P[0], \\ \{-1\} & \text{otherwise.} \end{cases}$$

For $q \in \{0, \ldots, m-2\}$: $$\Delta(q, c) = \begin{cases} \{q+1\} & \text{if } c = P[q+1], \\ \emptyset & \text{otherwise.} \end{cases}$$

For $q = m-1$: $$\Delta(m-1, c) = \emptyset$$

Implementation: Object-Oriented?

```python
class Node:
    def __init__(self, number, label, children):
        # do initialization of state
        pass

    def get_children_for_character(self, character):
        # for example ...
        return self.children[character]
```

Implementation: Object-Oriented?

```
class Node:
    def __init__(self, number, label, children):
        # do initialization of state
        pass

    def get_children_for_character(self, character):
        # for example ...
        return self.children[character]
```

Not recommended!

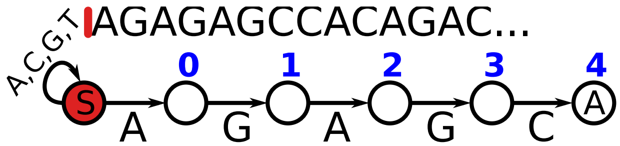
- An object-oriented implementation will be very inefficient.
- In an NFA, more than one state can be active; up to $m + 1$ states.
- In the motif's NFA, each target set size is bounded by 2.
- Thus, each step (text character) may execute $2(m + 1)$ activations.
- Total: $O(mn)$ steps

Efficient Implementation: Bit Parallelism

- On modern CPUs, logical and arithmetic operations on many bits (64 bits) take place in parallel in constant time: "**Bit parallelism**" $(+, -, \cdot, /, \oplus, \&, |, \sim, \ll, \gg)$
- The Pattern Search NFA is a linear chain of states, like bits in a CPU register.
- We only need one bit to represent whether a state is active or not.
- **Note:** States are numbered from left to right, bits from right to left!

Efficient Implementation: The Shift-And Algorithm

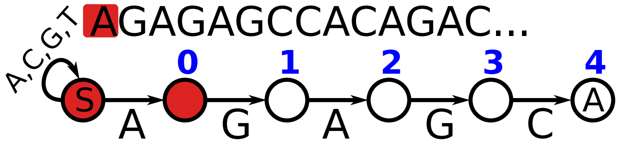
- We encode active states (without start) as a bit vector D, initially $D = 0$.
- The pattern is encoded in bit masks, one for each character.
- **Update:** $D \leftarrow ((D \ll 1) \mid 1)$ & $\text{mask}(\alpha)$, where α is the current text character:
  shift $\ll 1$: propagates activity to next state, $\mid 1$ propagates the start state;
  and & $\text{mask}(\alpha)$: removes falsely propagated activity.
- After each update, test whether accepting state $m-1$ is active.



D:	0	0	0	0	0
mask(A):	1	0	1	0	0
mask(G):	0	1	0	1	0
mask(C):	0	0	0	0	1

Efficient Implementation: The Shift-And Algorithm

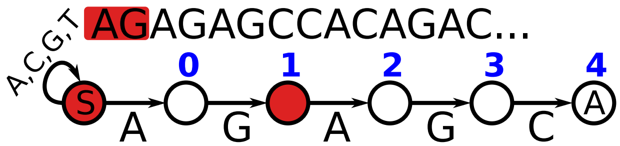
- We encode active states (without start) as a bit vector D, initially $D = 0$.
- The pattern is encoded in bit masks, one for each character.
- **Update:** $D \leftarrow ((D \ll 1) \mid 1) \mathbin{\&} \text{mask}(\alpha)$, where α is the current text character:
  shift $\ll 1$: propagates activity to next state, $\mid 1$ propagates the start state;
  and $\mathbin{\&} \text{mask}(\alpha)$: removes falsely propagated activity.
- After each update, test whether accepting state $m - 1$ is active.



	0	1	2	3	4
D:	1	0	0	0	0
mask(A):	1	0	1	0	0
mask(G):	0	1	0	1	0
mask(C):	0	0	0	0	1

Efficient Implementation: The Shift-And Algorithm

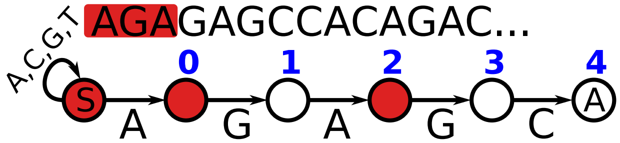
- We encode active states (without start) as a bit vector D, initially $D = 0$.
- The pattern is encoded in bit masks, one for each character.
- **Update:** $D \leftarrow ((D \ll 1) \mid 1)$ & $\text{mask}(\alpha)$, where α is the current text character:
  shift $\ll 1$: propagates activity to next state, $\mid 1$ propagates the start state;
  and & $\text{mask}(\alpha)$: removes falsely propagated activity.
- After each update, test whether accepting state $m-1$ is active.



D:	0	1	0	0	0
mask(A):	1	0	1	0	0
mask(G):	0	1	0	1	0
mask(C):	0	0	0	0	1

Efficient Implementation: The Shift-And Algorithm

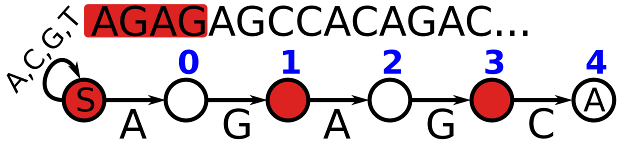
- We encode active states (without start) as a bit vector D, initially $D = 0$.
- The pattern is encoded in bit masks, one for each character.
- **Update:** $D \leftarrow ((D \ll 1) \mid 1) \;\&\; \text{mask}(\alpha)$, where α is the current text character:
  shift $\ll 1$: propagates activity to next state, $\mid 1$ propagates the start state;
  and $\&$ mask(α): removes falsely propagated activity.
- After each update, test whether accepting state $m - 1$ is active.



D:	1	0	1	0	0
mask(A):	1	0	1	0	0
mask(G):	0	1	0	1	0
mask(C):	0	0	0	0	1

Efficient Implementation: The Shift-And Algorithm

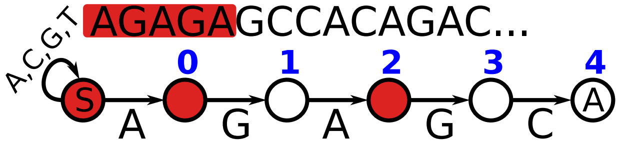
- We encode active states (without start) as a bit vector D, initially $D = 0$.
- The pattern is encoded in bit masks, one for each character.
- **Update:** $D \leftarrow ((D \ll 1) \mid 1) \,\&\, \text{mask}(\alpha)$, where α is the current text character:
  shift $\ll 1$: propagates activity to next state, $\mid 1$ propagates the start state;
  and $\&\, \text{mask}(\alpha)$: removes falsely propagated activity.
- After each update, test whether accepting state $m-1$ is active.



D:	0	1	0	1	0
mask(A):	1	0	1	0	0
mask(G):	0	1	0	1	0
mask(C):	0	0	0	0	1

Efficient Implementation: The Shift-And Algorithm

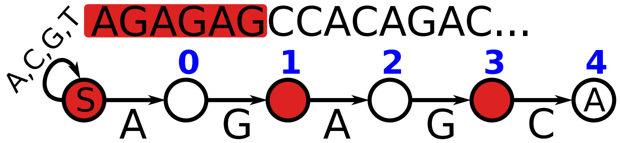
- We encode active states (without start) as a bit vector D, initially $D = 0$.
- The pattern is encoded in bit masks, one for each character.
- **Update:** $D \leftarrow ((D \ll 1) \mid 1)$ & $\mathrm{mask}(\alpha)$, where α is the current text character:
  shift $\ll 1$: propagates activity to next state, $\mid 1$ propagates the start state;
  and & $\mathrm{mask}(\alpha)$: removes falsely propagated activity.
- After each update, test whether accepting state $m - 1$ is active.



D:	1	0	1	0	0
mask(A):	1	0	1	0	0
mask(G):	0	1	0	1	0
mask(C):	0	0	0	0	1

Efficient Implementation: The Shift-And Algorithm

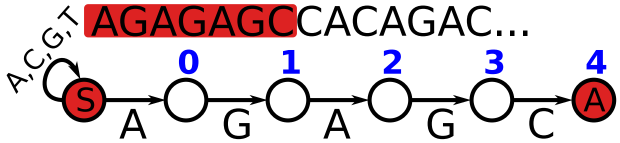
- We encode active states (without start) as a bit vector D, initially $D = 0$.
- The pattern is encoded in bit masks, one for each character.
- **Update:** $D \leftarrow ((D \ll 1) \mid 1)$ & $\mathrm{mask}(\alpha)$, where α is the current text character:
  shift $\ll 1$: propagates activity to next state, $\mid 1$ propagates the start state;
  and & $\mathrm{mask}(\alpha)$: removes falsely propagated activity.
- After each update, test whether accepting state $m-1$ is active.



D:	0	1	0	1	0
mask(A):	1	0	1	0	0
mask(G):	0	1	0	1	0
mask(C):	0	0	0	0	1

Efficient Implementation: The Shift-And Algorithm

- We encode active states (without start) as a bit vector D, initially $D = 0$.
- The pattern is encoded in bit masks, one for each character.
- **Update:** $D \leftarrow ((D \ll 1) \mid 1) \ \& \ \text{mask}(\alpha)$, where α is the current text character:
  shift $\ll 1$: propagates activity to next state, $\mid 1$ propagates the start state;
  and $\& \ \text{mask}(\alpha)$: removes falsely propagated activity.
- After each update, test whether accepting state $m - 1$ is active.



D:	0	0	0	0	1
mask(A):	1	0	1	0	0
mask(G):	0	1	0	1	0
mask(C):	0	0	0	0	1

Python Code for the Shift-And Algorithm

```python
from collections import defaultdict

def shift_and(P, T):
    masks = defaultdict(int)  # masks[c] == 0 if c not in masks
    bit = 1
    for c in P:
        masks[c] |= bit
        bit *= 2
    accept_state = bit // 2
    D = 0  # bit-mask of active states
    for i, c in enumerate(T):
        D = ((D << 1) | 1) & masks[c]  # Shift-And update!
        if (D & accept_state):
            yield i
```

Extending Shift-And for IUPAC Motifs

Current situation

- We have a bit-parallel method (implemented in Python) that deals with standard DNA motifs;
- no ambiguous IUPAC characters allowed yet
- no variable-length spacers allowed yet

Extending Shift-And for IUPAC Motifs

Current situation

- We have a bit-parallel method (implemented in Python) that deals with standard DNA motifs;
- no ambiguous IUPAC characters allowed yet
- no variable-length spacers allowed yet

Generalizations

1. Extend Shift-And method to "generalized strings" (allowing IUPAC)
2. Extend NFA and Shift-And to allow variable-length spacers

Extending Shift-And for IUPAC Motifs

Current situation

- We have a bit-parallel method (implemented in Python) that deals with standard DNA motifs;
- no ambiguous IUPAC characters allowed yet
- no variable-length spacers allowed yet

Generalizations

1. Extend Shift-And method to “generalized strings” (allowing IUPAC)
2. Extend NFA and Shift-And to allow variable-length spacers

Definition

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.
The non-empty subsets of $\{A,C,G,T\}$ are represented by IUPAC characters.

The Shift-And Algorithm for Generalized Strings

Recall the Shift-And update with active state bits D:
$D \leftarrow ((D \ll 1) \mid 1)\ \&\ \text{mask}(c)$

The Shift-And Algorithm for Generalized Strings

Recall the Shift-And update with active state bits D:
$D \leftarrow ((D \ll 1) \mid 1)$ & $\mathrm{mask}(c)$

Surprise!

- The Shift-And algorithm can process generalized strings **without modifications**.
- The bit masks 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 update with active state bits D:
$D \leftarrow ((D \ll 1) \mid 1)$ & $\text{mask}(c)$

Surprise!

- The Shift-And algorithm can process generalized strings **without modifications**.
- The bit masks tell which characters are allowed at which position.
  It is no problem that more than one bit is set at some positions.
- **Example:** $P = \texttt{GNAGGA}$:

bit	012345
P	GNAGGA
mask[A]	011001
mask[G]	110110

An NFA for Variable-Length Spacers

- We need ϵ-transitions, an extension of the standard NFA definition: ϵ-transitions happen instantaneously, without consuming a character.

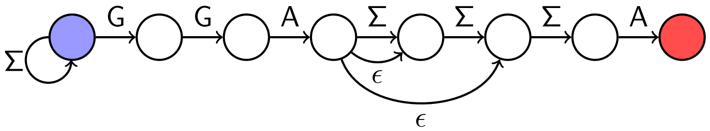
An NFA for Variable-Length Spacers

- 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** `N`s in each run are optional.
  (One could do it differently, but that would be harder to implement!)

An NFA for Variable-Length Spacers

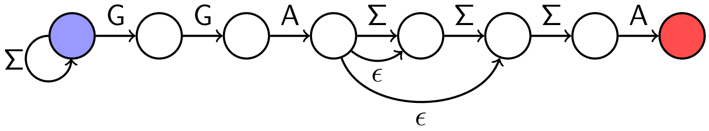
- We need ϵ-transitions, an extension of the standard NFA definition: ϵ-transitions happen instantaneously, without consuming a character.
- The ϵ-transitions allow us to skip the optional characters. For technical reasons, they **exit the initial state** of the run; the **first** `N`s in each run are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

Example: $P =$ `GGAN(1,3)A`:



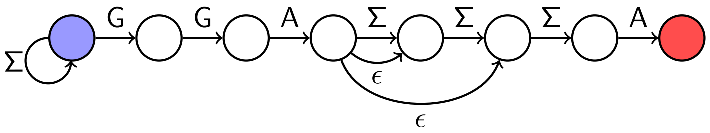
Note: Σ represents the full DNA alphabet $\{A, C, G, T\}$.

Bit-parallel Implementation



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

Bit-parallel Implementation

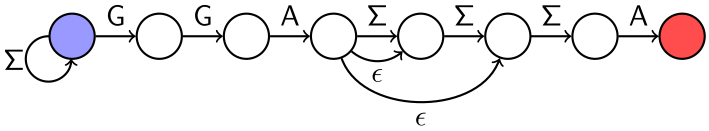


- We use the Shift-And algorithm on the maximal-length pattern as a basis. Then we additionally implement the ϵ-transitions.
- Masks are constructed as before (for `N`: 1-bits for each character).

	ANNNAGG
mask[A]	1111100
mask[C]	0111000
mask[G]	0111011
mask[T]	0111000

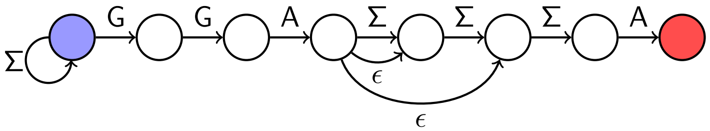
(Bits are usually numbered from right to left; hence, we show the masks in this way.)

Implementation of ϵ-Transitions



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

Implementation of ϵ-Transitions



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

	`ANNNAGG`
F	0100000
I	0000100

Propagation of Ones

- Let A be the bit mask of active states.
  Then $A \mathbin{\&} I$ selects active I-states. (Here, $\&$ is bitwise AND.)
- Subtraction $F - (A \mathbin{\&} I)$ propagates 1-bits and zeroes F-bits if I-state is active:

$$\begin{array}{rr} F & 0100000 \\ A \mathbin{\&} I & 0000100 \\ \hline - & 0011100 \end{array}$$

Propagation of Ones

- Let A be the bit mask of active states.
  Then $A \,\&\, I$ selects active I-states. (Here, $\&$ is bitwise AND.)
- Subtraction $F - (A \,\&\, I)$ propagates 1-bits and zeroes F-bits if I-state is active:

$$\begin{array}{r l} F & 0100000 \\ A \,\&\, I & 0000100 \\ \hline - & 0011100 \end{array}$$

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

$$\begin{array}{r l} F & 010000100000 \\ I & 000010000100 \\ A \,\&\, I & 000000000100 \\ \hline F - (A \,\&\, I) & 010000011100 \end{array}$$

Propagation of Ones (Continued)

- **Problem:** Inactive I-states keep corresponding F-bit set.
- **Solution:** Zero out F-bits by a bitwise `and` with the negation of F:

$$
\begin{array}{ll}
F & 010000100000 \\
A \mathbin{\&} I & 000000000100 \\
\hline
F - (A \mathbin{\&} I) & 010000011100
\end{array}
$$

Propagation of Ones (Continued)

- **Problem:** Inactive I-states keep corresponding F-bit set.
- **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)

- **Problem:** Inactive I-states keep corresponding F-bit set.
- **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 update 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)
   ```

Python Code

```python
def find_matches_python(mask, I, F, accept, text):
    """yield each end position of aN NFA match against the sequence"""
    # mask: a defaultdict(int) to give 0 if a character is not present
    # I, F: the bit masks for the epsilon transitions as shown
    # accept: the bit mask for the accept state
    A = 0
    for i, c in enumerate(text):
        A = ((A << 1) | 1) & int(mask[c])
        A = A | ((F - (A & I)) & ~F)
        if A & accept:
            yield i  # makes this a generator function
```

Python Code

```python
def find_matches_python(mask, I, F, accept, text):
    """yield each end position of aN NFA match against the sequence"""
    # mask: a defaultdict(int) to give 0 if a character is not present
    # I, F: the bit masks for the epsilon transitions as shown
    # accept: the bit mask for the accept state
    A = 0
    for i, c in enumerate(text):
        A = ((A << 1) | 1) & int(mask[c])
        A = A | ((F - (A & I)) & ~F)
        if A & accept:
            yield i  # makes this a generator function
```

The motif finding code is simple enough. In Python, it's slow, though.
Also, we still need to parse the given motif into the NFA (masks) and read the genome.

Our Command-Line Interface (CLI)

```python
import argparse

def get_argument_parser():
    p = argparse.ArgumentParser(description="DNA Motif Searcher")
    p.add_argument("--motif", "-m",
        default="RCTGTGYRN(17,23)CYTCTCTG",  # Nucl. Acid Res. 47(2):707, 2019
        help="DNA motif (IUPAC) with optional N(min,max) elements")
    p.add_argument("--fasta", "-f", required=True,
        help="FASTA file of genome")
    p.add_argument("--maxresults", "-R", type=int, default=10_000_000,
        help="maximum number of output positions per chromosome [10 mio] "
        "when using the numba-compiled version")
    p.add_argument("--slow", action="store_true",
        help="use a slower pure Python implementation")
    return p

if __name__ == "__main__":
    main(get_argument_parser().parse_args())
```

The main Function of the Tool

```python
def main(args):
    if not args.slow:
        results = np.zeros(args.maxresults, dtype=np.uint64)
    nfa = build_nfa(args.motif)
    for header, sequence in fasta_items(args.fasta):
        print("#", header.decode("ASCII"))
        if args.slow:
            for pos in find_matches_python(*nfa, sequence):
                print(pos)
        else:
            nresults = find_matches_fast(*nfa, sequence, results)
            if nresults > args.maxresults:
                print(f"! Too many results, showing first {args.maxresults}")
                nresults = args.maxresults
            print(*list(results[:nresults]), sep="\n")
```

- supports two modes: slow Python, compiled numba
- still need to implement `build_nfa`, `fasta_items`, `find_matches_fast`

Using FASTA Format for Genome Sequences

Usage examples:

```
python motifmatcher.py --fasta genome.fa --motif 'RCTGTGYRN(17,23)CYTCTCTG'
python motifmatcher.py --fasta <(xz -cd genome.fa.xz) --motif 'CG'
python motifmatcher.py --fasta <(pigz -cd -p 2 genome.fa.gz) --motif 'CG'
```

- We assume that the "text" (genome) is given as a FASTA file.
  This also works with compressed files, if we decompress them in a sub-process.

Using FASTA Format for Genome Sequences

Usage examples:

```
python motifmatcher.py --fasta genome.fa --motif 'RCTGTGYRN(17,23)CYTCTCTG'
python motifmatcher.py --fasta <(xz -cd genome.fa.xz) --motif 'CG'
python motifmatcher.py --fasta <(pigz -cd -p 2 genome.fa.gz) --motif 'CG'
```

- We assume that the “text” (genome) is given as a FASTA file.
  This also works with compressed files, if we decompress them in a sub-process.
- We do not discuss details of FASTA parsing, but we provide simple Python code.
  (We prefer not to use BioPython or other libraries b/c they typically add bloat.)

Reading and Parsing FASTA

The FASTA format is a text format that consists of **header** lines and **sequence** lines. Header lines start with >.

```
>NC_000913.3 Escherichia coli str. K-12 substr. MG1655, complete genome
AGCTTTTCATTCTGACTGCAACGGGCAATATGTCTCTGTGTGGATTAAAAAAAGAGTGTCTGATAGCAGCTTCTGAACTG
GTTACCTGCCGTGAGTAAATTAAAATTTTATTGACTTAGGTCACTAAATACTTTAACCAATATAGGCATAGCGCACAGAC
AGATAAAAATTACAGAGTACACAACATCCATGAAACGCATTAGCACCACCATTACCACCACCATCACCATTACCACAGGT
AACGGTGCGGGCTGACGCGTACAGGAAACACAGAAAAAAGCCCGCACCTGACAGTGCGGGCTTTTTTTTTCGACCAAAGG
TAACGAGGTAACAACCATGCGAGTGTTGAAGTTCGGCGGTACATCAGTGGCAAATGCAGAACGTTTTCTGCGTGTTGCCG
ATATTCTGGAAAGCAATGCCAGGCAGGGGCAGGTGGCCACCGTCCTCTCTGCCCCCGCCAAAATCACCAACCACCTGGTG
GCGATGATTGAAAAAACCATTAGCGGCCAGGATGCTTTACCCAATATCAGCGATGCCGAACGTATTTTTGCCGAACTTTT
GACGGGACTCGCCGCCGCCCAGCCGGGGTTCCCGCTGGCGCAATTGAAAACTTTCGTCGATCAGGAATTTGCCCAAATAA
AACATGTCCTGCATGGCATTAGTTTGTTGGGGCAGTGCCCGGATAGCATCAACGCTGCGCTGATTTGCCGTGGCGAGAAA
ATGTCGATCGCCATTATGGCCGGCGTATTAGAAGCGCGCGGTCACAACGTTACTGTTATCGATCCGGTCGAAAAACTGCT
GGCAGTGGGGCATTACCTCGAATCTACCGTCGATATTGCTGAGTCCACCCGCCGTATTGCGGCAAGCCGCATTCCGGCTG
ATCACATGGTGCTGATGGCAGGTTTCACCGCCGGTAATGAAAAAGGCGAACTGGTGGTGCTTGGACGCAACGGTTCCGAC
TACTCTGCTGCGGTGCTGGCTGCCTGTTTACGCGCCGATTGTTGCGAGATTTGGACGGACGTTGACGGGGTCTATACCTG
```

Reading FASTA

We read FASTA as bytes (not text), convert to uppercase ACGT, and store everything in a numpy array of data type `uint8`.

```
def fasta_items(filename):
    """
    generator function that yields each (header, sequence) pair from a FAS
    The header is given as an immutable 'bytes' object;
    The sequence is given as a mutable numpy array of dtype uint8.
    """
    with open(filename, "rb") as f:
        for (header, seq) in _fasta_reads_from_filelike(f):
            seqb = np.frombuffer(seq, dtype=np.uint8)
            seq_to_upper(seqb)  # translate in-place
            yield (header, seqb)
```

Parsing FASTA

```python
def _fasta_reads_from_filelike(f, COMMENT=b';'[0], HEADER=b'>'[0]):
    """yield each FASTA record as (header: bytes, seq: bytearray)"""
    strip = bytes.strip
    header = seq = None
    for line in f:
        line = strip(line)
        if len(line) == 0 or line[0] == COMMENT:
            continue
        if line[0] == HEADER:
            if header is not None:
                yield (header, seq)
            header = line[1:]
            seq = bytearray()
            continue
        seq.extend(line)
    if header is not None:
        yield (header, seq)
```

Converting Sequences to Uppercase (using numba!)

```python
def make_toupper_table():
    T = np.arange(256, dtype=np.uint8)
    T[ord('a')] = ord('A')
    T[ord('c')] = ord('C')
    T[ord('g')] = ord('G')
    T[ord('t')] = ord('T')
    T[ord('u')] = ord('T')
    T[ord('n')] = ord('N')
    return T

@njit
def seq_to_upper(seq, T=make_toupper_table()):
    n = len(seq)
    for i in range(n):
        seq[i] = T[seq[i]]
```

Using numba makes sense here: simple task, long chromosomes; can be vectorized.

Building the NFA from the Motif

- The motif is given as a string: `'RCTGTGYRN(17,23)CYTCTCTG'`
- We must split it into “fixed” parts (runs of single letters) and variable-length spacers ($\mathtt{N}(L, U)$ parts).
- A single `N` (without the (L, U) specifier) is treated like any IUPAC letter.
- The length of the NFA is determined by the **maximal length** of any motif instance.

```
def build_nfa(motif, iupac=_IUPAC):
    """Build an NFA from a IUPAC motif with additonal N(low,high) elements"""
    # _IUPAC is a dictionary of the IUPAC characters; see next slide.
    ...
    return mask, I, F, accept
    # mask is a numpy array of 256 elements (one for each byte).
    # Nucleotides are their ASCII codes, e.g. A=65, C=67, ...
    # I, F, accept are all single 64-bit integers (bit patterns).
```

Defining the IUPAC alphabet

```
_IUPAC = defaultdict(list,
    A=[ord('A')],  # character -> list of ASCII codes
    C=[ord('C')],
    G=[ord('G')],
    T=[ord('T')],
    R=[ord('A'), ord('G')],
    Y=[ord('C'), ord('T')],
    S=[ord('C'), ord('G')],
    W=[ord('A'), ord('T')],
    K=[ord('G'), ord('T')],
    M=[ord('A'), ord('C')],
    B=[ord('C'), ord('G'), ord('T')],
    D=[ord('A'), ord('G'), ord('T')],
    H=[ord('A'), ord('C'), ord('T')],
    V=[ord('A'), ord('C'), ord('G')],
    N=[ord('A'), ord('C'), ord('G'), ord('T')],
    )
```

Basic Parsing

```
def build_nfa(motif, iupac=_IUPAC):
    motif = motif.upper()  # ensure upper case
    mlist = []  # maximal-length list of IUPAC symbols
    masks = np.zeros(256, dtype=np.uint64)  # array of masks
    I = F = 0  # masks I and F
    spacer_allowed = False  # keep track of whether a spacer is currently OK
    parts = re.split(r"(N\(\d+,\d+\))", motif)  # split into alternating parts
    for part in parts:
        ...  # process run of symbols or an N() element
    mstring = "".join(mlist)  # maximal motif as string
    if len(mstring) > 64:
        raise ValueError(f"Error: maximal motif length is {len(mstring)} > 64")
    for bit, c in enumerate(mstring):
        value = (1 << bit)
        for a in iupac[c]:
            masks[a] |= np.uint64(value)
    accept = value
    return masks, I, F, accept
```

Looping over Parts

A part is either a run of single IUPAC symbols or an `N`(L, U) element.

```python
for part in parts:
    if not part:
        continue  # skip empty parts (just to be safe)
    if part.startswith("N("):  # variable-length spacer
        if not spacer_allowed:
            raise ValueError(f"Error: spacer not allowed here: {part}")
        maxlen, optionals = parse_spacer(part)  # parse `N(3,5)` etc.
        bit_I = len(mlist) - 1
        I |= (1 << bit_I)
        F |= (1 << (bit_I + optionals))
        mlist.extend(["N"] * maxlen)
        spacer_allowed = False
    else:  # run of single IUPAC symbols
        mlist.extend(list(part))
        spacer_allowed = True
```

Parsing a Variable-Length Spacer Description

We use a regular expression (`re` module) to parse the numbers out of the string,
and to ensure that the description ends with a closing parenthesis.
This will raise an Error (and terminate) if the string is not correctly formed.
In general, more and better error checking could be added to the code.

```
def parse_spacer(spacer):
    """parse a string like N(minlen,maxlen); return integers maxlen, optionals"""
    match = re.match(r"N\((\d+),(\d+)\)$", spacer)
    minlen = int(match.group(1))
    maxlen = int(match.group(2))
    optionals = maxlen - minlen
    return maxlen, optionals
```

Summary

Topics covered in this part

- IUPAC alphabet (symbols representing subsets of $\{A, C, G, T\}$)
- Non-deterministic finite automata (NFAs)
- NFAs with epsilon transitions to describe DNA motifs with variable-length spacers
- Bit-parallel implementation of NFAs:
  - Shift-And Algorithm
  - Extension to variable-length spacers (using subtraction)
- A small Python command-line application
- Reading and processing a FASTA genome
- Converting the motif description into an NFA (bit-masks)

Summary

Topics covered in this part

- IUPAC alphabet (symbols representing subsets of $\{A, C, G, T\}$)
- Non-deterministic finite automata (NFAs)
- NFAs with epsilon transitions to describe DNA motifs with variable-length spacers
- Bit-parallel implementation of NFAs:
  - Shift-And Algorithm
  - Extension to variable-length spacers (using subtraction)
- A small Python command-line application
- Reading and processing a FASTA genome
- Converting the motif description into an NFA (bit-masks)

Outlook

- How to speed up the Python application using `numba`
- In general: How to re-factor and transform Python code for `numba`
- Refinements: Parallelization; a more beautiful GUI