UNIVERSITÄT DES SAARLANDES

ZBI ZENTRUM FÜR BIOINFORMATIK

The Knuth-Morris-Pratt Algorithm

Algorithms for Sequence Analysis

Sven Rahmann

Summer 2024

Review

Algorithms for the Exact Pattern Search Problem

- Naive Algorithm:
  $O(mn)$ time worst-case, $O(n)$ time expected
- Horspool Algorithm:
  $O(mn)$ time worst-case, $O(m + n/m)$ best-case (and expected, under assumptions)
- Shift-And Algorithm:
  $O(m + mn/w)$ time, bit-parallel, w is register width (64 bits)

Question

Is there an algorithm that will **always** run in $O(m + n)$ time?

Review

Algorithms for the Exact Pattern Search Problem

- Naive Algorithm:
  $O(mn)$ time worst-case, $O(n)$ time expected
- Horspool Algorithm:
  $O(mn)$ time worst-case, $O(m + n/m)$ best-case (and expected, under assumptions)
- Shift-And Algorithm:
  $O(m + mn/w)$ time, bit-parallel, w is register width (64 bits)

Question

Is there an algorithm that will **always** run in $O(m + n)$ time?

Idea

Use a DFA for pattern search.
Process each text character exactly once in constant time.

Background from Theoretical Computer Science

Reminder: Equivalence of NFAs and DFAs

Let Σ be an alphabet.
Let L be a (finite or infinite) set of strings over Σ (a **language**).
Then the following statements are equivalent.

1. L is described by a regular expression.
2. There exists an NFA that accepts exactly L.
3. There exists a DFA that accepts exactly L.

A language accepted by any of the above is called a **regular language**.

Background from Theoretical Computer Science

Reminder: Equivalence of NFAs and DFAs

Let Σ be an alphabet.
Let L be a (finite or infinite) set of strings over Σ (a **language**).
Then the following statements are equivalent.

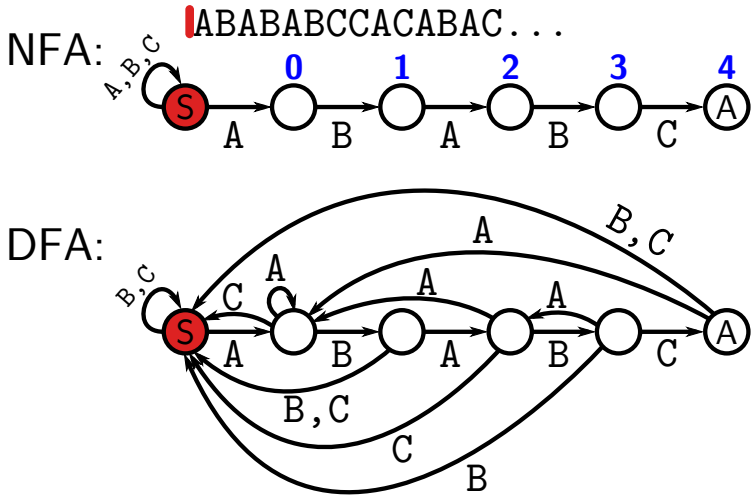
1. L is described by a regular expression.
2. There exists an NFA that accepts exactly L.
3. There exists a DFA that accepts exactly L.

A language accepted by any of the above is called a **regular language**.

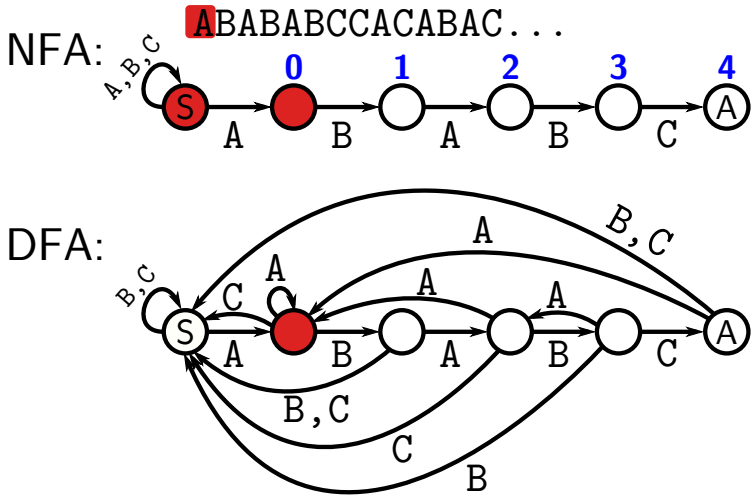
Consequence

For any pattern search NFA, there is an **equivalent** DFA
(i.e., one that accepts exactly the same set of strings $\Sigma^* P$).

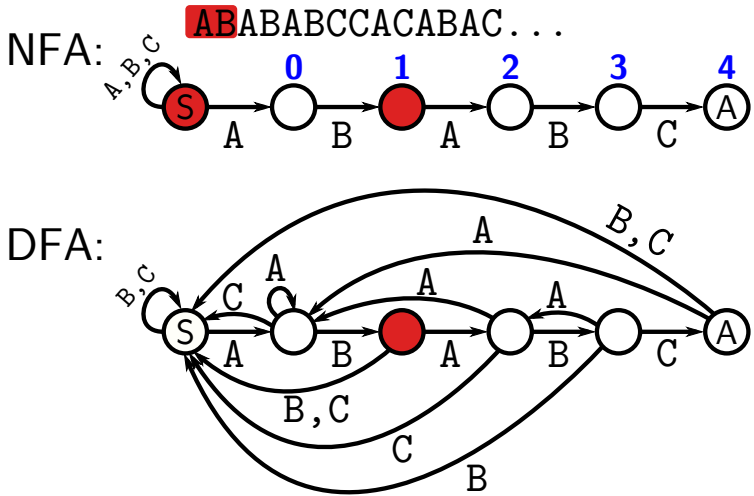
NFA-based vs. DFA-based Pattern Matching



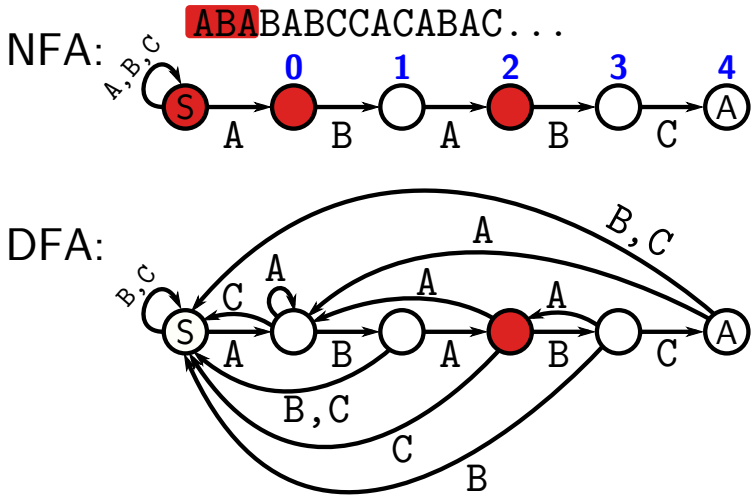
NFA-based vs. DFA-based Pattern Matching



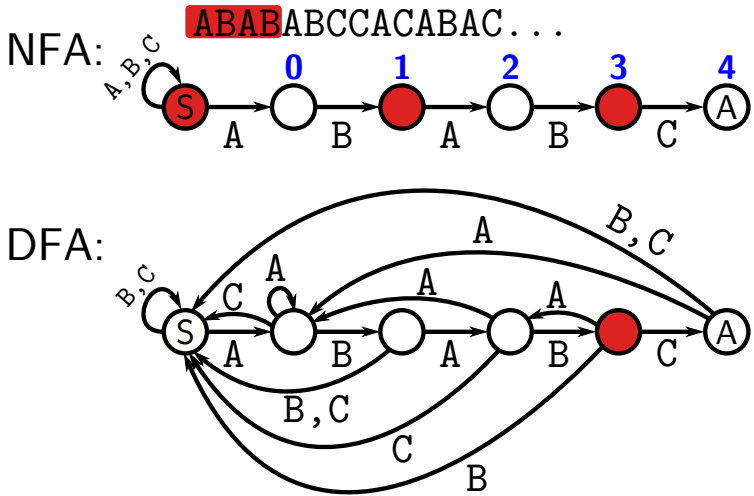
NFA-based vs. DFA-based Pattern Matching



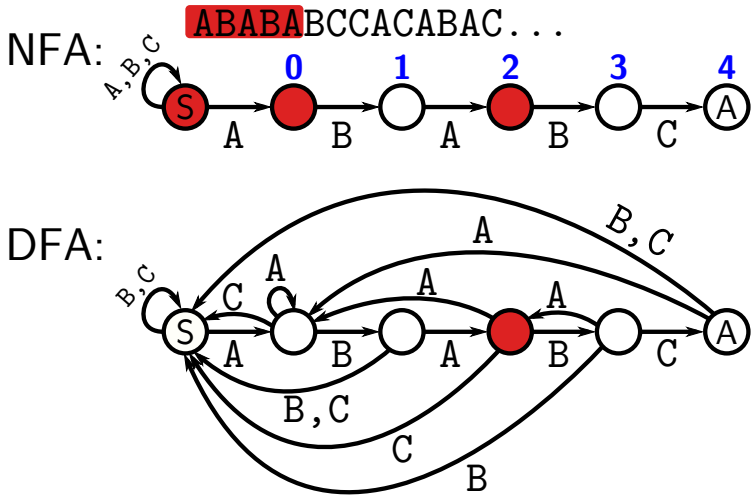
NFA-based vs. DFA-based Pattern Matching



NFA-based vs. DFA-based Pattern Matching



NFA-based vs. DFA-based Pattern Matching



The Subset Construction

Transforming a general NFA into an equivalent DFA

- **Idea:** create one DFA state for every **subset** of NFA states.
- We can omit state sets that are **unreachable** from the start set.
- Subset construction may lead to **exponential blow-up** of the number of states: $|Q|$ NFA states may turn into $2^{|Q|}$ DFA states.
- In practice, an exponential blow-up does not always happen (many state sets are **unreachable**).

Transforming a Pattern Search NFA into an equivalent DFA

Claim: The equivalent DFA has the **same** number of states as the NFA.
(There is no blow-up at all. This is remarkable. We will explain why this is so.)

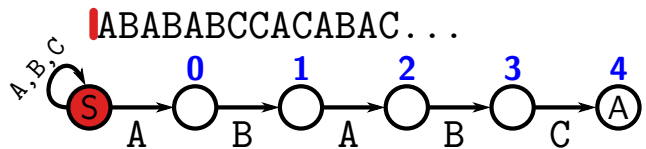
Reachable State Sets in the Pattern Search NFA

Lemma (from previous lecture): NFA state set invariant

Let $A \subset Q$ be a set of active states of the NFA.
Then, $q \in A \setminus \{-1\}$ iff the last $q+1$ read characters equal the pattern's prefix $P[\ldots q]$.
In particular, state $|P|-1$ is active iff the last $|P|$ characters equal the full pattern.

Lemma

Let A be the set of active states of a pattern matching NFA. Now, let $a^* := \max A$.
Then, A is completely determined by a^*.



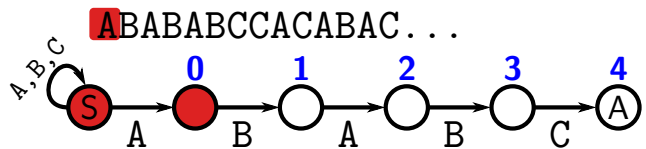
Reachable State Sets in the Pattern Search NFA

Lemma (from previous lecture): NFA state set invariant

Let $A \subset Q$ be a set of active states of the NFA.
Then, $q \in A \setminus \{-1\}$ iff the last $q+1$ read characters equal the pattern's prefix $P[\ldots q]$.
In particular, state $|P|-1$ is active iff the last $|P|$ characters equal the full pattern.

Lemma

Let A be the set of active states of a pattern matching NFA. Now, let $a^* := \max A$.
Then, A is completely determined by a^*.



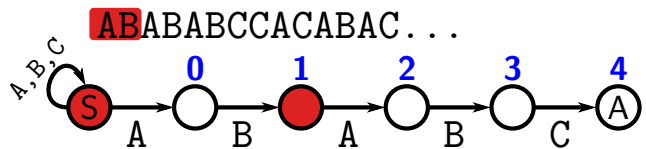
Reachable State Sets in the Pattern Search NFA

Lemma (from previous lecture): NFA state set invariant

Let $A \subset Q$ be a set of active states of the NFA.
Then, $q \in A \setminus \{-1\}$ iff the last $q+1$ read characters equal the pattern's prefix $P[\ldots q]$.
In particular, state $|P|-1$ is active iff the last $|P|$ characters equal the full pattern.

Lemma

Let A be the set of active states of a pattern matching NFA. Now, let $a^* := \max A$.
Then, A is completely determined by a^*.



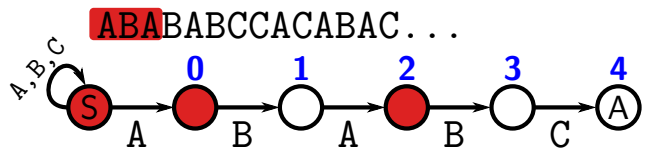
Reachable State Sets in the Pattern Search NFA

Lemma (from previous lecture): NFA state set invariant

Let $A \subset Q$ be a set of active states of the NFA.
Then, $q \in A \setminus \{-1\}$ iff the last $q+1$ read characters equal the pattern's prefix $P[\ldots q]$.
In particular, state $|P|-1$ is active iff the last $|P|$ characters equal the full pattern.

Lemma

Let A be the set of active states of a pattern matching NFA. Now, let $a^* := \max A$.
Then, A is completely determined by a^*.



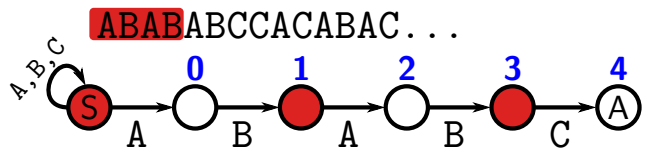
Reachable State Sets in the Pattern Search NFA

Lemma (from previous lecture): NFA state set invariant

Let $A \subset Q$ be a set of active states of the NFA.
Then, $q \in A \setminus \{-1\}$ iff the last $q+1$ read characters equal the pattern's prefix $P[\ldots q]$.
In particular, state $|P|-1$ is active iff the last $|P|$ characters equal the full pattern.

Lemma

Let A be the set of active states of a pattern matching NFA. Now, let $a^* := \max A$.
Then, A is completely determined by a^*.



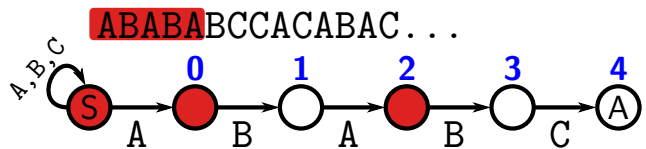
Reachable State Sets in the Pattern Search NFA

Lemma (from previous lecture): NFA state set invariant

Let $A \subset Q$ be a set of active states of the NFA.
Then, $q \in A \setminus \{-1\}$ iff the last $q+1$ read characters equal the pattern's prefix $P[\ldots q]$.
In particular, state $|P|-1$ is active iff the last $|P|$ characters equal the full pattern.

Lemma

Let A be the set of active states of a pattern matching NFA. Now, let $a^* := \max A$.
Then, A is completely determined by a^*.



Consequences of State Set Lemma

Lemma (from previous slide)

Let A be the set of active states of a pattern matching NFA. Now, let $a^* := \max A$. Then, A is completely determined by a^*.

Consequences

For each possible a^* from -1 to $m-1$, there is exactly one reachable set of active states.
$\Rightarrow$ there are exactly $m+1$ possible sets of active states.
$\Rightarrow$ there is an equivalent DFA with $m+1$ states.

Consequences of State Set Lemma

Lemma (from previous slide)

Let A be the set of active states of a pattern matching NFA. Now, let $a^* := \max A$. Then, A is completely determined by a^*.

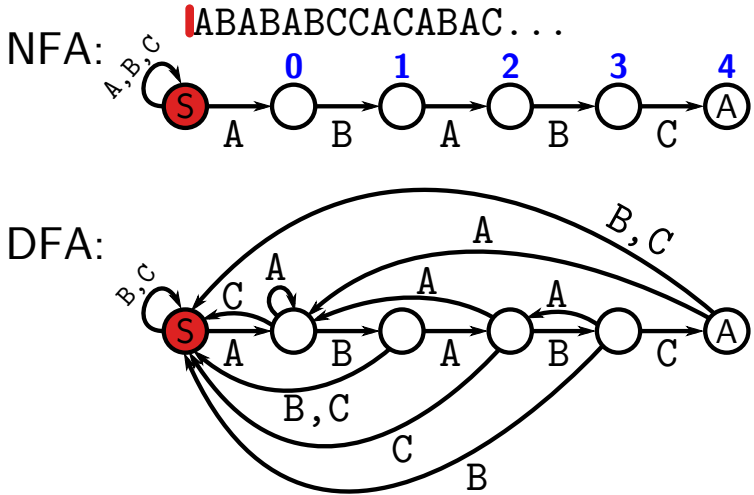
Consequences

For each possible a^* from -1 to $m-1$, there is exactly one reachable set of active states.
$\Rightarrow$ there are exactly $m+1$ possible sets of active states.
$\Rightarrow$ there is an equivalent DFA with $m+1$ states.

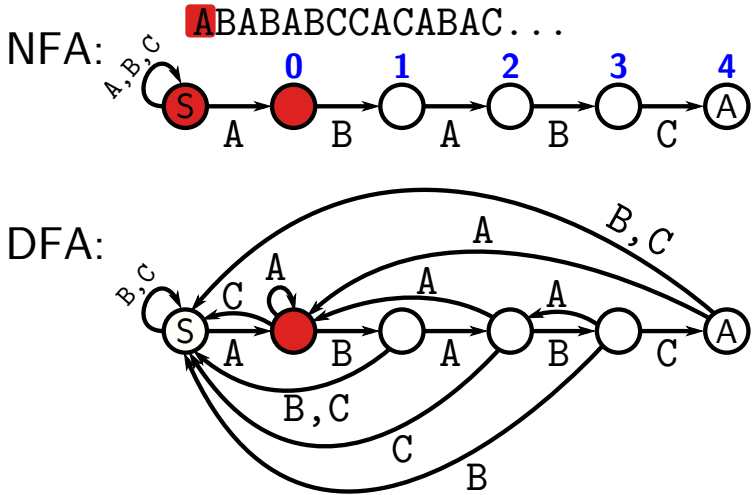
DFA state set

- We use same state set $Q = \{-1, \ldots, m-1\}$ for DFAs.
- Being in state $q \in Q$ in DFA $\iff$ NFA has active states set A with $\max A = q$.

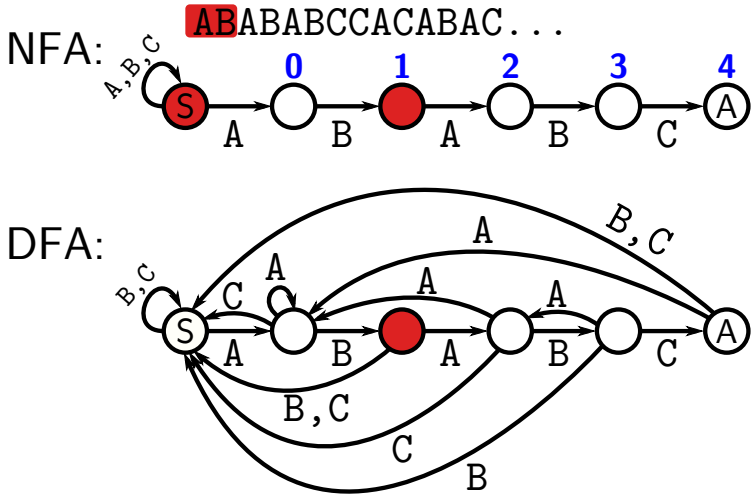
NFA-based vs. DFA-based Pattern Matching (again)



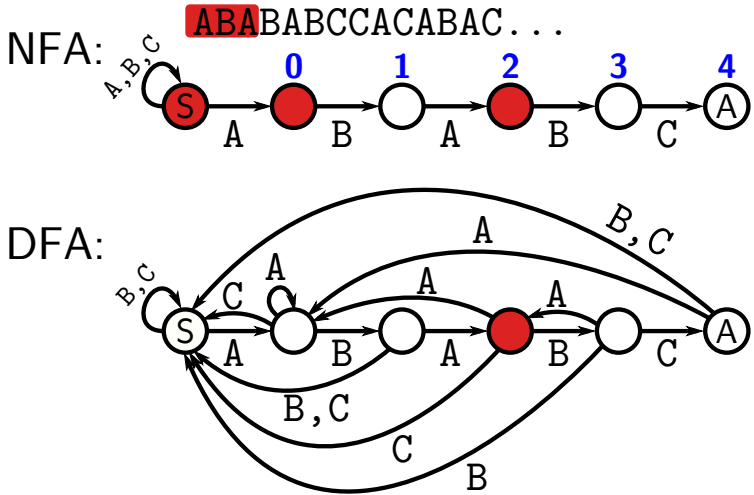
NFA-based vs. DFA-based Pattern Matching (again)



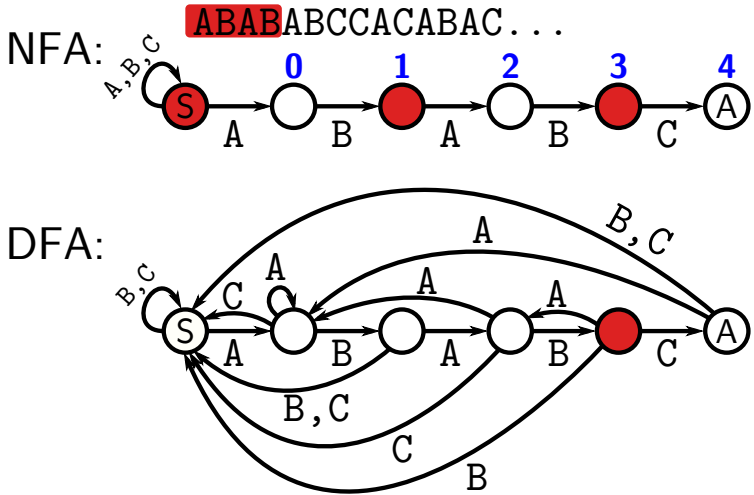
NFA-based vs. DFA-based Pattern Matching (again)



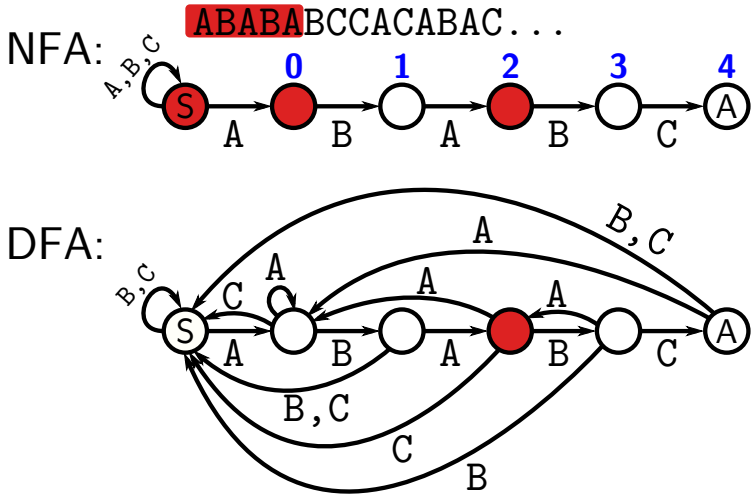
NFA-based vs. DFA-based Pattern Matching (again)



NFA-based vs. DFA-based Pattern Matching (again)



NFA-based vs. DFA-based Pattern Matching (again)



DFA-based Pattern Matching: Overview

Algorithm overview

1. Construct pattern search NFA in $O(m)$ time (last lecture).
2. Construct pattern search DFA
   by computing the set of active states for every possible a^* in $O(|\Sigma|m^2)$ time.
3. Use DFA for searching text in $O(n)$ time.

Running Times

- **Total time:** $O(|\Sigma|m^2 + n)$
- We want to improve this to $O(m + n)$.

DFA-based Pattern Matching: Code

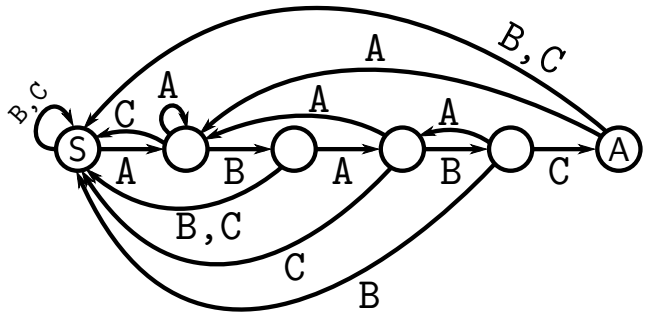
```
def matches_from_DFA_with_delta(T, m, delta):
    q = -1
    for i in range(len(T)):
        q = delta(q, T[i])  # O(1) time lookup
        if q == m - 1:
            yield (i-m+1, i+1)  # interval with P

def DFA(P, T):
    delta = compute_DFA_delta_table(P)  # not shown
    return matches_from_DFA_with_delta(T, len(P), delta)
```

The Knuth-Morris-Pratt Algorithm

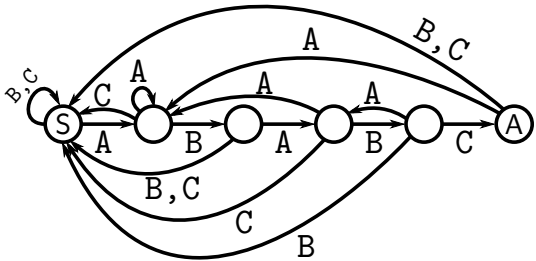
How to Compute the DFA Transition Function?

DFA transition function δ

- Matching character: just move right (easy)
- Non-matching character: move left (but how far?)



Example: Construction of DFA Transition Table (ABABC)



The *lps* Function Moves to the Next Smaller Active State

Given: state q and character c

Approach to compute $\delta(q, c)$

- If $c = P[q + 1]$, then $q \mapsto q + 1$.
- If not, try again for q', where q' is "the next NFA state that would be active" (if this was an NFA and not a DFA)

The *lps* Function Moves to the Next Smaller Active State

Given: state q and character c

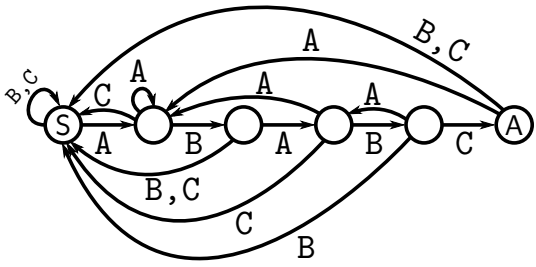
Approach to compute $\delta(q, c)$

- If $c = P[q+1]$, then $q \mapsto q+1$.
- If not, try again for q', where q' is "the next NFA state that would be active" (if this was an NFA and not a DFA)

The lps-function: $lps : \{0, \ldots, m-1\} \to \mathbb{N}$

- **Recall:** state q corresponds to prefix $P[\ldots q]$ of length $q+1$
- $lps(q)$ is the length of the **l**ongest **p**refix of P that is a true **s**uffix of $P[\ldots q]$.
- Then, we can get q' through $q' = lps(q) - 1$.

Example: *lps*-Table (ABABC)



Using lps to Compute δ on-the-fly

```
def DFA_delta_lps(q, c, P, lps):
    """state q, character c, pattern P, lps function/table"""
    m = len(P)
    while q == m-1 or (P[q+1] != c and q > -1):    # line 4
        q = lps[q] - 1                             # line 5
    if P[q+1] == c:  q += 1                        # line 6
    return q
```

Using lps to Compute δ on-the-fly

```
def DFA_delta_lps(q, c, P, lps):
    """state q, character c, pattern P, lps function/table"""
    m = len(P)
    while q == m-1 or (P[q+1] != c and q > -1):    # line 4
        q = lps[q] - 1                             # line 5
    if P[q+1] == c:  q += 1                        # line 6
    return q
```

Running time analysis of one step

- Still takes $O(m)$ time in the worst case (`while` loop in line 4), leading to $O(mn)$ time for pattern matching over the whole text.

Using lps to Compute δ on-the-fly

```
def DFA_delta_lps(q, c, P, lps):
    """state q, character c, pattern P, lps function/table"""
    m = len(P)
    while q == m-1 or (P[q+1] != c and q > -1):    # line 4
        q = lps[q] - 1                             # line 5
    if P[q+1] == c:  q += 1                        # line 6
    return q
```

Running time analysis of one step

- Still takes $O(m)$ time in the worst case (`while` loop in line 4), leading to $O(mn)$ time for pattern matching over the whole text.
- But m iterations cannot happen for all n text characters!
  Amortized analysis: How many times can lines 4, 5 be executed in total?

Using lps to Compute δ on-the-fly

```
def DFA_delta_lps(q, c, P, lps):
    """state q, character c, pattern P, lps function/table"""
    m = len(P)
    while q == m-1 or (P[q+1] != c and q > -1):    # line 4
        q = lps[q] - 1                             # line 5
    if P[q+1] == c:  q += 1                        # line 6
    return q
```

Running time analysis of one step

- Still takes $O(m)$ time in the worst case (`while` loop in line 4), leading to $O(mn)$ time for pattern matching over the whole text.
- But m iterations cannot happen for all n text characters!
  Amortized analysis: How many times can lines 4, 5 be executed in total?
- Line 5 decreases q, but q cannot drop below -1.
  Only line 6 can ever increase q, at most once per iteration.

Knuth-Morris-Pratt Algorithm

```
def DFA_delta_lps(q, c, P, lps):
    """state q, character c, pattern P, lps function/table"""
    m = len(P)
    while q == m-1 or (P[q+1] != c and q > -1):
        q = lps[q] - 1
    if P[q+1] == c:  q += 1
    return q

def KMP(P, T):
    lps = compute_lps(P)  # to show: O(m) time
    m, q = len(P), -1
    for i in range(len(T)):  # O(n) iterations
        q = DFA_delta_lps(q, T[i], P, lps)  # amortized O(1)
        if q == m - 1: yield i
```

Knuth-Morris-Pratt Algorithm

```
def DFA_delta_lps(q, c, P, lps):
    """state q, character c, pattern P, lps function/table"""
    m = len(P)
    while q == m-1 or (P[q+1] != c and q > -1):
        q = lps[q] - 1
    if P[q+1] == c:  q += 1
    return q

def KMP(P, T):
    lps = compute_lps(P)  # to show: O(m) time
    m, q = len(P), -1
    for i in range(len(T)):  # O(n) iterations
        q = DFA_delta_lps(q, T[i], P, lps)  # amortized O(1)
        if q == m - 1: yield i
```

Running time: $O(n+m)$ since `DFA_delta_lps` takes **amortized** constant time.

Computing the lps Table

Idea

The lps table is computed by matching P against itself.

```
def compute_lps(P):
    m, q = len(P), -1
    lps = [0] * m  # [0,0,...,0]; lps[0] = 0 is correct
    for i in range(1, m):  # compute lps[i]
        while q > -1 and P[q+1] != P[i]:
            q = lps[q] - 1
        if P[q+1] == P[i]:  q += 1
        lps[i] = q+1  # Invariant (Q) holds here
    return lps
```

Computing the lps Table

Idea

The lps table is computed by matching P against itself.

```
def compute_lps(P):
    m, q = len(P), -1
    lps = [0] * m  # [0,0,...,0]; lps[0] = 0 is correct
    for i in range(1, m):  # compute lps[i]
        while q > -1 and P[q+1] != P[i]:
            q = lps[q] - 1
        if P[q+1] == P[i]:  q += 1
        lps[i] = q+1  # Invariant (Q) holds here
    return lps
```

Invariant (Q): $q = \max\{k < i : P[i-k \ldots i] = P[0 \ldots k]\}$

Summary: Knuth-Morris-Pratt Algorithm

Knuth-Morris-Pratt Algorithm

- lps-function gives a succinct representation of δ.
- Using lps to evaluate δ takes **amortized** constant time.
- Constructing lps-table works similarly and takes $O(m)$ time.
- KMP algorithm has optimal running time of $O(m + n)$.

Historical Note

In the original paper (1977), the algorithm is not presented in terms of DFAs.
However, the authors point out that
"automata theory had actually been helpful in this practical problem."

The Aho-Corasick Algorithm: Sets of Patterns

Searching a Set of Patterns

Setting

- Instead of a single pattern, we consider a **set of patterns** $P = \{P^1, \ldots, P^K\}$, such as {Meier, Meyer, Maier, Mayer}.
- The patterns may have different lengths, e.g. $|P^i| \neq |P^j|$
- Let $m_k := |P^k|$ for $1 \leq k \leq K$, and $m := \sum_k m_k$.

Task

Efficiently find all instances of P in a text T.
Output: all triples (i, j, k) such that $T[i, j] = P^k$.

Searching a Set of Patterns

Setting

- Instead of a single pattern, we consider a **set of patterns** $P = \{P^1, \ldots, P^K\}$, such as {Meier, Meyer, Maier, Mayer}.
- The patterns may have different lengths, e.g. $|P^i| \neq |P^j|$
- Let $m_k := |P^k|$ for $1 \leq k \leq K$, and $m := \sum_k m_k$.

Task

Efficiently find all instances of P in a text T.
Output: all triples (i, j, k) such that $T[i, j] = P^k$.

Naive Solution

- Run Knuth-Morris-Pratt (or another algorithm) separately for every P^i.
- Multiple KMP has runnning time $O(m + Kn)$. Can we do it in $O(m + n)$?

Aho-Corasick Algorithm

Basic idea

The basic idea is the same as for KMP algorithm:

- Keep track of the **longest suffix of the text read so far** that is a **prefix of a pattern** in the set P.
- **Needed:** data structure allowing us to maintain this invariant, using constant time per character.
- **Solution:** use a **trie** (DFA in principle), plus lps-like links.

Tries

Definition (Trie)

A **trie** over a finite set of words $S \subset \Sigma^+$ is an edge-labeled tree
over the **node set** $\mathit{prefixes}(S)$ (all prefixes of all words in S)
with the following features:

- Node s is a child of t if and only if $s = ta$ for one $a \in \Sigma$;
  the edge $t \to s$ is then labeled with a.
  We identify every node v with concatenated edge labels from root to v.
- Outgoing edge labels are disjoint
  (we cannot have two outgoing edges labeled with the same character).

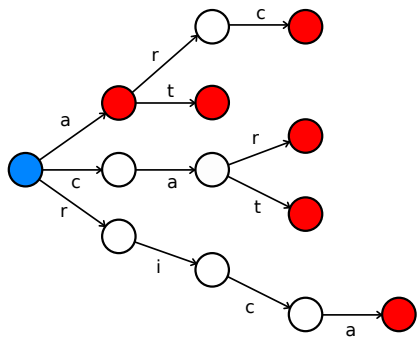
Alternative names: **prefix trie**, **pattern trie** or **keyword trie**.

Tries - Example

Trie built over the pattern set $P = \{\texttt{cat}, \texttt{car}, \texttt{arc}, \texttt{rica}, \texttt{at}, \texttt{a}\}$

- Root node is blue.
- States that match explicit words from P are red.

How can we use this trie to do pattern search in **optimal time** $O(n + m)$?



Aho-Corasick-Algorithm: *lps*

Idea

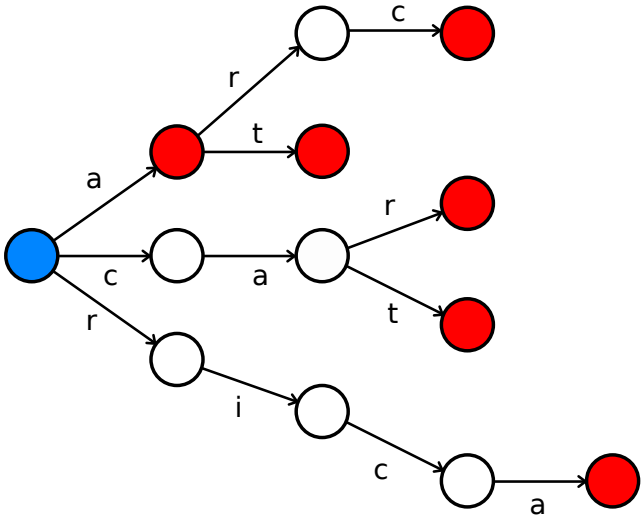
- Trie is like an NFA.
- Transform it to equivalent DFA.
- As for KMP, define DFA transition function δ implicitly, using *lps*.
- Define the *lps* function on **states** (strings; not numbers as in KMP).

Definition

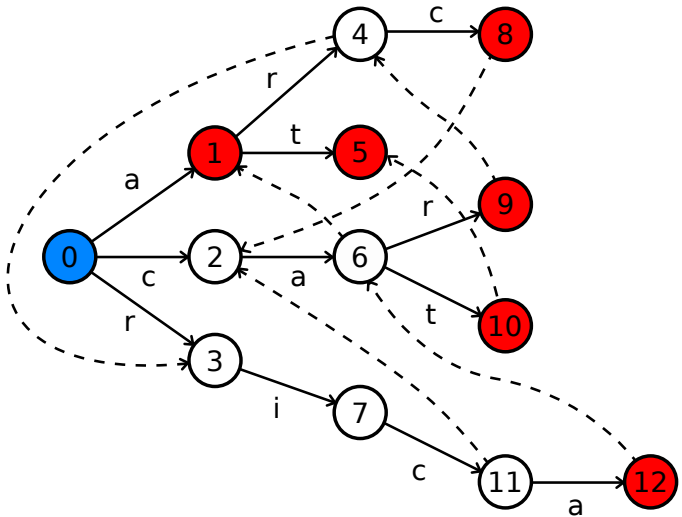
Let q be both a node and its label (stiring) of the trie.
We define

$$lps(q) := \arg\max_{p} \{|p| \ : \ p \in prefixes(P),\ |p| < |q|,\ p = q[(|q| - |p|) :]\}$$

Example: Aho-Corasick *lps* function



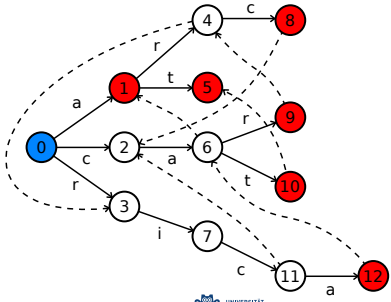
Example: Aho-Corasick *lps* function



Computing the *lps* function

Enumerate nodes via breadth-first search (BFS), starting with the root node ϵ:

- Depth 0: $lps[\epsilon]$ is not defined
- Depth 1: $lps[c] = \epsilon$ for all c of depth 1
- Depth $j \geq 2$: In node va, check if node $lps[v]$ has an edge a.
  If yes, set $lps[va] := lps[v]a$.
  If not, move to $v := lps[v]$ till either $lps[v]$ has an edge a, or does not exist.



Aho-Corasick Algorithm: *output* function

For every node v, we need to know the subset of P that ends in v.

Definition: Output function

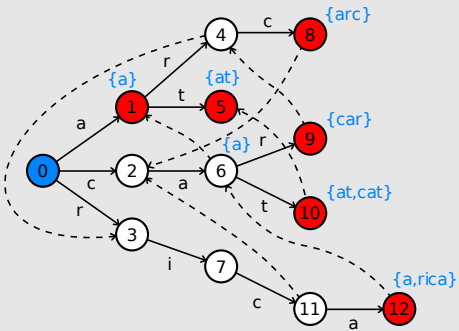
The output function is the mapping $output : Q = prefixes(P) \to 2^P$,
$v \mapsto output(v) := \{p \in P : p \text{ is a suffix of } v\}$.

Aho-Corasick Algorithm: *output* function

For every node v, we need to know the subset of P that ends in v.

Definition: Output function

The output function is the mapping $output : Q = prefixes(P) \to 2^P$,
$v \mapsto output(v) := \{p \in P : p \text{ is a suffix of } v\}$.



- Output pattern sets are shown near nodes in blue font.
- The output function can be implemented by condensing *lps* links.
- It can be constructed in $O(m)$ time and space.

Aho-Corasick Algorithm

Input: pattern set P, text T

Preprocessing of P:

- Build trie from P in $O(m)$ time
- Compute the *lps* function (BFS) in $O(m)$ time
- Compute the *output* function in $O(m)$ time

Searching P in T:

- $v \leftarrow \epsilon$ (start in the root)
- For each character c in T:
  - Attempt $v \leftarrow vc$.
    If not possible, set $v \leftarrow lps[v]$ and try again, as often as needed.
  - Report $output[v]$

Analysis of the Aho-Corasick Algorithm

Running time

The total running time is $O(m + n + z)$,
where z is the number of matches of P in T (triples (i, j, k)):

- $O(m)$ for preprocessing (building trie and lps, output functions).
- $O(1)$ per text character for state transitions, amortized analysis.
- $O(z)$ from constant time per successful match (output).

Analysis of the Aho-Corasick Algorithm

Running time

The total running time is $O(m + n + z)$,
where z is the number of matches of P in T (triples (i, j, k)):

- $O(m)$ for preprocessing (building trie and lps, output functions).
- $O(1)$ per text character for state transitions, amortized analysis.
- $O(z)$ from constant time per successful match (output).

Historical Notes

- The original Knuth-Morris-Pratt paper appeared in 1977.
- The Aho-Corasick algorithm appeared in 1975.
  Aho, Alfred V.; Corasick, Margaret J. (June 1975).
  "Efficient string matching: An aid to bibliographic search". Communications of the ACM.
- The unix tool `fgrep` and BLAST use the Aho-Corasick automaton (trie).

Summary

- **Subset construction**: general method to transform NFA to equivalent DFA.
- Pattern Search DFA: same number of states as NFA (maximal active state determines all other active states).
- DFA transition function δ: computable in $O(m^2)$ from NFA, improvement by using lps function (**Knuth-Morris-Pratt**) to $O(m)$ time.
- The **lps function** is a compact implicit representation of the δ table that allows to compute δ on the fly in **amortized** constant time.
- KMP has running time $O(m + n)$ in the best and worst case.
- Generalization to a set of patterns: **Aho-Corasick** automaton: lps function between states (prefixes).

Possible exam questions

- Explain the subset construction (from NFA to DFA).
- Why do Pattern Search DFAs have the same number of states as the NFAs?
- Explain the Knuth-Morris-Pratt (KMP) algorithm and its relation to DFAs.
- How can we construct the *lps* function and in what time?
- What is the running time of KMP (worst-case, best-case)?