Information Retrieval

Christian Rinderknecht

31 October 2008

Focus of this course

The most popular application of Information Retrieval is searching information on the World Wide Web, by means of **search engines** as Naver, Google, Yahoo etc.

There are other applications, like **document databases** (e.g., as found in libraries) and **text searches**, always relying on **complex mathematical models** (e.g., based on probability theory) to assess the *relevance* of the retrieved information or the *efficiency* of the retrieving algorithms.

In this course, however, we shall focus exclusively on text search, because it is the simplest way to start the study of this wide topics.

Focus of this course (cont)

Text search, in its simplest form, refers to the search of the occurrences of a word in a text, i.e. determine whether a word occurs in a given text and, if so, where.

All text editors include such a feature, as well as scripting languages as Perl and awk.

There is a famous utility called `grep` under UNIX which allows to search a set of strings in a text very efficiently.

Focus of this course (cont)

A **text** is nothing else than a string of characters and a word is a special case of a **pattern**.

In general, a pattern can describe a set of words, not only a single word. For example, the pattern `abra.*bra` defines the set of words starting by `abra` and ending by `bra`.

The efficiency of a search algorithm is usually measured by **the number of letter comparisons**: the lower, the better.

Basic definitions

An **alphabet** is a finite set whose elements are called **letters**.

A **string** is a sequence of letters of an alphabet. A **text** and a **word** are other names for strings, depending on the context. The empty string is noted ϵ.

The **length** of a string x is the length of the associated sequence and is noted $|x|$.

Basic definitions (cont)

The sequence of letters is written simply by enumerating in order the letters, like `abraca`.

For $i = 1, 2, \ldots, |x|$, we note $x[i]$ the letter at **index** (or position) i in x. For example, if $x = \texttt{abba}$ then $x[1] = x[4] = \texttt{a}$.

From the previous definitions, we have $x = x[1]x[2]\ldots x[|x|]$ for all non-empty string x.

Basic definitions (cont)

We can define the equality of two strings using the equality of two letters: by definition $x = y$ if $x = y = \epsilon$ or $|x| = |y|$ and then for all i such that $1 \leqslant i \leqslant |x|$, we have $x[i] = y[i]$.

The **product** or **concatenation** of two strings x and y is the string composed of the letters of x followed by the letters of y. It is noted xy or $x \cdot y$.

Property $x \cdot \epsilon = \epsilon \cdot x = x$ holds for all strings x.

Basic definitions (cont)

A word x is a **factor** of a word y if there exists two words u and v such that $y = uxv$.

Then there exists a position i such that $x = y[i]y[i+1]\ldots y[i+|x|-1]$, noted more simply as $x = y[i\ldots i+|x|-1]$. One says that x **occurs** in y.

Also, when $y = xv$, then x is a **prefix** of y, noted $x \preccurlyeq y$.

When $y = ux$, then x is a **suffix** of y. A factor x of y such that $x \neq y$ is a **proper factor**. We note $x \prec y$ if x is a proper prefix of y.

Basic definitions (cont)

The ***n*-th power** of word x is defined as $x^0 = \epsilon$ and $x^{n+1} = x^n x$, for all $n \geqslant 0$. This notation denotes the repetition of a word. So, for example, if $x = \mathtt{abb}$, then

$$\begin{aligned}
x^0 &= \epsilon \\
x^1 &= x^0 x = \varepsilon x = x = \mathtt{abb} \\
x^2 &= x^1 x = (x^0 x)x = xx = (\mathtt{abb})(\mathtt{abb}) = \mathtt{abbabb} \\
x^3 &= x^2 x = (x^1 x)x = ((x^0 x)x)x = xxx \\
&= (\mathtt{abb})(\mathtt{abb})(\mathtt{abb}) = \mathtt{abbabbabb}
\end{aligned}$$

It is good to remember a similar concept (power) and notation for functions. If f is a function from a set onto itself, then, for all $n \geqslant 0$

$$f^0(x) = x \qquad f^{n+1}(x) = f^n(f(x))$$

Basic definitions (cont)

So, for example, if $f(x) = x + 1$, then

$$\begin{aligned} f^0(x) &= x \\ f^1(x) &= f^0(f(x)) = f(x) = x + 1 \\ f^2(x) &= f^1(f(x)) = f^0(f(f(x))) = f(f(x)) = f(x + 1) = (x + 1) + 1 \\ &= x + 2 \end{aligned}$$

Note that it was possible to define the power of a function f as

$$f^0(x) = x \qquad f^{n+1}(x) = f(f^n(x))$$

It is possible to compose two different functions too. If f and g are two composable functions, then we define the composed function $f \circ g$ as

$$(f \circ g)(x) = f(g(x))$$

Naive search

Let t be a text and x a word such that $|x| \leqslant |t|$, both using the alphabet Σ.

Formally, the problem of text search is stated as follows: determine if x is a factor of t and, if so, give the position of the first letter of x in y.

The simplest algorithm, often called **naive**, consists in comparing x to $t[k+1 \ldots k+|x|]$ for all the $k \in \{0, 1, \ldots, |t| - |x|\}$ in the worst case.

It helps to imagine the text is an array of characters and the word we are looking up is sliding along this array until we find a match. This is referred to as the **sliding window**, because we can think of a frame sliding along the text.

Naive search/Algorithm

The naive search algorithm we just described can be more formally described as

```
NAIVE($x$, $t$)
$i \leftarrow 1$; $j \leftarrow 1$
while $i \leqslant |x|$ and $j \leqslant |t|$
    do if $t[j] = x[i]$                          ▷ Compare a letter of $x$ with a letter of $t$.
        then $j \leftarrow j + 1$; $i \leftarrow i + 1$   ▷ Schedule comparison with next letter in $x$.
        else $j \leftarrow j - i + 2$; $i \leftarrow 1$   ▷ Failed: we slide $x$ on $t$ and start again.
if $i > |x|$
   then ...                                      ▷ Occurrence of $x$ in $t$ at position $j - |x|$.
   else ...                                      ▷ No occurrences.
```

Naive search/Analysis

How many comparisons does the naive algorithm performs in the worst case?

In the worst case, x is not in t and the test at line 3 always fails on the last letter of x, leading to make x slide of one position at line 5 the latest as possible.

An example of positive match in the worst case is $t = \mathtt{a}^{m-1}\mathtt{b}$ and $x = \mathtt{a}^{n-1}\mathtt{b}$.

There are $n - m + 1$ positions in t to compare with the first letter of x, and since there are m letters in x, it makes a total of $(n - m + 1)m = nm - m^2 + m < |t| \times |x|$ positions.

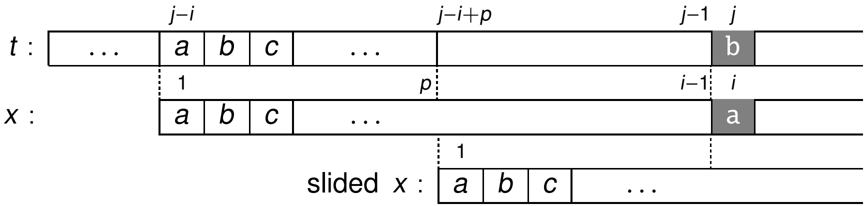
Morris-Pratt algorithm

The slowness of the naive algorithm in the worst case is due to the fact that, in case of failure, it starts again to compare the first letters of x etc. (see line 5), *without using the information of the partial success of the previous attempt.*

If the failure occurred at the index i in the word x and index j in the text, we have

$$x[1 \ldots i-1] = t[j-i \ldots j-1]$$
$$x[i] \neq t[j]$$

Morris-Pratt algorithm (cont)



Any subsequent search in the factor $t[j-i\dots j-1]$ could use the information that it is a prefix of x — the first comparison failure is coloured in grey.

If a new search starts in t at position $j-i+p$, with $2 \leqslant p \leqslant i-1$, it compares $x[1\dots]$ to $t[j-i+p\dots]$. Since $t[j-i+p\dots j-1]$ is a suffix of $t[j-i\dots j-1]$, which is equal to $x[1\dots i-1]$, it compares $x[1\dots]$ to a suffix of $x[1\dots i-1]$, i.e., to a part of itself!

Morris-Pratt algorithm (cont)

The important point here is that these comparisons are **independent** of the text t, since they apply to part of the pattern itself, i.e., the word x.

The strategy devised here is to compute all the comparisons on x *before* we start the search in the text, and then use this information to avoid repeating the same comparisons at different positions.

This **preprocessing** on x will indeed allow to recognise the sole configurations where the search may succeed.

Therefore the subproblem now is to determine the positions i where the word $x[1 \dots i-1]$ ends with a prefix of x.

Morris-Pratt algorithm/Maximum borders

Let u be a non-empty word. A **border** of u is a word different from u which is both prefix and suffix of u. For example, the word $u = \texttt{abacaba}$ has the three borders ϵ, $\texttt{a}$ and $\texttt{aba}$.

Let us note $\textsc{Border}(x)$ the **longest border** of a non-empty word x.

For a given word x, let us define a function β_x on all its prefixes as

$$\beta_x(i) = |\textsc{Border}(x[1 \ldots i])| \qquad 1 \leqslant i \leqslant |x|$$

or, equivalently

$$\beta_x(|y|) = |\textsc{Border}(y)| \qquad y \leqslant x$$

Morris-Pratt algorithm/Pattern preprocessing (cont)

Here is the table of the maximum borders for the prefixes of the word abacabac:

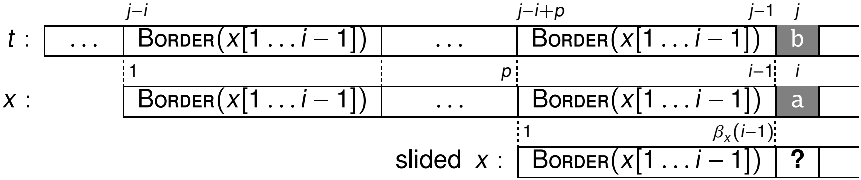
x	a	b	a	c	a	b	a	c
i	1	2	3	4	5	6	7	8
Border(y)	ϵ	ϵ	a	ϵ	a	ab	aba	abac
$\beta_x(i)$	0	0	1	0	1	2	3	4

Each row corresponds to a prefix y of x, not just to a letter. So, at index 7, you should read Border($\underline{\text{aba}}\text{c}\underline{\text{aba}}$) $=$ aba, at index 4, Border(abac) $= \varepsilon$ because abac $= \underline{\varepsilon}\text{abac}\underline{\varepsilon}$.

Also, note that borders can overlap, e.g.,. aaaa $= \underline{\text{aaa}}\text{a} = \text{a}\underline{\text{aaa}}$, so Border(aaaa) $=$ aaa.

Morris-Pratt algorithm/Pattern preprocessing (cont)

Coming back to the naive search:



$j{-}i$ $j{-}i{+}p$ $j{-}1$ j

t : … $\textsc{Border}(x[1\ldots i-1])$ … $\textsc{Border}(x[1\ldots i-1])$ b

1 p $i{-}1$ i

x : $\textsc{Border}(x[1\ldots i-1])$ … $\textsc{Border}(x[1\ldots i-1])$ a

1 $\beta_x(i{-}1)$

slided x : $\textsc{Border}(x[1\ldots i-1])$ **?**

the difference here is that, in case of failure (in grey), we do **not** decrement the position in t and we slide x of $p = i - 1 - \beta_x(i-1)$ positions with respect to t, i.e., the current position in x does not decrement to 1 but becomes $1 + \beta_x(i-1)$.

Morris-Pratt algorithm/Pattern preprocessing (cont)

What happens in the extreme case $i = 1$, that is, when the comparison fails on the first letter of the pattern?

In this case, there is no optimisation left, we just can try our luck by comparing the next position in t to the first letter of x (again): this is exactly what the naive algorithm would do.

This means that the sliding offset must equal one when $i = 1$:

$$i - 1 - \beta_x(i - 1) = 1 \quad \text{where } i = 1$$

which is equivalent to

$$\beta_x(0) = -1$$

Therefore we need to extend β to be defined on 0 as $\beta_x(0) = -1$ for all words x.

Morris-Pratt algorithm/Pattern preprocessing (cont)

In practice, it is convenient to use a function $s_x : \{1, \dots, |x|\} \to \{0, \dots, |x|\}$ defined as

$$s_x(i) = 1 + \beta_x(i-1) \qquad 1 \leqslant i \leqslant |x|$$

Then the sliding consists simply in replacing i by $s_x(i)$.

This function s_x is called the **failure function** of pattern x. For the word abacabac, it is

x		a	b	a	c	a	b	a	c
i	0	1	2	3	4	5	6	7	8
$\beta_x(i)$	−1	0	0	1	0	1	2	3	4
$s_x(i)$	□	0	1	1	2	1	2	3	4

Morris-Pratt algorithm/Imperative pseudo-code

```
MP($x, t$)
$s \leftarrow$ MP-FAIL($x$)                      ▷ The failure function as an array.
$i \leftarrow 1$; $j \leftarrow 1$
while $i \leqslant |x|$ and $j \leqslant |t|$
    do if $i = 0$ or $x[i] = t[j]$               ▷ Test on $i$ because now $i$ can be 0.
          then $i \leftarrow i + 1$; $j \leftarrow j + 1$   ▷ Schedule next letter in $x$.
          else $i \leftarrow s[i]$                ▷ Failed: we slide $x$ on $t$.
if $|x| < i$
   then ...                                      ▷ Occurrence of $x$ in $t$ at $j - |x|$.
   else ...                                      ▷ No occurrences.
```

Morris-Pratt algorithm/Single-assignment pseudo-code

```
MP($x, t$)
$s \leftarrow$ MP-FAIL($x$)
$(i, j) \leftarrow$ OFFSETS($s, x, t, 1, 1$)
if $|x| < i$
    then ... else ...
```

```
OFFSETS($s, x, t, i, j$)
if $i \leqslant |x|$ and $j \leqslant |t|$
    then if $i = 0$ or $x[i] = t[j]$
            then OFFSETS $\leftarrow$ OFFSETS($s, x, t, i + 1, j + 1$)
            else OFFSETS $\leftarrow$ OFFSETS($s, x, t, s[i], j$)
    else OFFSETS $\leftarrow (i, j)$
```

Morris-Pratt algorithm/Analysis and example

The Morris-Pratt algorithm finds an occurrence of a word x in a text t, or fail to do so, in at most $2|t| - 1$ letter comparisons, if we already have the failure function on x.

Example. Consider the following table for the search of the word $x = \mathtt{abacabac}$ in the text $t = \mathtt{babacacabacaab}$.

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	b	a	b	a	c	a	c	a	b	a	c	a	a	b
i	1	1	2	3	4	5	6	1	2	3	4	5	6	2
	0						2						2	
							1						1	
							0							

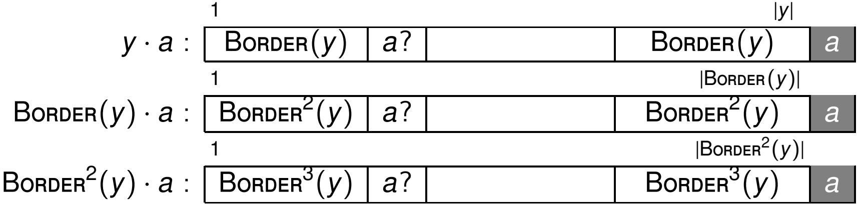
Morris-Pratt algorithm/Example

	1	2	3	4	5	6	7	8	9	10	11	12	13	14						
t :	b	a	b	a	c	a	c	a	b	a	c	a	a	b						
x :	a	b	a	c	a	b	a	c												
		a	b	a	c	a	b	a	c											
						a	b	a	c	a	b	a	c							
							a	b	a	c	a	b	a	c						
								a	b	a	c	a	b	a	c					
												a	b	a	c	a	b	a	c	
													a	b	a	c	a	b	a	c

Morris-Pratt algorithm/Maximum borders (cont)

We need to show how to compute the function β or, equivalently, the failure function s. Both rely on the computation of maximum-length borders.

How do we compute the longest borders?



$y \cdot a$: $\text{Border}(y)$ | a? | | $\text{Border}(y)$ | a — positions 1 … $|y|$

$\text{Border}(y) \cdot a$: $\text{Border}^2(y)$ | a? | | $\text{Border}^2(y)$ | a — positions 1 … $|\text{Border}(y)|$

$\text{Border}^2(y) \cdot a$: $\text{Border}^3(y)$ | a? | | $\text{Border}^3(y)$ | a — positions 1 … $|\text{Border}^2(y)|$

Morris-Pratt algorithm/Maximum borders (cont)

Let us find the longest border of a word ya, i.e., $\textsc{Border}(ya)$, where a is a letter.

The idea is to, recursively, consider $\textsc{Border}(y) \cdot a$.

If $\textsc{Border}(y) \cdot a$ is a prefix of y, then $\textsc{Border}(ya) = \textsc{Border}(y) \cdot a$.

Otherwise, we must consider the border of the border of y, i.e., $\textsc{Border}^2(y) \cdot a$, instead of $\textsc{Border}(y) \cdot a$ and repeat this process, until $\textsc{Border}^k(y) \cdot a$ is a prefix of y of $\textsc{Border}^k(y)$ is empty.

Morris-Pratt algorithm/Maximum borders (cont)

This can be formally summarised as follows. For all word $y \neq \epsilon$ and all letter a:

$$\textsc{Border}(a) = \epsilon$$

$$\textsc{Border}(y \cdot a) = \begin{cases} \textsc{Border}(y) \cdot a & \text{if } \textsc{Border}(y) \cdot a \preccurlyeq y \\ \textsc{Border}(\textsc{Border}(y) \cdot a) & \text{otherwise} \end{cases}$$

For example, let $y = \mathtt{ababbb}$ and assume $\textsc{Border}(y) = \epsilon$.
Then $\textsc{Border}(y \cdot \mathtt{a}) = \textsc{Border}(y) \cdot \mathtt{a} = \mathtt{a}$ because $\mathtt{a} \preccurlyeq y$. Other examples:

$$\begin{array}{lll} y = \mathtt{babbaa} & \textsc{Border}(y) = \epsilon & \textsc{Border}(y \cdot \mathtt{a}) = \textsc{Border}(\epsilon \cdot \mathtt{a}) = \epsilon \\ y = \mathtt{abaaab} & \textsc{Border}(y) = \mathtt{ab} & \textsc{Border}(y \cdot \mathtt{a}) = \textsc{Border}(y) \cdot \mathtt{a} = \mathtt{aba} \\ y = \mathtt{abbaab} & \textsc{Border}(y) = \mathtt{ab} & \textsc{Border}(y \cdot \mathtt{a}) = \textsc{Border}(\mathtt{ab} \cdot \mathtt{a}) = \mathtt{a} \end{array}$$

Morris-Pratt algorithm/Maximum borders (cont)

Let us prove that, when we take the border of a word, we get a prefix of this word, i.e.,

$$\textsc{Border}(y) \prec y \qquad y \neq \varepsilon$$

First, the property holds on words of length 1 since, by definition,

$$\textsc{Border}(a) = \varepsilon \prec a$$

Assume now that the property holds for all words of length lower or equal to n and let us prove that it holds for words of length $n + 1$.

Morris-Pratt algorithm/Maximum borders (cont)

Let y be a word of length n, $1 \leqslant n$, and a letter a.

By definition, we have

$$\textsc{Border}(ya) = \begin{cases} \textsc{Border}(y) \cdot a & \text{if } \textsc{Border}(y) \cdot a \preccurlyeq y \\ \textsc{Border}(\textsc{Border}(y) \cdot a) & \text{otherwise} \end{cases}$$

So, if $\textsc{Border}(y) \cdot a \preccurlyeq y$, then

$$\textsc{Border}(ya) = \textsc{Border}(y) \cdot a \preccurlyeq y \prec ya$$

This is the property for words of length $n+1$ such that $\textsc{Border}(y) \cdot a \preccurlyeq y$.

Morris-Pratt algorithm/Maximum borders (cont)

Otherwise, if $\textsc{Border}(y) \cdot a \not\preccurlyeq y$ then

$$\textsc{Border}(ya) = \textsc{Border}(\textsc{Border}(y) \cdot a).$$

Since $\textsc{Border}(y) \prec y$ and $|y| = n$ then

$$\begin{aligned} |\textsc{Border}(y)| < |y| \Leftrightarrow |\textsc{Border}(y)| < n &\Leftrightarrow |\textsc{Border}(y)| + 1 < n + 1 \\ &\Leftrightarrow |\textsc{Border}(y)| + 1 \leqslant n \end{aligned}$$

Thus $|\textsc{Border}(y) \cdot a| = |\textsc{Border}(y)| + 1 \leqslant n$, and the inductive assumption holds:

$$\begin{aligned} \textsc{Border}(\textsc{Border}(y) \cdot a) &\prec \textsc{Border}(y) \cdot a \\ \textsc{Border}(ya) &\preccurlyeq \textsc{Border}(y) \prec y \prec ya \end{aligned}$$

This is the property for words of length $n + 1$ such that $\textsc{Border}(y) \cdot a \not\preccurlyeq y$, so this achieves the proof.

Morris-Pratt algorithm/Maximum borders (cont)

Let us try now the following.

$\textsc{Border}(ya)$

$$\begin{aligned}
&= \textsc{Border}(\textsc{Border}(y) \cdot a) && \textsc{Border}(y) \cdot a \not\preccurlyeq y \\
&= \textsc{Border}(\textsc{Border}^2(y) \cdot a) && \textsc{Border}^2(y) \cdot a \not\preccurlyeq \textsc{Border}(y) \\
&= \ldots && \ldots \\
&= \textsc{Border}(\textsc{Border}^{k-1}(y) \cdot a) && \textsc{Border}^{k-1}(y) \cdot a \not\preccurlyeq \textsc{Border}^{k-2}(y)
\end{aligned}$$

and $\forall p \leqslant k - 2 . \textsc{Border}^p(y) \neq \varepsilon$.

Then there is a smallest k such that $\textsc{Border}^k(y) \cdot a \preccurlyeq \textsc{Border}^{k-1}(y)$ or $\textsc{Border}^{k-1}(y) = \varepsilon$.

Morris-Pratt algorithm/Pattern preprocessing (resumed)

We proved page 29, that

$$\textsc{Border}(y) \prec y \qquad y \neq \varepsilon$$

Therefore

$$\textsc{Border}^{k}(y) \prec \textsc{Border}^{k-1}(y) \prec y \qquad 2 \leqslant k$$

Then,

$$\textsc{Border}^{k}(y) \cdot a \preccurlyeq \textsc{Border}^{k-1}(y) \iff \textsc{Border}^{k}(y) \cdot a \preccurlyeq y \qquad 1 \leqslant k$$

(Make a picture to convince yourself.)

Morris-Pratt algorithm/Pattern preprocessing (resumed)

If $\textsc{Border}^{k}(y) \cdot a \preccurlyeq y$ and $\textsc{Border}^{k-1}(y) \neq \varepsilon$ then, using the definition

$$\textsc{Border}(ya) = \textsc{Border}(\textsc{Border}^{k-1}(y) \cdot a) = \textsc{Border}^{k}(y) \cdot a$$

Otherwise, if $\textsc{Border}^{k-1}(y) = \varepsilon$ then

$$\textsc{Border}(ya) = \textsc{Border}(\textsc{Border}^{k-1}(y) \cdot a) = \textsc{Border}(a) = \varepsilon$$

and we cannot have $\textsc{Border}^{k}(y) \cdot a \preccurlyeq y$ since $\textsc{Border}^{k}(y) = \textsc{Border}(\textsc{Border}^{k-1}(y))$ is then undefined (the empty word has no border). Finally, we can summarise all cases:

$$\textsc{Border}(ya) = \begin{cases} \textsc{Border}^{k}(y) \cdot a & \text{if } \textsc{Border}^{k}(y) \cdot a \preccurlyeq y \\ \varepsilon & \text{if } \textsc{Border}^{k-1}(y) = \varepsilon \end{cases}$$

Morris-Pratt algorithm/Pattern preprocessing (resumed)

Let us give a computational definition of function β_x. Let $ya \preccurlyeq x$.

$$\begin{aligned} \beta_x(|ya|) &= |\textsc{Border}(ya)| && \text{by substitution of } y \text{ by } ya \text{ in definition} \\ \beta_x(|y|+1) &= |\textsc{Border}(ya)| && \text{by property of length} \qquad (1) \end{aligned}$$

From the definition of β_x page 17, we deduce the corollary

$$\beta_x^k(|y|) = |\textsc{Border}^k(y)| \quad y \preccurlyeq x \qquad (2)$$

which we can prove by induction on k. The property is true for $k = 1$ (even $k = 0$), by definition. Then assume $\beta_x^k(|y|) = |\textsc{Border}^k(y)|$ and do

$$\begin{aligned} \beta_x^{k+1}(|y|) &= \beta_x^k(\beta_x(|y|)) = \beta_x^k(|\textsc{Border}(y)|) = |\textsc{Border}^k(\textsc{Border}(y))| \\ &= |\textsc{Border}^{k+1}(y)| \end{aligned}$$

which proves that the property holds for all k such that $1 \preccurlyeq k$.

Morris-Pratt algorithm/Pattern preprocessing (resumed)

For the sake of clarity, let us mimic the setting of figure page 19 by letting $|ya| = i$.

$\textsc{Border}^k(y) \cdot a \preccurlyeq y$

$$\begin{aligned}
&\iff y[|\textsc{Border}^k(y)| + 1] = a && \text{since } \textsc{Border}(y) \prec y\\
&\iff y[\beta_x^k(|y|) + 1] = a && \text{by property (2)}\\
&\iff x[\beta_x^k(|y|) + 1] = x[|y| + 1] && \text{since } ya \preccurlyeq x\\
&\iff x[\beta_x^k(i-1) + 1] = x[i] && \text{since } |y| = i - 1 \qquad (3)
\end{aligned}$$

Pattern preprocessing (cont)

Let us consider again the recursive definition of Border at page 34:

$$\textsc{Border}(ya) = \begin{cases} \textsc{Border}^{k}(y) \cdot a & \text{if } \textsc{Border}^{k}(y) \cdot a \preccurlyeq y \\ \varepsilon & \text{if } \textsc{Border}^{k-1}(y) = \varepsilon \end{cases}$$

where k is the smallest non-zero integer such that $\textsc{Border}^{k}(y) \cdot a \preccurlyeq y$ or $\textsc{Border}^{k-1}(y) = \varepsilon$.

By taking the lengths of each side of the equations, it comes

$$|\textsc{Border}(ya)| = \begin{cases} |\textsc{Border}^{k}(y) \cdot a| & \text{if } \textsc{Border}^{k}(y) \cdot a \preccurlyeq y \\ |\varepsilon| & \text{if } \textsc{Border}^{k-1}(y) = \varepsilon \end{cases}$$

Pattern preprocessing (cont)

By equation (1) we get

$$\beta_x(|y|+1) = \begin{cases} |\textsc{Border}^k(y)|+1 & \text{if } \textsc{Border}^k(y)\cdot a \preccurlyeq y \\ 0 & \text{if } \textsc{Border}^{k-1}(y) = \varepsilon \end{cases}$$

Using equations (2) and (3):

$$\beta_x(|y|+1) = \begin{cases} \beta_x^k(|y|)+1 & \text{if } x[\beta_x^k(i-1)+1] = x[i] \\ 0 & \text{if } \beta_x^{k-1}(i-1) = 0 \end{cases}$$

Pattern preprocessing (cont)

Finally, taking into account the value in 0 and 1, we have, for $|x| = i \geqslant 2$

$$\beta_x(0) = -1 \qquad \beta_x(1) = 0$$

$$\beta_x(i) = \begin{cases} 1 + \beta_x^k(i-1) & \text{if } x[1 + \beta_x^k(i-1)] = x[i] \\ 0 & \text{if } \beta_x^{k-1}(i-1) = 0 \end{cases}$$

where k is the smallest nonzero integer such that $x[1 + \beta_x^k(i-1)] = x[i]$ or $\beta_x^{k-1}(i-1) = 0$.

Pattern preprocessing (cont)

From that definition, it is clear that β returns the value -1 only on 0.

Thus

$$\beta_x^{k-1}(i-1) = 0 \iff \beta_x(\beta_x^{k-1}(i-1)) = \beta_x(0) \iff \beta_x^k(i-1) = -1$$
$$\iff 1 + \beta_x^k(i-1) = 0$$

Then we can rewrite the previous definition of β_x as

$$\beta_x(0) = -1 \qquad \beta_x(i) = 1 + \beta_x^k(i-1) \quad 1 \leqslant i$$

where k is the smallest non-zero integer such that

- either $1 + \beta_x^k(i-1) = 0$
- or $x[1 + \beta_x^k(i-1)] = x[i]$

Pattern preprocessing/Imperative

It is then straightforward to write an algorithm for β_x, assuming $0 \leqslant i \leqslant |x|$:

```
Beta$(x, i)$
if $i = 0$
    then Beta $\leftarrow -1$                          ▷ $\beta_x(0) = -1$
    else $offset \leftarrow i - 1$
         repeat
                 $offset \leftarrow$ Beta$(x, offset)$
           until $offset = -1$ or $x[1 + offset] = x[i]$
         Beta $\leftarrow 1 + offset$                  ▷ $\beta_x(i) = 1 + \beta_x^k(i - 1)$
```

Pattern preprocessing/Single-assignment pseudo-code

```
Beta($x, i$)
if $i = 0$
    then Beta $\leftarrow -1$
    else Beta $\leftarrow 1 +$ Offset($x$, Beta($x, i - 1$))
```

```
Offset($x, offset$)
if $offset = -1$ or $x[1 + offset] = x[i]$
    then Offset $\leftarrow offset$
    else Offset $\leftarrow$ Offset($x$, Beta($x, offset$))
```

Pattern preprocessing (cont)

The problem here is inefficiency:

1. successive calls with the same arguments trigger again the same computations,
2. the recursive calls are expensive.

It is much preferable to have β as an array:

1. the values of β are computed only once each
2. and are accessed in constant time instead of relying on function calls.

Pattern preprocessing/Imperative

The idea is to compute all the values of $\beta_x(i)$ for increasing values of i:

```
MAKE-BETA($x$)
$\beta_x[0] \leftarrow -1$
for $i \leftarrow 1$ to $|x|$
    do $offset \leftarrow i - 1$
       repeat
              $offset \leftarrow \beta_x[offset]$
         until $offset = -1$ or $x[1 + offset] = x[i]$
       $\beta_x[i] \leftarrow 1 + offset$                ▷ $\beta_x(i) = 1 + \beta_x^k(i-1)$
MAKE-BETA $\leftarrow \beta_x$
```

Pattern preprocessing / Single-assignment pseudo-code

```
Make-Beta($x$)
$\beta_x[0] \leftarrow -1$
for $i \leftarrow 1$ to $|x|$
      do $\beta_x[i] \leftarrow 1 + $ Offset($x, \beta_x, \beta_x[i-1]$)
Make-Beta $\leftarrow \beta_x$
```

```
Offset($x, \beta_x, offset$)
if $offset = -1$ or $x[1 + offset] = x[i]$
   then Offset $\leftarrow offset$
   else Offset $\leftarrow$ Offset($x, \beta_x, \beta_x[offset]$)
```

Pattern preprocessing (cont)

In order to finish, we need to give the algorithm for the failure function s_x (see page 21). Recall that

$$s_x(i) = 1 + \beta_x(i-1) \quad 1 \leqslant i$$

We could then keep BETA and create a separate function MP-FAIL simply by sticking to the above definition:

```
MP-FAIL($\beta_x, i$)
MP-FAIL ← $1 + \beta_x[i-1]$          ▷ $s_x[i] \leftarrow 1 + \beta_x[i-1]$
```

But, again, this is not efficient: it would be better to precompute and store all the values of MP-FAIL in an array.

Pattern preprocessing (cont)

Instead, let us prove by induction on k that

$$s_x^k(i) = 1 + \beta_x^k(i-1) \quad 1 \leqslant i$$

The case $k = 1$ is the definition of s. Let us then assume that the property is true for all values from 1 to k. We have

$$s_x^{k+1}(i) = s_x^k(s_x(i)) = 1+\beta_x^k(s_x(i)-1) = 1+\beta_x^k(\beta_x(i-1)) = 1+\beta_x^{k+1}(i-1)$$

which proves the property for all $1 \leqslant k$.

Pattern preprocessing (cont)

Let us recall the definition of β_x (page 40):

$$\beta_x(0) = -1 \qquad \beta_x(i) = 1 + \beta_x^k(i-1) \qquad 1 \leqslant i$$

where k is the smallest non-zero integer such that

- either $1 + \beta_x^k(i-1) = 0$ or $x[1 + \beta_x^k(i-1)] = x[i]$

This becomes

$$s_x(1) = 0 \qquad s_x(1+i) = 1 + s_x^k(i) \qquad 1 \leqslant i$$

where k is the smallest non-zero integer such that

- either $s_x^k(i) = 0$ or $x[s_x^k(i)] = x[i]$

Pattern preprocessing/Imperative

```
MP-FAIL($x$)
$s_x[1] \leftarrow 0$
for $i \leftarrow 1$ to $|x| - 1$
    do $offset \leftarrow i$
       repeat
           $offset \leftarrow s_x[offset]$
       until $offset = 0$ or $x[offset] = x[i]$
       $s_x[1 + i] \leftarrow 1 + offset$
MP-FAIL $\leftarrow s$
```

Pattern preprocessing/Single-assignment pseudo-code

```
$\textsc{MP-Fail}(x)$
$s_x[1] \leftarrow 0$
**for** $i \leftarrow 1$ **to** $|x| - 1$
        **do** $s_x[1 + i] \leftarrow 1 + \textsc{Offset}(x, s_x, s_x[i])$
$\textsc{MP-Fail} \leftarrow s_x$
```

```
$\textsc{Offset}(x, s_x, \mathit{offset})$
**if** $\mathit{offset} = 0$ **or** $x[\mathit{offset}] = x[i]$
    **then** $\textsc{Offset} \leftarrow \mathit{offset}$
    **else** $\textsc{Offset} \leftarrow \textsc{Offset}(x, s_x, s_x[\mathit{offset}])$
```

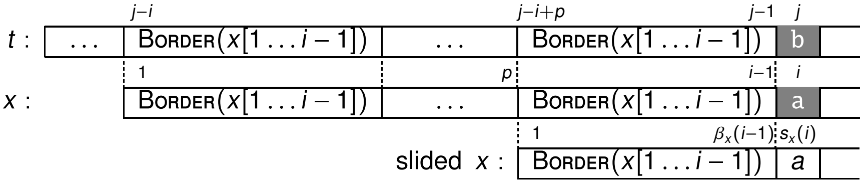
Analysis

The Morris-Pratt algorithm makes at most $2(|t| + |x| - 2)$ comparisons to find a word x in a text t or to fail.

This is much better than the naive algorithm whose cost was $(|t| - |x| + 1) \times |x|$.

Knuth-Morris-Pratt algorithm

The algorithm we present now is an improvement due to Knuth of the Morris-Pratt algorithm, based on avoiding situations which lead to certain letter comparison failures.



$j{-}i$ $\quad j{-}i{+}p$ $\quad j{-}1$ $\quad j$

t : $\dots$ $\textsc{Border}(x[1\dots i-1])$ $\dots$ $\textsc{Border}(x[1\dots i-1])$ b

1 $\quad p$ $\quad i{-}1$ $\quad i$

x : $\textsc{Border}(x[1\dots i-1])$ $\dots$ $\textsc{Border}(x[1\dots i-1])$ a

1 $\quad \beta_x(i{-}1)$ $\quad s_x(i)$

slided x : $\textsc{Border}(x[1\dots i-1])$ a

The observation is that if $a = \mathrm{a}$ then the sliding would lead to a comparison failure. So let us enforce that a sliding to compare $x[s_x(i)]$ to $t[j]$ is done if and only if $x[s_x(i)] \neq x[i]$.

Knuth-Morris-Pratt/Maximum disjoint borders

If $x[s_x(i)] = x[i]$, what should we do?

Let us note $y = x[1...i-1]$. We should consider successively $\textsc{Border}^2(y) \cdot \mathrm{a}$, $\textsc{Border}^3(y) \cdot \mathrm{a}$ etc., until we find a k such that $\textsc{Border}^k(y) \cdot \mathrm{a} \not\preccurlyeq x$ or $\textsc{Border}^k(y) = \varepsilon$. Such a border $\textsc{Border}^k(y)$ is called a **maximum disjoint border** of y in x.

If it is non-empty, we slide the word so that we compare $x[s_x^k(i)]$ and $t[j]$.

This disjointedness constraint can be precomputed on the pattern x *alone* before comparing it to the text t: all we have to do is to change the failure function s_x of the algorithm of Morris and Pratt and provide a new one. *The search algorithm itself does not change.*

Knuth-Morris-Pratt/Maximum disjoint borders (cont)

The disjointedness implicitly entails that it is a relative concept: we consider the disjoint border of a *proper prefix* of another word. Thus, the letter with follows the prefix is used as a constraint, i.e., it must not follow the disjoint border.

Let $ya \preccurlyeq x$. Let us note $\overline{\textsc{Border}}_x(y)$ the maximum disjoint border of y *in* x. In this case, the letter a is used to constrain the definition of the disjoint border, as we must have $\overline{\textsc{Border}}_x(y) \cdot a \not\preccurlyeq y$.

What if $y = x$, then? In this case, there is no right-context, like the letter a above, to constrain the maximum disjoint border. In this case, we still can take the maximum border, i.e., $\overline{\textsc{Border}}_y(y) = \textsc{Border}(y)$.

Knuth-Morris-Pratt/Maximum disjoint borders (cont)

More precisely, if the maximum border of y is not followed by a, then it is also the maximum disjoint border we are looking for. In other words:

$$\overline{\textsc{Border}}_x(y) = \textsc{Border}(y) \qquad \text{if } ya \preccurlyeq x \text{ and } \textsc{Border}(y) \cdot a \not\preccurlyeq y$$

If the maximum border of y is followed by a we must take the maximum disjoint border of the maximum border:

$$\overline{\textsc{Border}}_x(y) = \overline{\textsc{Border}}_x(\textsc{Border}(y)) \qquad \text{if } ya \preccurlyeq x \text{ and } \textsc{Border}(y) \cdot a \preccurlyeq y$$

By extension, if $x = y$, we take the maximum border:

$$\overline{\textsc{Border}}_y(y) = \textsc{Border}(y)$$

Knuth-Morris-Pratt/Maximum disjoint borders (cont)

In summary, assuming $ya \preccurlyeq x$ **and** $y \neq \varepsilon$

$$\overline{\textsc{Border}}_y(y) = \textsc{Border}(y)$$

$$\overline{\textsc{Border}}_x(y) = \begin{cases} \overline{\textsc{Border}}_x(\textsc{Border}(y)) & \text{if } \textsc{Border}(y) \cdot a \preccurlyeq y \\ \textsc{Border}(y) & \text{otherwise} \end{cases}$$

Knuth-Morris-Pratt/Maximum disjoint borders (cont)

By taking the length of each side of the equations,

$$|\overline{\textsc{Border}}_x(y)| = \begin{cases} |\textsc{Border}(y)| & \text{if } x = y \text{ or } \textsc{Border}(y) \cdot a \nleqslant y \\ |\overline{\textsc{Border}}_x(\textsc{Border}(y))| & \text{otherwise} \end{cases}$$

Let $\gamma_x(i)$ be the length of the disjoint maximum border of $x[1 \ldots i]$:

$$\gamma_x(i) = |\overline{\textsc{Border}}_x(x[1...i])| \qquad 1 \leqslant i \leqslant |x|$$

or, equivalently,

$$\gamma_x(|y|) = |\overline{\textsc{Border}}_x(y)| \qquad y \leqslant x$$

Knuth-Morris-Pratt/Maximum disjoint borders (cont)

Therefore

$$\gamma_x(|y|) = \begin{cases} \beta_x(|y|) & \text{if } x = y \text{ or } \textsc{Border}(y) \cdot a \not\preceq y \\ \gamma_x(|\textsc{Border}(y)|) & \text{otherwise} \end{cases}$$

that is to say

$$\gamma_x(|y|) = \begin{cases} \beta_x(|y|) & \text{if } x = y \text{ or } \textsc{Border}(y) \cdot a \not\preceq y \\ \gamma_x(\beta_x(|y|)) & \text{otherwise} \end{cases}$$

If $|y| = i$, we can write instead

$$\gamma_x(i) = \begin{cases} \beta_x(i) & \text{if } x = y \text{ or } \textsc{Border}(y) \cdot a \not\preceq y \\ \gamma_x(\beta_x(i)) & \text{otherwise} \end{cases}$$

Knuth-Morris-Pratt/Maximum disjoint borders (cont)

The condition can also be rewritten in terms of β_x and i, just as we did with the Morris-Pratt algorithm, pages 35 and 19. Let $|y| = i$, then

$$\begin{aligned} \textsc{Border}(y) \cdot a \not\preccurlyeq y &\iff x[\beta_x(i) + 1] \neq x[i+1] \\ x = y &\iff |x| = i \end{aligned}$$

So, finally, for $1 \leqslant i \leqslant |x|$,

$$\gamma_x(i) = \begin{cases} \beta_x(i) & \text{if } i = |x| \text{ or } x[\beta_x(i) + 1] \neq x[i+1] \\ \gamma_x(\beta_x(i)) & \text{otherwise} \end{cases}$$

which allows us to naturally extends γ_x on 0: $\gamma_x(0) = \beta_x(0) = -1$.

Knuth-Morris-Pratt/Maximum disjoint borders (cont)

Before going further, let us check by hand the following values of $\gamma_x(i)$ and compare them to $\beta_x(i)$: we must always have $\gamma_x(i) \leqslant \beta_x(i)$, since we plan an optimisation.

x		a	b	c	a	b	a	b	c	a	c
i	0	1	2	3	4	5	6	7	8	9	10
$\beta_x(i)$	−1	0	0	0	1	2	1	2	3	4	0
$\gamma_x(i)$	−1	0	0	−1	0	2	0	0	−1	4	0

One difference between β_x and γ_x is that, in the worst case, there is always an empty maximum border ε, i.e., $\beta_x(i) = 0$, whereas there may be no maximum disjoint border at all, i.e., $\gamma_x(i) = -1$. For instance, the prefix abc of x has an empty maximum border, i.e., $\beta_x(3) = 0$, but has no maximum disjoint border, i.e., $\gamma_x(3) = -1$, since $x[1] = x[4]$.

Knuth-Morris-Pratt/Maximum disjoint borders (cont)

This definition of γ_x relies on β_x, more precisely, the computation of $\gamma_x(i)$ requires $\beta_x^p(i)$, i.e., values $\beta_x(j)$ with $j < i$, since

$$\beta_x(0) = -1 \qquad \beta_x(i) = 1 + \beta_x^k(i-1) \qquad 1 \leqslant i$$

where k is the smallest non-zero integer such that

- either $1 + \beta_x^k(i-1) = 0$ or $x[1 + \beta_x^k(i-1)] = x[i]$

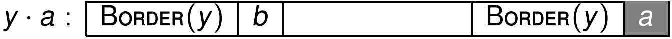
or, equivalently,

$$\textsc{Border}(ya) = \begin{cases} \textsc{Border}^k(y) \cdot a & \text{if } \textsc{Border}^k(y) \cdot a \preccurlyeq y \\ \varepsilon & \text{if } \textsc{Border}^{k-1}(y) = \varepsilon \end{cases}$$

where k is the smallest non-zero integer such that $\textsc{Border}^k(y) \cdot a \preccurlyeq y$ or $\textsc{Border}^{k-1}(y) = \varepsilon$.

Knuth-Morris-Pratt/Maximum disjoint borders (cont)

Therefore, let us find another definition of $\textsc{Border}(ya)$ which relies on $\textsc{Border}(y)$ but **not** on $\textsc{Border}^q(y)$ with $2 \leqslant q$. This can be achieved by considering again the figure



Indeed, if $a = b$ then $\textsc{Border}(ya) = \textsc{Border}(y)$. Else, $a \neq b$, and thus the maximum border can be found among the *disjoint* borders of $\textsc{Border}(y)$, otherwise $\textsc{Border}(ya) = \varepsilon$.

$$\textsc{Border}(ya) = \begin{cases} \overline{\textsc{Border}}_x^{\,q}(\textsc{Border}(y)) \cdot a & \text{if } \overline{\textsc{Border}}_x^{\,q}(\textsc{Border}(y)) \cdot a \leqslant y \\ \varepsilon & \text{if } y = \varepsilon \text{ or } \overline{\textsc{Border}}_x^{\,q-1}(\textsc{Border}(y)) = \varepsilon \end{cases}$$

where $ya \leqslant x$ and q is the smallest integer such that $\overline{\textsc{Border}}_x^{\,q}(\textsc{Border}(y)) \cdot a \leqslant y$ or $\overline{\textsc{Border}}_x^{\,q-1}(\textsc{Border}(y)) = \varepsilon$.

Knuth-Morris-Pratt/Maximum disjoint borders (cont)

By taking the lengths and letting $0 \leqslant i \leqslant |x| - 1$ and $|y| = i$,

$$\beta_x(0) = -1$$

$$\beta_x(i+1) = \begin{cases} \gamma_x^q(\beta_x(i)) + 1 & \text{if } x[\gamma_x^q(\beta_x(i)) + 1] = x[i+1] \\ 0 & \text{if } i = 0 \text{ or } \gamma_x^{q-1}(\beta_x(i)) = 0 \end{cases}$$

where q is the smallest integer such that

- either $x[\gamma_x^q(\beta_x(i)) + 1] = x[i+1]$
- or $\gamma_x^{q-1}(\beta_x(i)) = 0$

Knuth-Morris-Pratt/Maximum disjoint borders (cont)

Since $\gamma_x(i) = -1$ if and only if $i = 0$, we have

$$\begin{aligned}\gamma_x^{q-1}(\beta_x(i)) = 0 \iff \gamma_x(\gamma_x^{q-1}(\beta_x(i))) = \gamma_x(0) \iff \gamma_x^{q}(\beta_x(i)) = -1 \\ \iff \gamma_x^{q}(\beta_x(i)) + 1 = 0\end{aligned}$$

Therefore we can simplify the new definition of β_x as

$$\begin{aligned}\beta_x(0) &= -1 \\ \beta_x(1) &= 0 \\ \beta_x(i+1) &= \gamma_x^{q}(\beta_x(i)) + 1 \qquad 1 \leqslant i \leqslant |x| - 1\end{aligned}$$

where q is the smallest integer such that

- either $x[\gamma_x^{q}(\beta_x(i)) + 1] = x[i+1]$
- or $\gamma_x^{q}(\beta_x(i)) + 1 = 0$

Knuth-Morris-Pratt/Maximum disjoint borders/Imperative

```
GAMMA($x$)
$\gamma_x[0] \leftarrow -1$; $offset \leftarrow -1$                    $\triangleright$ $offset = \beta_x(0)$
for $i \leftarrow 1$ to $|x| - 1$
    do repeat
                $offset \leftarrow \gamma_x[offset]$
          until $offset = -1$ or $x[offset+1] = x[i]$
        $offset \leftarrow offset + 1$        $\triangleright$ $offset = \gamma_x^q(\beta_x(i-1)) + 1 = \beta_x(i)$
        if $i = |x|$ or $x[offset+1] = x[i+1]$
          then $\gamma_x[i] \leftarrow offset$                       $\triangleright$ $\gamma_x(i) = \beta_x(i)$
          else $\gamma_x[i] \leftarrow \gamma_x[offset]$               $\triangleright$ $\gamma_x(i) = \gamma_x(\beta_x(i))$
GAMMA $\leftarrow \gamma_x$
```

Knuth-Morris-Pratt/Better failure function

With the algorithm of Morris and Pratt, it was convenient to use a failure function $s(i) = 1 + \beta(i - 1)$. Similarly, we need here another failure function, noted r, such that $r(i) = 1 + \gamma(i - 1)$, for $1 \leqslant i$.

In order to finish, we need to give the algorithm for the failure function r.

We could then keep Beta and create a separate function KMP-Fail simply by sticking to the above definition:

```
KMP-Fail($\gamma_x, i$)
KMP-Fail $\leftarrow 1 + \gamma_x[i - 1]$                    $\triangleright$ $r_x[i] \leftarrow 1 + \gamma_x[i - 1]$
```

But this is not efficient: it would be better to precompute and store all the values of KMP-Fail in an array.

Knuth-Morris-Pratt/Better failure function (cont)

Let us recall the definition of γ_x. Let $1 \leqslant i \leqslant |x|$ and

$$\gamma_x(0) = -1$$

$$\gamma_x(i) = \begin{cases} \beta_x(i) & \text{if } i = |x| \text{ or } x[\beta_x(i)+1] \neq x[i+1] \\ \gamma_x(\beta_x(i)) & \text{otherwise} \end{cases}$$

Then, for $1 \leqslant i \leqslant |x| - 1$

$$r_x(1) = 1 + \gamma_x(0) = 0$$

$$r_x(1+i) = 1 + \gamma_x(i) = \begin{cases} 1 + \beta_x(i) & \text{if } x[\beta_x(i)+1] \neq x[i+1] \\ 1 + \gamma_x(\beta_x(i)) & \text{otherwise} \end{cases}$$

Knuth-Morris-Pratt/Better failure function (cont)

This can be slightly simplified into

$$r_x(1) = 0$$

$$r_x(1+i) = \begin{cases} r_x(1+\beta_x(i)) & \text{if } x[1+\beta_x(i)] = x[1+i] \\ 1+\beta_x(i) & \text{otherwise} \end{cases}$$

where for $1 \leqslant i \leqslant |x| - 1$.

Now, we need to express β_x in terms of r_x instead of γ_x.

Knuth-Morris-Pratt/Better failure function (cont)

First, let us prove by induction on $0 \leqslant p$ that

$$r_x^p(1+i) = 1 + \gamma_x^p(i) \qquad 0 \leqslant i \leqslant |x| - 1$$

Because

$$r_x^0(1+i) = 1 + \gamma_x^0(i) \Longleftrightarrow 1 + i = 1 + i$$

and, assuming it is true up to p, we have

$$r_x^{p+1}(1+i) = r_x(r_x^p(1+i)) = r_x(1+\gamma_x^p(i)) = 1+\gamma_x(\gamma_x^p(i)) = 1+\gamma_x^{p+1}(i)$$

which is the property at rank $p+1$.

Knuth-Morris-Pratt/Better failure function (cont)

So, the definition of β_x

$$\beta_x(0) = -1 \qquad \beta_x(1) = 0 \qquad \beta_x(1+i) = 1+\gamma_x^q(\beta_x(i)) \qquad 1 \leqslant i \leqslant |x|-1$$

where q is the smallest integer such that either $x[\gamma_x^q(1+\beta_x(i))] = x[1+i]$ or $1+\gamma_x^q(\beta_x(i)) = 0$, is equivalent to

$$\beta_x(0) = -1 \qquad \beta_x(1) = 0 \qquad \beta_x(1+i) = r_x^q(1+\beta_x(i)) \qquad 1 \leqslant i \leqslant |x|-1$$

where q is the smallest integer such that

- either $x[r_x^q(1+\beta_x(i))] = x[1+i]$
- or $r_x^q(1+\beta_x(i)) = 0$

Knuth-Morris-Pratt/Better failure function (cont)

Or, by changing $i+1$ into i,

$$\beta_x(0) = -1 \qquad \beta_x(1) = 0 \qquad \beta_x(i) = r_x^q(1+\beta_x(i-1)) \qquad 2 \leqslant i \leqslant |x|$$

where q is the smallest integer such that

- either $x[r_x^q(1+\beta_x(i-1))] = x[i]$
- or $r_x^q(1+\beta_x(i-1)) = 0$

Knuth-Morris-Pratt/Better failure function/Imperative

```
KMP-FAIL($x$)
$r_x[1] \leftarrow 0$; $offset \leftarrow 0$                      ▷ $offset = 1 + \beta_x(1 - 1)$
for $i \leftarrow 1$ to $|x| - 1$
    do repeat
                $offset \leftarrow r_x[offset]$
          until $offset = 0$ or $x[offset] = x[i]$
        $offset \leftarrow offset + 1$                            ▷ $offset = 1 + \beta_x(i)$
                                                 ▷ $offset = 1 + r_x^q(1 + \beta_x(i - 1))$
        if $x[offset] = x[1 + i]$
          then $r_x[i] \leftarrow r_x[offset]$                      ▷ $r_x(i) = r_x(1 + \beta_x(i))$
          else $r_x[i] \leftarrow offset$                            ▷ $r_x(i) = 1 + \beta_x(i)$
KMP-FAIL $\leftarrow r_x$
```

Knuth-Morris-Pratt/The code

The Knuth-Morris-Pratt algorithm is exactly the Morris-Pratt algorithm, except that we use a better failure function *r* instead of *s*:

```
KMP($x, t$)
$r \leftarrow$ KMP-FAIL($x$)
$i \leftarrow 1; j \leftarrow 1$
while $i \leqslant |x|$ and $j \leqslant |t|$
    do if $i = 0$ or $x[i] = t[j]$
          then $i \leftarrow i + 1; j \leftarrow j + 1$
          else $i \leftarrow r[i]$
if $|x| < i$
   then ...            ▷ Occurrence of $x$ in $t$ at position $j - |x|$.
   else ...                                          ▷ No occurrences.
```

Knuth-Morris-Pratt/The code (cont)

If we allow arguments to be functions, we can elegantly factorise the two text search algorithms into one:

```
Search($x, t, f$)
$i \leftarrow 1; j \leftarrow 1$
while $i \leqslant |x|$ and $j \leqslant |t|$
      do if $i = 0$ or $x[i] = t[j]$
            then $i \leftarrow i + 1; j \leftarrow j + 1$
            else $i \leftarrow f[i]$
if $|x| < i$
   then ...                ▷ Occurrence of $x$ in $t$ at position $j - m$.
   else ...                                       ▷ No occurrences.
```

Knuth-Morris-Pratt/The code (cont)

Then

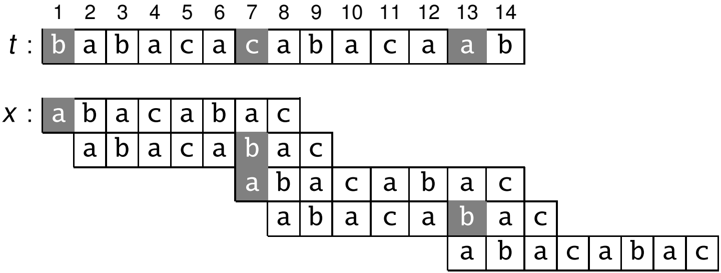
$$\text{MP}(x,t) = \textsc{Search}(x,t,\text{MP-}\textsc{Fail})$$
$$\text{KMP}(x,t) = \textsc{Search}(x,t,\text{KMP-}\textsc{Fail})$$

Knuth-Morris-Pratt/Maximum disjoint borders (cont)

Example. Consider the following table for the search of the word $x = \mathtt{abacabac}$ in the text $t = \mathtt{babacacabacaab}$.

j	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	b	a	b	a	c	a	c	a	b	a	c	a	a	b
i	1	1	2	3	4	5	6	1	2	3	4	5	6	2
	0						1						1	
							0							

Knuth-Morris-Pratt/Example



	1	2	3	4	5	6	7	8	9	10	11	12	13	14						
t :	b	a	b	a	c	a	c	a	b	a	c	a	a	b						
x :	a	b	a	c	a	b	a	c												
		a	b	a	c	a	b	a	c											
							a	b	a	c	a	b	a	c						
								a	b	a	c	a	b	a	c					
													a	b	a	c	a	b	a	c

Comparison of Morris-Pratt and Knuth-Morris-Pratts

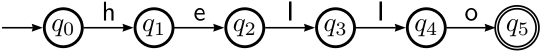
For example, here is the table comparing the values of the two failure functions for the same pattern:

x	a	b	a	c	a	b	a	c
i	1	2	3	4	5	6	7	8
$s_x(i)$	0	1	1	2	1	2	3	4
$r_x(i)$	0	1	0	2	0	1	0	2

Deterministic finite automata

An **automaton** is a very useful and pervasive concept in software and telecommunication engineering.

Let us present first the simplest of the automata, called the **deterministic finite automaton**, or **DFA**. The most intuitive presentation of automata is by means of a graphic:

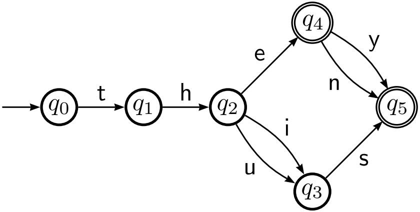


One component is a circle or a double-circle, called **node**, with a name inside, called **state**. The other component is an arrow which connect two circles with a letter: these are **edges** and the letters are **labels**. The arrow which has no originating circle is a marker to distinguish a state, just as the double-circle denotes another special state: the former is an **initial state** and the latter is a **final state**.

DFA (cont)

It is possible to have several edges between the same pair of states but they must carry different labels. There may be several final states.

For instance, consider



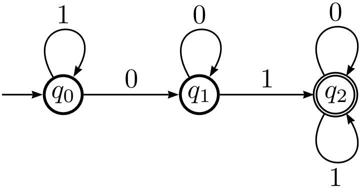
Note that here

1. the **alphabet** of labels is the English alphabet,
2. there are nodes without out-going edges carrying all possible letters: this DFA is not **complete**.

DFA (cont)

The graphic representation of an automaton is called a **transition diagram**.

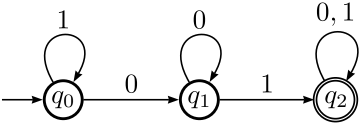
Here is another transition diagram over the alphabet $\{0, 1\}$, called binary alphabet.



Note that there are here some edges that connect a node to itself: these edges are called **loops**.

DFA (cont)

It is possible to simplify the transition diagram by merging edges which have the same pair of nodes and listing the labels, separated by commas:



DFA/Trie

Before giving a formal definition of DFAs, we can explain on transition diagrams how they are used on an example.

Let a short list of English words be composed of then, they, thus and this. Imagine we are given a word and asked to check if this word belongs to our list.

We can of course, compare the given word to each word in the list, one after the other, until a match is found or the end of the list is reached. But this is not efficient.

DFA/Trie (cont)

A better approach is to sort the words in the list in alphabetic order. This way, if we check the words in order and the current word is “greater” than the given word, we know it is not in the list.

This is the way we search a word in a dictionary (well, not exactly, but the analogy is close enough).

Our dictionary is: this, then, they and thus (order matters). Let us search the word they.

DFA/Trie (cont)

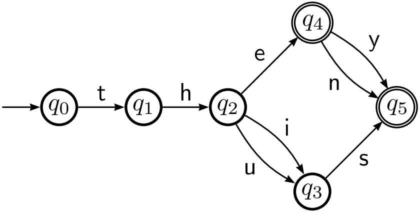
First, they is compared to this. The failure happens at the third letter (i.e., after three letter comparisons) and we must try the next word, then.

Second, they is compared to then. But then, we compare in turn the first letters t and h, i.e., we start again from the beginning of the searched word.

There is a way to avoid this inefficiency by considering a model of the dictionary that is not one-dimensional, like a list, but two-dimensional, like a DFA — called in this context a **trie**.

DFA/Trie (cont)

The trie corresponding to our example dictionary is exactly the transition diagram given page 80. Let us assume here that the word is followed by a special marker \$.



At the beginning of the search, the current state is the initial state, q_0, and the current letter in the searched word is t.

Then if there is an out-going edge from the current state that matches the current letter, we change the current state to the state pointed to by this edge and the current letter becomes the following one in the word or is \$.

DFA/Trie (cont)

If the current letter is $, i.e., all letters of the word have been successfully matched, and the current state is a final state (double-circled node), then the words belongs to the dictionary (in other words, the trie accepts the word or recognises it).

Otherwise, the word is not in the dictionary.

In the worst case, a word of length *n* is in the dictionary and it is found in exactly *n* letter comparisons.

DFA/Formal definition

Formally, a DFA $\mathcal{D}$ is a 5-tuple $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$ where

1. a finite set of *states*, often noted Q;
2. an *initial state* $q_0 \in Q$;
3. a set of *final (or accepting) states* $F \subseteq Q$;
4. a finite set of *input symbols*, called *alphabet*, often noted Σ;
5. a *transition function* δ that takes a state and an input symbol and returns a state: if q is a state with an edge labeled a, the edge leads to state $\delta(q, a)$.

DFA/Recognised words

Independently of the interpretation of the states, we can define how a given word is accepted (or recognised) or rejected by a given DFA.

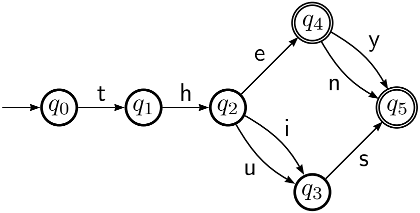
The word $a_1 a_2 \cdots a_n$, where $a_i \in \Sigma$, is recognised by the DFA $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$ if

- for all $0 \leqslant i \leqslant n - 1$
- there is a sequence of states $q_i \in Q$ such that
- $\delta(q_i, a_{i+1}) = q_{i+1}$
- and $q_n \in F$.

The language recognised by $\mathcal{D}$, noted $L(\mathcal{D})$ is the set of words recognised by $\mathcal{D}$.

DFA/Recognised words/Example

For example, consider again the DFA page 80:



The word then is recognised because there is a sequence of states $(q_0, q_1, q_2, q_4, q_5)$ connected by edges which satisfies

$$\begin{aligned}\delta(q_0, \mathrm{t}) &= q_1\\ \delta(q_1, \mathrm{h}) &= q_2\\ \delta(q_2, \mathrm{e}) &= q_4\\ \delta(q_4, \mathrm{n}) &= q_5\end{aligned}$$

and $q_5 \in F$, i.e., q_5 is a final state.

DFA/Recognised language

It is easy to define formally $L(\mathcal{D})$.

Let $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$.

First, let us extend δ to words and let us call this extension $\hat{\delta}$:

- for all state $q \in Q$, let $\hat{\delta}(q, \varepsilon) = q$, where ε is the empty string;
- for all state $q \in Q$, all word $w \in \Sigma^*$, all input $a \in \Sigma$, let $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$.

Then the word w is recognised by $\mathcal{D}$ if $\hat{\delta}(q_0, w) \in F$. The language $L(\mathcal{D})$ recognised by $\mathcal{D}$ is defined as

$$L(\mathcal{D}) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\}$$

DFA/Recognised language (cont)

For example, in our last example:

$$\begin{aligned}
\hat{\delta}(q_0, \epsilon) &= q_0 \\
\hat{\delta}(q_0, \mathrm{t}) &= \delta(\hat{\delta}(q_0, \epsilon), \mathrm{t}) = \delta(q_0, \mathrm{t}) = q_1 \\
\hat{\delta}(q_0, \mathrm{th}) &= \delta(\hat{\delta}(q_0, \mathrm{t}), \mathrm{h}) = \delta(q_1, \mathrm{h}) = q_2 \\
\hat{\delta}(q_0, \mathrm{the}) &= \delta(\hat{\delta}(q_0, \mathrm{th}), \mathrm{e}) = \delta(q_2, \mathrm{e}) = q_4 \\
\hat{\delta}(q_0, \mathrm{then}) &= \delta(\hat{\delta}(q_0, \mathrm{the}), \mathrm{n}) = \delta(q_4, \mathrm{n}) = q_5 \in F
\end{aligned}$$

DFA/Recognised language (cont)

What is the language recognised by the previous DFA? The definition states that this language is composed of all the words that are recognised.

Each recognised word corresponds to a **path**, i.e., a sequence of states connected by edges, from the initial state to a final state.

In our example, the language recognised is $\{\text{then}, \text{they}, \text{this}, \text{thus}\}$.

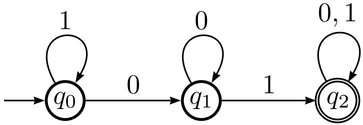
In other words:

$$\hat{\delta}(q_0, \text{then}) \in F$$
$$\hat{\delta}(q_0, \text{they}) \in F$$
$$\hat{\delta}(q_0, \text{this}) \in F$$
$$\hat{\delta}(q_0, \text{thus}) \in F$$

It is a finite language.

DFA/Recognised language (cont)

What is the language recognised by the DFA page 82?



In English, this language can be defined as the set of strings of 0 and 1 containing 01.

Note that this an *infinite* language.

DFA/Transition diagrams

We can also redefine transition diagrams in terms of the concept of DFA.

A transition diagram for a DFA $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$ is a graph defined as follows:

1. for each state q in Q there is a **node**, i.e., a single circle with q inside;
2. for each state $q \in Q$ and each input symbol $a \in \Sigma$, if $\delta(q, a)$ exists, then there is an **edge**, i.e., an arrow, from the node denoting q to the node denoting $\delta(q, a)$ labeled by a; multiple edges can be merged into one and the labels are then separated by commas;
3. there is an edge coming to the node denoting q_0 without origin;
4. nodes corresponding to final states (i.e., in F) are double-circled.

DFA/Transition table

There is a compact textual way to represent the transition function of a DFA: a **transition table**.

The rows of the table correspond to the states and the columns correspond to the inputs (symbols). In other words, the entry for the row corresponding to state q and the column corresponding to input a is the state $\delta(q, a)$:

δ	$\ldots$	a	$\ldots$
$\vdots$			
q		$\delta(q, a)$	
$\vdots$			

DFA/Transition table/Example

The transition table corresponding to the function δ of our last example is

$\mathcal{D}$	0	1
$\rightarrow q_0$	q_1	q_0
q_1	q_1	q_2
$\# q_2$	q_2	q_2

Actually, we added some extra information in the table: the initial state is marked with → and the final states are marked with $\#$.

Therefore, it is not only δ which is defined by means of the transition table here, but the whole DFA $\mathcal{D}$.

DFA/Example

We want to define formally a DFA which recognises the language L whose words contain an even number of 0's and an even number of 1's (the alphabet is binary).

We should understand that the role of the states here is to **not** to count the exact number of 0's and 1's that have been recognised before but this number **modulo 2**.

Therefore, there are four states because there are four cases:

1. there has been an even number of 0's and 1's (state q_0);
2. there has been an even number of 0's and an odd number of 1's (state q_1);
3. there has been an odd number of 0's and an even number of 1's (state q_2);
4. there has been an odd number of 0's and 1's (state q_3).

DFA/Example (cont)

What about the initial and final states?

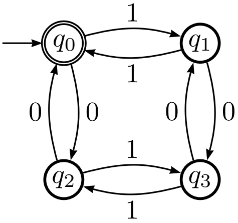
- State q_0 is the initial state because before considering any input, the number of 0's and 1's is zero and zero is even.
- State q_0 is the lone final state because its definition matches exactly the characteristic of L and no other state matches.

We know now almost how to specify the DFA for language L. It is

$$\mathcal{D} = (\{q_0, q_1, q_2, q_3\}, \{0, 1\}, \delta, q_0, \{q_0\})$$

where the transition function δ is described by the following transition diagram.

DFA/Example (cont)



Notice how each input 0 causes the state to cross the horizontal line.

Thus, after seeing an even number of 0's we are always above the horizontal line, in state q_0 or q_1, and after seeing an odd number of 0's we are always below this line, in state q_2 or q_3.

There is a vertically symmetric situation for transitions on 1.

DFA/Example (cont)

We can also represent this DFA by a transition table:

$\mathcal{D}$	0	1
# $\rightarrow q_0$	q_2	q_1
q_1	q_3	q_0
q_2	q_0	q_3
q_3	q_1	q_2

We can use this table to illustrate the construction of $\hat{\delta}$ from *delta*. Suppose the input is 110101. Since this string has even numbers of 0's and 1's, it belongs to L, i.e., we expect $\hat{\delta}(q_0, 110101) = q_0$, since q_0 is the sole final state.

DFA/Example (cont)

We can check this by computing step by step $\hat{\delta}(q_0, \mathtt{110101})$, from the shortest prefix to the longest (which is the word $\mathtt{110101}$ itself):

$$\begin{aligned}
\hat{\delta}(q_0, \varepsilon) &= q_0 \\
\hat{\delta}(q_0, \mathtt{1}) &= \delta(\hat{\delta}(q_0, \varepsilon), \mathtt{1}) = \delta(q_0, \mathtt{1}) = q_1 \\
\hat{\delta}(q_0, \mathtt{11}) &= \delta(\hat{\delta}(q_0, \mathtt{1}), \mathtt{1}) = \delta(q_1, \mathtt{1}) = q_0 \\
\hat{\delta}(q_0, \mathtt{110}) &= \delta(\hat{\delta}(q_0, \mathtt{11}), \mathtt{0}) = \delta(q_0, \mathtt{0}) = q_2 \\
\hat{\delta}(q_0, \mathtt{1101}) &= \delta(\hat{\delta}(q_0, \mathtt{110}), \mathtt{1}) = \delta(q_2, \mathtt{1}) = q_3 \\
\hat{\delta}(q_0, \mathtt{11010}) &= \delta(\hat{\delta}(q_0, \mathtt{1101}), \mathtt{0}) = \delta(q_3, \mathtt{0}) = q_1 \\
\hat{\delta}(q_0, \mathtt{110101}) &= \delta(\hat{\delta}(q_0, \mathtt{11010}), \mathtt{1}) = \delta(q_1, \mathtt{1}) = q_0 \in F
\end{aligned}$$

DFA/Dictionary

Let us use DFAs for implementing a dictionary, that is, a trie as presented at page 80.

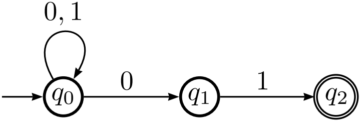
First, here is how to add a word $w \in \Sigma^+$ to a trie $\mathcal{D} = (Q, \Sigma, \delta, q_0, F)$. The transition function δ is considered here as an array which is modified in place.

```
ADD$(w, (Q, \Sigma, \delta, q_0, F))$
$p \leftarrow q_0$; $i \leftarrow 1$
while $\delta[p, w[i]]$ is defined
      do $p \leftarrow \delta[p, w[i]]$; $i \leftarrow i + 1$
for $j \leftarrow i$ to $|w|$
      do $q \leftarrow$ NEW-STATE$(Q)$; $Q \leftarrow Q \cup \{q\}$; $\delta[p, w[j]] \leftarrow q$; $p \leftarrow q$
$F \leftarrow F \cup \{p\}$
```

Non-deterministic finite automata

A **non-deterministic finite automaton** (**NFA**) has the same definition as a DFA except that δ returns a set of states instead of one state.

Consider



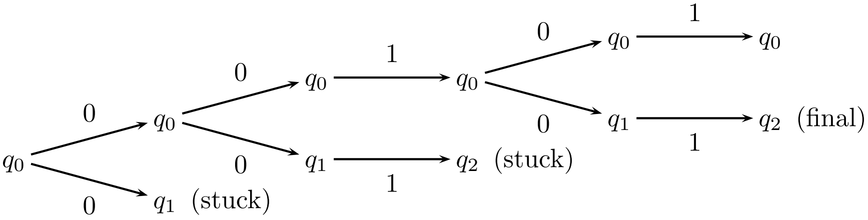
There are two out-going edges from state q_0 which are labeled 0, hence two states can be reached when 0 is input: q_0 (loop) and q_1.

This NFA recognises the language of words on the binary alphabet whose suffix is 01.

NFA (cont)

Before describing formally what is a recognisable language by a NFA, let us consider as an example the previous NFA and the input 00101.

Let us represent each transition for this input by an edge in a tree where nodes are states of the NFA.



NFA/Formal definitions

A NFA is represented essentially like a DFA: $\mathcal{N} = (Q_N, \Sigma, \delta_N, q_0, F_N)$ where the names have the same interpretation as for DFA, except δ_N which returns a subset of Q — not an element of Q.

For example, the NFA whose transition diagram is page 104 can be specified formally as

$$\mathcal{N} = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta_N, q_0, \{q_2\})$$

where the transition function δ_N is given by the transition table:

$\mathcal{N}$	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\varnothing$	$\{q_2\}$
$\# q_2$	$\varnothing$	$\varnothing$

NFA/Formal definitions (cont)

Note that, in the transition table of a NFA, all the cells are filled: there is no transition between two states if and only if the corresponding cell contains $\emptyset$.

In case of a DFA, the cell would remain empty.

It is common also to set that in case of the empty word input, ε, both for the DFA and NFA, the state remains the same:

- for DFA: $\forall q \in Q.\delta_D(q, \varepsilon) = q$
- for NFA: $\forall q \in Q.\delta_N(q, \varepsilon) = \{q\}$

NFA/Formal definitions (cont)

As we did for the DFAs, we can *extend the transition function* δ_N to accept words and not just letters (labels). The extended function is noted $\hat{\delta}_N$ and defined as follows:

- for all state $q \in Q$, let $\hat{\delta}_N(q, \varepsilon) = \{q\}$
- for all state $q \ \in Q$, all words $w \in \Sigma^*$, all input $a \in \Sigma$, let

$$\hat{\delta}_N(q, wa) = \bigcup_{q' \in \hat{\delta}_N(q,w)} \delta_N(q', a)$$

The language $L(\mathcal{N})$ recognised by a NFA $\mathcal{N}$ is defined as

$$L(\mathcal{N}) = \{w \in \Sigma^* \mid \hat{\delta}_N(q_0, w) \cap F \neq \varnothing\}$$

which means that the processing of the input stops successfully as soon as **at least one current state belongs to** F.

NFA/Example

Let us use $\hat{\delta}_N$ to describe the processing of the input $\mathtt{00101}$ by the NFA page 104:

1. $\hat{\delta}_N(q_0, \varepsilon) = q_0$
2. $\hat{\delta}_N(q_0, \mathtt{0}) = \delta_N(q_0, \mathtt{0}) = \{q_0, q_1\}$
3. $\hat{\delta}_N(q_0, \mathtt{00}) = \delta_N(q_0, \mathtt{0}) \cup \delta_N(q_1, \mathtt{0}) = \{q_0, q_1\} \cup \varnothing = \{q_0, q_1\}$
4. $\hat{\delta}_N(q_0, \mathtt{001}) = \delta_N(q_0, \mathtt{1}) \cup \delta_N(q_1, \mathtt{1}) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\}$
5. $\hat{\delta}_N(q_0, \mathtt{0010}) = \delta_N(q_0, \mathtt{0}) \cup \delta_N(q_2, \mathtt{0}) = \{q_0, q_1\} \cup \varnothing = \{q_0, q_1\}$
6. $\hat{\delta}_N(q_0, \mathtt{00101}) = \delta_N(q_0, \mathtt{1}) \cup \delta_N(q_1, \mathtt{1}) = \{q_0\} \cup \{q_2\} = \{q_0, q_2\} \ni q_2$

Because q_2 is a final state, actually $F = \{q_2\}$, we get $\hat{\delta}_N(q_0, \mathtt{00101}) \cap F \neq \varnothing$ thus the string $\mathtt{00101}$ is recognised by the NFA.

Equivalence of DFAs and NFAs

NFA are easier to build than DFA because one does not have to worry, for any state, of having out-going edges carrying a unique label.

The surprising thing is that NFA and DFA actually have the same expressiveness, i.e. all that can be defined by means of a NFA can also be defined with a DFA (the converse is trivial since a DFA is already a NFA).

More precisely, there is a procedure, called **the subset construction**, which converts any NFA to a DFA.

Subset construction

Consider that, in a NFA, from a state q with several out-going edges with the same label a, the transition function δ_N leads, in general, to *several* states.

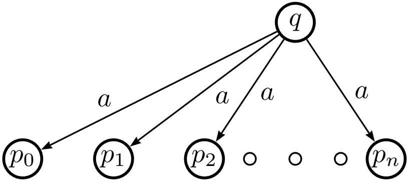
The idea of the subset construction is to create a new automaton where these edges are merged.

So we create a state p which corresponds to the set of states $\delta_N(q, a)$ in the NFA. Accordingly, we create a state r which corresponds to the set $\{q\}$ in the NFA. We create an edge labeled a between r and p. The important point is that *this edge is unique*.

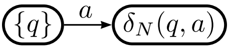
This is the first step to create a DFA from a NFA.

Subset construction (cont)

Graphically, instead of the non-determinism



where $\delta_N(q, a) = \{p_0, p_1, \ldots, p_n\}$, we get the determinism



Subset construction (cont)

Now, let us present the complete algorithm for the subset construction.

Let us start from a NFA $\mathcal{N} = (Q_N, \Sigma, \delta_N, q_0, F_N)$.

The goal is to construct a DFA $\mathcal{D} = (Q_D, \Sigma, \delta_D, \{q_0\}, F_D)$ such that $L(\mathcal{D}) = L(\mathcal{N})$.

Notice that the input alphabet of the two automata are the same and the initial state of $\mathcal{D}$ if the set containing only the initial state of $\mathcal{N}$.

The other components of $\mathcal{D}$ are constructed as follows.

- Q_D is the set of subsets of Q_N; i.e. Q_D is the **power set** of Q_N. Thus, if Q_D has n states, Q_D has 2^n states. Fortunately, often not all these states are **accessible** from the initial state of Q_D, so these inaccessible states can be discarded.

Subset construction (cont)

Why is 2^n the number of subsets of a finite set of cardinal n?

Let us order the n elements. Let us represent each subset by an n-bit string where bit i corresponds to the i-th element: it is 1 if the i-th element is present in the subset and 0 if not.

This way, we counted all the subsets and not more (a bit cannot always be 0 since all elements are used to form subsets and cannot always be 1 if there is more than one element).

There are 2 possibilities, 0 or 1, for the first bit; 2 possibilities for the second bit etc. Since the choices are independent, we multiply all of them: $\underbrace{2 \times 2 \times \cdots \times 2}_{n \text{ times}} = 2^n$.

Hence the number of subsets of an n-element set is also 2^n.

Subset construction (cont)

Resuming the definition of DFA $\mathcal{D}$, the other components are defined as follows.

- F_D is the set of subsets S of Q_N such that $S \cap F_N \neq \emptyset$. That is, F_D is all sets of N's states that include at least one final state of $\mathcal{N}$.
- for each set $S \subseteq Q_N$ and for each input $a \in \Sigma$,

$$\delta_D(S, a) = \bigcup_{q \in S} \delta_N(q, a)$$

In other words, to compute $\delta_D(S, a)$ we look at all the states q in S, see what states of $\mathcal{N}$ are reached from q on input a and take the union of all those states to make the next state of $\mathcal{D}$.

Subset construction/Example/Transition table

Let us consider the NFA given by its transition table page 106:

NFA $\mathcal{N}$	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\varnothing$	$\{q_2\}$
$\#q_2$	$\varnothing$	$\varnothing$

and let us create an equivalent DFA.

First, we form all the subsets of the sets of the NFA and put them in the first column:

DFA $\mathcal{D}$	0	1
$\varnothing$		
$\{q_0\}$		
$\{q_1\}$		
$\{q_2\}$		
$\{q_0, q_1\}$		
$\{q_0, q_2\}$		
$\{q_1, q_2\}$		
$\{q_0, q_1, q_2\}$		

Subset construction/Example/Transition table (cont)

Then we annotate in this first column the states with → if and only if they contain the initial state of the NFA, here q_0, and we add a $\#$ if and only if the states contain at least a final state of the NFA, here q_2.

DFA $\mathcal{D}$	0	1
$\varnothing$		
$\rightarrow\{q_0\}$		
$\{q_1\}$		
$\#\{q_2\}$		
$\{q_0, q_1\}$		
$\#\{q_0, q_2\}$		
$\#\{q_1, q_2\}$		
$\#\{q_0, q_1, q_2\}$		

Subset construction/Example/Transition table (cont)

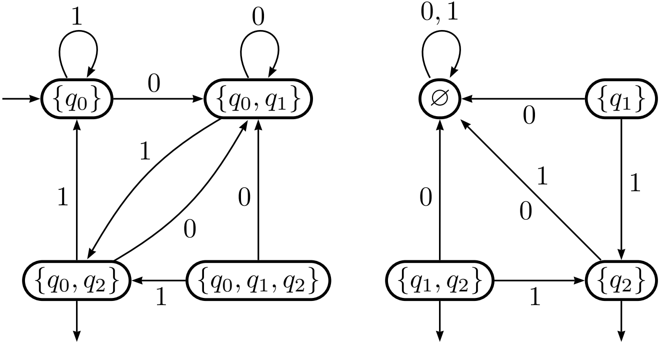
DFA $\mathcal{D}$	0	1
$\varnothing$	$\varnothing$	$\varnothing$
$\rightarrow\{q_0\}$	$\delta_N(q_0, 0)$	$\delta_N(q_0, 1)$
$\{q_1\}$	$\delta_N(q_1, 0)$	$\delta_N(q_1, 1)$
$\#\{q_2\}$	$\delta_N(q_2, 0)$	$\delta_N(q_2, 1)$
$\{q_0, q_1\}$	$\delta_N(q_0, 0) \cup \delta_N(q_1, 0)$	$\delta_N(q_0, 1) \cup \delta_N(q_1, 1)$
$\#\{q_0, q_2\}$	$\delta_N(q_0, 0) \cup \delta_N(q_2, 0)$	$\delta_N(q_0, 1) \cup \delta_N(q_2, 1)$
$\#\{q_1, q_2\}$	$\delta_N(q_1, 0) \cup \delta_N(q_2, 0)$	$\delta_N(q_1, 1) \cup \delta_N(q_2, 1)$
$\#\{q_0, q_1, q_2\}$	$\delta_N(q_0, 0) \cup \delta_N(q_1, 0) \cup \delta_N(q_2, 0)$	$\delta_N(q_0, 1) \cup \delta_N(q_1, 1) \cup \delta_N(q_2, 1)$

Subset construction/Example/Transition table (cont)

DFA $\mathcal{D}$	0	1
$\emptyset$	$\emptyset$	$\emptyset$
$\rightarrow\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_1\}$	$\emptyset$	$\{q_2\}$
$\#\{q_2\}$	$\emptyset$	$\emptyset$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\#\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\#\{q_1, q_2\}$	$\emptyset$	$\{q_2\}$
$\#\{q_0, q_1, q_2\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

Subset construction/Example/Transition diagram

The transition diagram of the DFA $\mathcal{D}$ is then

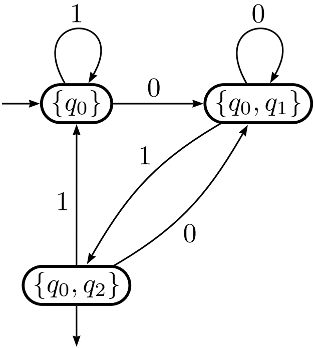


where states with out-going edges which have no end are final states.

Subset construction/Example/Transition diagram (cont.)

If we look carefully at the transition diagram, we see that the DFA is actually made of two parts which are disconnected, i.e. not joined by an edge.

In particular, since we have only one initial state, this means that one part is not accessible, i.e. some states are never used to recognise or reject an input word, and we can remove this part.



Subset construction/Example/Transition diagram (cont.)

It is important to understand that the states of the DFA are subsets of the NFA states.

This is due to the construction and, when finished, it is possible to hide this by **renaming the states**. For example, we can rename the states of the previous DFA in the following manner: $\{q_0\}$ into A, $\{q_0, q_1\}$ in B and $\{q_0, q_2\}$ in C.

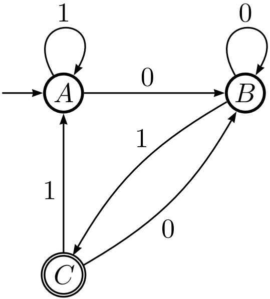
So the transition table changes:

DFA $\mathcal{D}$	0	1
$\rightarrow\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\#\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

DFA $\mathcal{D}$	0	1
$\rightarrow A$	B	A
B	B	C
$\#C$	B	A

Subset construction/Example/Transition diagram (cont.)

So, finally, the DFA is simply



Subset construction/Optimisation

Even if in the worst case the resulting DFA has an exponential number of states of the corresponding NFA, it is in practice often possible to avoid the construction of inaccessible states.

- The singleton containing the initial state (in our example, $\{q_0\}$) is accessible.
- Assume we have a set S of accessible states; then for each input symbol a, we compute $\delta_D(S, a)$: this new set is also accessible.
- Repeat the last step, starting with $\{q_0\}$, until no new (accessible) sets are found.

Subset construction/Optimisation/Example

Let us consider the NFA given by its transition table page 106:

NFA $\mathcal{N}$	0	1
$\rightarrow q_0$	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\varnothing$	$\{q_2\}$
$\# q_2$	$\varnothing$	$\varnothing$

Initially, the sole subset of accessible states is $\{q_0\}$:

DFA $\mathcal{D}$	0	1
$\rightarrow\{q_0\}$	$\delta_N(q_0, 0)$	$\delta_N(q_0, 1)$

that is

DFA $\mathcal{D}$	0	1
$\rightarrow\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$

Subset construction/Optimisation/Example (cont)

Therefore $\{q_0, q_1\}$ and $\{q_0\}$ are accessible sets. But $\{q_0\}$ is not a new set, so we only add to the table entries $\{q_0, q_1\}$ and compute the transitions from it:

DFA $\mathcal{D}$	0	1
$\rightarrow\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$

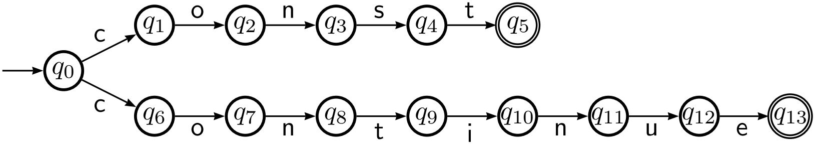
This step uncovered a new set of accessible states, $\{q_0, q_2\}$, which we add to the table and repeat the procedure, and mark it as final state since $q_2 \in \{q_0, q_2\}$:

DFA $\mathcal{D}$	0	1
$\rightarrow\{q_0\}$	$\{q_0, q_1\}$	$\{q_0\}$
$\{q_0, q_1\}$	$\{q_0, q_1\}$	$\{q_0, q_2\}$
$\#\{q_0, q_2\}$	$\{q_0, q_1\}$	$\{q_0\}$

We are done since there is no more new accessible sets.

Subset construction/Tries

Compilers try to recognise a prefix of the input character stream (i.e the first meaningful unit, called **lexeme**, of the given program). Consider the C keywords **const** and **continue**:

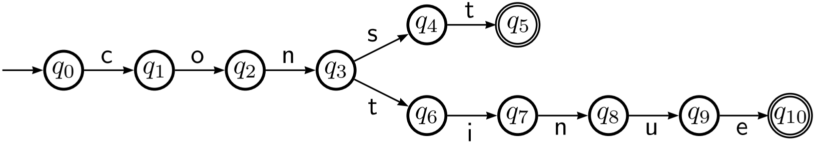


This example shows that a NFA is much more comfortable than a DFA for specifying lexemes: we design *separately* the automata for each kind of lexeme, called **token**, and then merge their initial states into one, leading to one (possibly big) NFA.

It is possible to apply the subset construction to this NFA.

Subset construction/Tries (cont)

After forming the corresponding NFA as in the previous example, it is actually easy to construct an equivalent DFA by **sharing their prefixes**, hence obtaining a tree-like automaton called **trie** (pronounced as the word ‘try’):



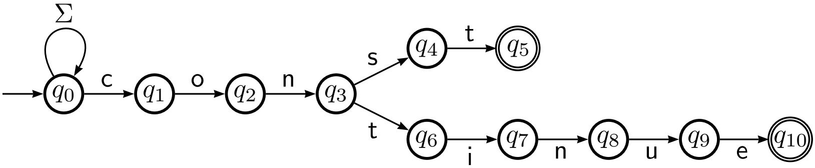
The origin of this word comes from the word re**trie**val.

Note that this construction only works for a list of constant words, like keywords.

Subset construction/Text searching

This technique can easily be generalized for searching constant strings (like keywords) in a text, i.e. not only as a prefix of a text, but *at any position*.

It suffices to add a loop on the initial state for each possible input symbol. If we note Σ the language alphabet, we get the NFA



Subset construction/Text searching (cont)

It is possible to apply the subset construction to this NFA or to use it directly for searching *all occurrences* of the two keywords at *any position* in a text.

In case of direct use, the difference between this NFA and the trie page 128 is that there is no need here to “restart” by hand the recognition process once a keyword has been recognised: we just do not stop on final states but continue reading the input after discarding the recognised prefix (at the last final state).

Subset construction/Text searching (cont)

This works because of the loop on the initial state.

Try for instance the input `constantcontinue`. We have $\hat{\delta}(q_0, \mathtt{const}) = \{q_0, q_5\}$. Since q_5 is a final state, we recognised the input `const`.

But we can continue because $\hat{\delta}(q_0, \mathtt{consta}) = \{q_0\}$ etc., until we reach the end of the input.

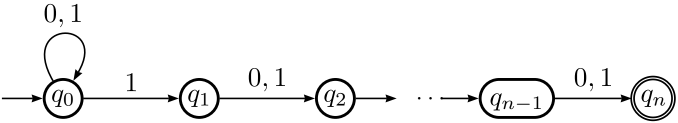
Subset construction/Bad case

The subset construction can lead, in the worst case, to a number of states which is the total number of state subsets of the NFA.

In other words, if the NFA has n states, the equivalent DFA by subset construction can have 2^n states (see page 114 for the count of all the subsets of a finite set).

Subset construction/Bad case (cont)

Consider the following NFA, which recognises all binary strings which have 1 at the *n*-th position from the end:



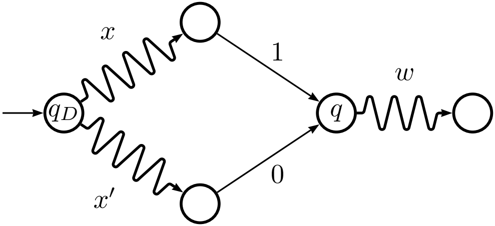
The language recognised by this NFA is $\Sigma^* 1 \Sigma^{n-1}$, where $\Sigma = \{0, 1\}$, that is: all words of length greater or equal to *n* are accepted as long as the *n*-th bit from the **end** is 1.

Therefore, in any equivalent DFA, all the prefixes of length *n* should not lead to a stuck state, because the automaton must wait until the **end** of the word to accept or reject it.

Subset construction/Bad case (cont)

If the states reached by these prefixes are all different, then there are at least 2^n states in the DFA.

Equivalently (by contraposition), if there are less than 2^n states, then some states can be reached by several strings of length *n*:



where words $x1w$ and $x'0w$ have length *n*.

Subset construction/Bad case (cont)

Let us call the DFA $\mathcal{D} = (Q_D, \Sigma, \delta_D, q_D, F_D)$, where $q_D = \{q_0\}$.

The extended transition function is noted $\hat{\delta}_D$ as usual. The situation of the previous picture can be formally expressed as

$$\hat{\delta}_D(q_D, x1) = \hat{\delta}_D(q_D, x'0) = q \tag{1}$$

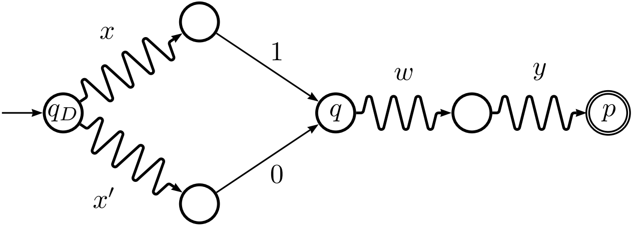
$$|x1w| = |x'0w| = n \tag{2}$$

where $|u|$ is the length of u.

Subset construction/Bad case (cont)

Let y be a any string of 0 and 1 such that $|wy| = n - 1$.

Then $\hat{\delta}_D(q_D, x1wy) \in F_D$ since there is a 1 at the n-th position from the end:



Also, $\hat{\delta}_D(q_D, x'0wy) \notin F_D$ because there is a 0 at the n-th position from the end.

Subset construction/Bad case (cont)

On the other hand, equation (1) implies

$$\hat{\delta}_D(q_D, x1wy) = \hat{\delta}_D(q_D, x'0wy) = p$$

So there is contradiction because a state (here, p) must be either final or not final, it cannot be both...

As a consequence, we must reject our initial assumption: there are at least 2^n states in the equivalent DFA.

This is a very bad case, even if it is not the worst case (2^{n+1} states).

NFA with ϵ-transitions

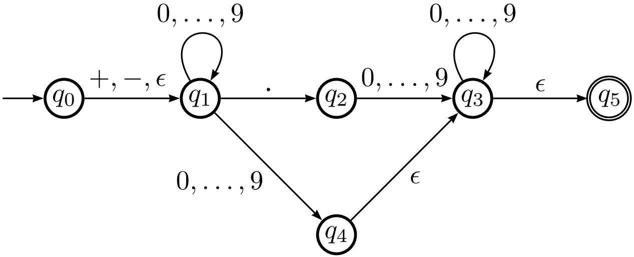
We shall now introduce another extension to NFA, called ϵ-NFA, which is a NFA whose labels can be the empty string, noted ϵ.

The interpretation of this new kind of transition, called ϵ-transition, is that the current state changes by following this transition *without reading any input*. This is sometimes referred as a **spontaneous transition**.

The rationale, i.e., the intuition behind that, is that $\epsilon a = a\epsilon = a$, so recognising ϵa or $a\epsilon$ is the same as recognising a. In other words, we do not need to read something more than a as input.

ϵ-NFA/Example

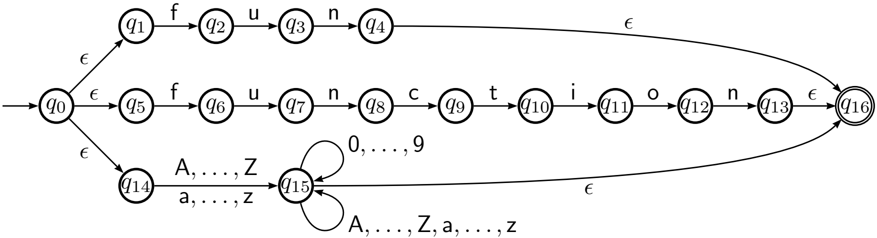
For example, we can specify signed natural and decimal numbers by means of the ϵ-NFA



This is not the simplest ϵ-NFA we can imagine for these numbers, but note the utility of the ϵ-transition between q_0 and q_1.

ϵ-NFA (cont)

In the case of compilers, ϵ-NFA allow to design separately a NFA for each token, then create an initial (respectively final) state connected to all their initial (respectively, final) states with an ϵ-transition. For instance, for keywords **fun** and **function** and identifiers:



ϵ-NFA (cont)

In order to perform a text search, once we have a single ϵ-NFA, we can

1. either remove all the ϵ-transitions and
   1.1 either create a NFA and then maybe a DFA;
   1.2 or create directly a DFA,
2. or use a formal definition of ϵ-NFA that directly leads to a recognition algorithm, just as we did for DFA and NFA.

Both methods assume that it is always possible to create an equivalent NFA, hence a DFA, from a given ϵ-NFA.

In other words, **DFA, NFA and ϵ-NFA have the same expressive power.**

ϵ-NFA (cont)

The first method constructs explicitly the NFA and maybe the DFA, while the second does not, at the possible cost of more computations at run-time.

Before entering into the details, we need to define formally an ϵ-NFA, as suggested by the second method.

The only difference between an NFA and an ϵ-NFA is that the transition function δ_E takes as second argument an element in $\Sigma \cup \{\epsilon\}$, with $\epsilon \notin \Sigma$, instead of Σ — but the alphabet still remains Σ.

ϵ-NFA/ϵ-closure

We need now a function called **ϵ-close**, which takes an ϵ-NFA $\mathcal{E}$, a state q of $\mathcal{E}$ and returns all the states which are accessible in $\mathcal{E}$ from q with label ϵ.

The idea is to achieve a **depth-first traversal** of $\mathcal{E}$, starting from q and following only ϵ-transitions.

Let us call ϵ-DFS ("ϵ-Depth-First-Search") the function such that $\epsilon\text{-DFS}(q, Q)$ is the set of states reachable from q following ϵ-transitions and which is not included in Q, *Q being interpreted as the set of states already visited in the traversal*. The set Q ensures the termination of the algorithm even in presence of cycles in the automaton. Therefore, let

$$\epsilon\text{-close}(q) = \epsilon\text{-DFS}(q, \varnothing) \quad \text{if } q \in Q_E$$

where the ϵ-NFA is $\mathcal{E} = (Q_E, \Sigma, \delta_E, q_0, F_E)$.

ϵ-NFA/ϵ-closure (cont)

Now we define ϵ-DFS as follows:

$$\epsilon\text{-DFS}(q, Q) = \emptyset \qquad \text{if } q \in Q \qquad (3)$$

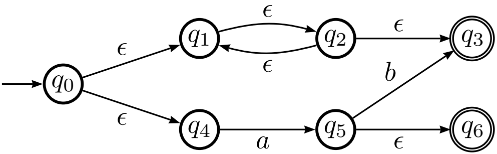
$$\epsilon\text{-DFS}(q, Q) = \{q\} \quad \cup \bigcup_{p \in \delta_E(q,\epsilon)} \epsilon\text{-DFS}(p, Q \cup \{q\}) \qquad \text{if } q \notin Q \qquad (4)$$

The ϵ-NFA page 139 leads to the following ϵ-closures:

$$\begin{aligned}
\epsilon\text{-close}(q_1) &= \{q_1\} & \epsilon\text{-close}(q_2) &= \{q_2\} \\
\epsilon\text{-close}(q_0) &= \{q_0, q_1\} & \epsilon\text{-close}(q_3) &= \{q_3, q_5\} \\
\epsilon\text{-close}(q_5) &= \{q_5\} & \epsilon\text{-close}(q_4) &= \{q_4, q_3, q_5\}
\end{aligned}$$

ϵ-NFA/ϵ-closure (cont)

Consider, as a more difficult example, the following ϵ-NFA $\mathcal{E}$:



$$
\begin{aligned}
\epsilon\text{-close}(q_0) &= \epsilon\text{-DFS}(q_0, \varnothing) && \text{since } q_0 \in Q_E \\
&= \{q_0\} \cup \epsilon\text{-DFS}(q_1, \{q_0\}) \cup \epsilon\text{-DFS}(q_4, \{q_0\}) && \text{by eq. 4} \\
&= \{q_0\} \cup \left(\{q_1\} \cup \bigcup_{p \in \delta_E(q_1, \epsilon)} \epsilon\text{-DFS}(p, \{q_0, q_1\}) \right) && \text{by eq. 4} \\
&\qquad \cup \left(\{q_4\} \cup \bigcup_{p \in \delta_E(q_4, \epsilon)} \epsilon\text{-DFS}(p, \{q_0, q_4\}) \right) && \text{by eq. 4}
\end{aligned}
$$

ϵ-NFA/ϵ-closure (cont)

$$\begin{aligned}
\epsilon\text{-close}(q_0) &= \{q_0\} \cup \left(\{q_1\} \cup \bigcup_{p \in \{q_2\}} \epsilon\text{-DFS}(p, \{q_0, q_1\})\right) \\
&\qquad \cup \left(\{q_4\} \cup \bigcup_{p \in \varnothing} \epsilon\text{-DFS}(p, \{q_0, q_4\})\right) \\
&= \{q_0\} \cup (\{q_1\} \cup \epsilon\text{-DFS}(q_2, \{q_0, q_1\})) \cup (\{q_4\} \cup \varnothing) \\
&= \{q_0, q_1, q_4\} \cup \epsilon\text{-DFS}(q_2, \{q_0, q_1\}) \\
&= \{q_0, q_1, q_4\} \cup \left(\{q_2\} \cup \bigcup_{p \in \delta_E(q_2, \epsilon)} \epsilon\text{-DFS}(p, \{q_0, q_1, q_2\})\right)
\end{aligned}$$

ϵ-NFA/ϵ-closure (cont)

$$
\begin{aligned}
\epsilon\text{-close}(q_0) &= \{q_0, q_1, q_4\} \cup \left(\{q_2\} \cup \bigcup_{p \in \{q_1, q_3\}} \epsilon\text{-DFS}(p, \{q_0, q_1, q_2\})\right) \\
&= \{q_0, q_1, q_2, q_4\} \cup \epsilon\text{-DFS}(q_1, \{q_0, q_1, q_2\}) \\
&\qquad\qquad\qquad\quad \cup\, \epsilon\text{-DFS}(q_3, \{q_0, q_1, q_2\}) \\
&= \{q_0, q_1, q_2, q_4\} \cup \varnothing \qquad\qquad \text{by eq. 3, since } q_1 \in \{q_0, q_1, q_2\} \\
&\quad \cup \left(\{q_3\} \cup \bigcup_{p \in \delta_E(q_3, \epsilon)} \epsilon\text{-DFS}(p, \{q_0, q_1, q_2, q_3\})\right) \quad \text{by eq. 4} \\
&= \{q_0, q_1, q_2, q_3, q_4\} \cup \bigcup_{p \in \varnothing} \epsilon\text{-DFS}(p, \{q_0, q_1, q_2, q_3\}) \\
&= \{q_0, q_1, q_2, q_3, q_4\}
\end{aligned}
$$

ϵ-NFA/ϵ-closure (cont)

It is useful to extend ϵ-close to sets of states, not just states.

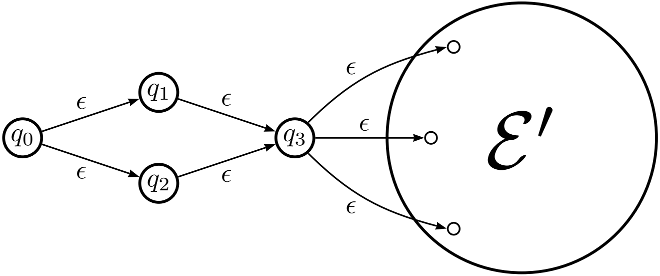
Let us note $\overline{\epsilon\text{-close}}$ this extension, which we can easily define as

$$\overline{\epsilon\text{-close}}(Q) = \bigcup_{q \in Q} \epsilon\text{-close}(q)$$

for any subset $Q \subseteq Q_E$ where the ϵ-NFA is $\mathcal{E} = (Q_E, \Sigma, \delta_E, q_E, F_E)$.

ϵ-NFA/ϵ-closure/Optimisation

Compute the ϵ-closure of q_0 in the following ϵ-NFA $\mathcal{E}$:



where the sub-ϵ-NFA $\mathcal{E}'$ contains only ϵ-transitions and all its Q' states are accessible from q_3.

ϵ-NFA/ϵ-closure/Optimisation (cont)

$$\begin{aligned}
\epsilon\text{-close}(q_0) &= \epsilon\text{-DFS}(q_0, \varnothing) \\
&= \{q_0\} \cup \epsilon\text{-DFS}(q_1, \{q_0\}) \cup \epsilon\text{-DFS}(q_2, \{q_0\}) \\
&= \{q_0\} \cup (\{q_1\} \cup \epsilon\text{-DFS}(q3, \{q_0, q_1\})) \\
&\qquad\quad \cup (\{q_2\} \cup \epsilon\text{-DFS}(q_3, \{q_0, q_2\})) \\
&= \{q_0, q_1, q_2\} \cup \epsilon\text{-DFS}(q3, \{q_0, q_1\}) \cup \epsilon\text{-DFS}(q_3, \{q_0, q_2\}) \\
&= \{q_0, q_1, q_2, q_3, \} \cup (\{q_3\} \cup Q') \cup (\{q_3\} \cup Q') \\
&= \{q_0, q_1, q_2, q_3, \} \cup Q'
\end{aligned}$$

We compute $\{q_3\} \cup Q'$ two times, that is, we traverse two times q_3 and all the states of $\mathcal{E}'$, which can be inefficient if Q' is big.

ϵ-NFA/ϵ-closure/Optimisation (cont)

The way to avoid duplicating traversals is to change the definitions of ϵ-close and $\overline{\epsilon\text{-close}}$.
Dually, we need a new definition of ϵ-DFS and create function $\overline{\epsilon\text{-DFS}}$ which is similar to ϵ-DFS except that it applies to set of states instead of one state:

$$\begin{aligned} \epsilon\text{-close}(q) &= \epsilon\text{-DFS}(q, \varnothing) && \text{if } q \in Q_E \\ \overline{\epsilon\text{-close}}(Q) &= \overline{\epsilon\text{-DFS}}(Q, \varnothing) && \text{if } Q \subseteq Q_E \end{aligned}$$

We interpret Q' in ϵ-DFS(q, Q') and $\overline{\epsilon\text{-DFS}}(Q, Q')$ as the set of states that have already been visited in the depth-first search.

Variables q and Q denote, respectively, a state and a set of states that have to be explored.

ϵ-NFA/ϵ-closure/Optimisation (cont)

In the first definition we computed the *new reachable states*, but in the new one we compute the *currently reached states*. Then let us redefine ϵ-DFS this way:

$$\begin{aligned}
&\epsilon\text{-DFS}(q, Q') = Q' && \text{if } q \in Q' && (1')\\
&\epsilon\text{-DFS}(q, Q') = \overline{\epsilon\text{-DFS}}(\delta_E(q, \epsilon), Q' \cup \{q\}) && \text{if } q \notin Q' && (2')
\end{aligned}$$

Contrast with the first definition

$$\begin{aligned}
&\epsilon\text{-DFS}(q, Q') = \varnothing && \text{if } q \in Q' && (1)\\
&\epsilon\text{-DFS}(q, Q') = \{q\} \quad \cup \bigcup_{p \in \delta_E(q,\epsilon)} \epsilon\text{-DFS}(p, Q' \cup \{q\}) && \text{if } q \notin Q' && (2)
\end{aligned}$$

Hence, in (1) we return $\varnothing$ because there is no new state, i.e., none not already in Q', whereas in (1') we return Q' itself.

ϵ-NFA/ϵ-closure/Optimisation (cont)

The new definition of $\overline{\epsilon\text{-DFS}}$ is not more difficult than the first one:

$$\overline{\epsilon\text{-DFS}}(\varnothing, Q') = Q' \tag{5}$$

$$\overline{\epsilon\text{-DFS}}(\{q\} \cup Q, Q') = \overline{\epsilon\text{-DFS}}(Q, \epsilon\text{-DFS}(q, Q')) \qquad \text{if } q \notin Q \tag{6}$$

Notice that the definitions of ϵ-DFS and $\overline{\epsilon\text{-DFS}}$ are **mutually recursive**, i.e., they depend on each other.

In (2) we traverse states in *parallel* (consider the union operator), starting from each element in $\delta_E(q, \epsilon)$, whereas in (2') and (6), we traverse them *sequentially* so we can use the information collected (currently reached states) in the previous searches.

ϵ-NFA/ϵ-closure/Optimisation (cont)

Coming back to our example page 149, we find

$$
\begin{aligned}
\epsilon\text{-close}(q_0) &= \epsilon\text{-DFS}(q_0, \varnothing) && q_0 \in Q_E\\
&= \overline{\epsilon\text{-DFS}}(\{q_1, q_2\}, \{q_0\}) && \text{by eq. (2')}\\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \epsilon\text{-DFS}(q_1, \{q_0\})) && \text{by eq. (4)}\\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \overline{\epsilon\text{-DFS}}(\{q_3\}, \{q_0, q_1\})) && \text{by eq. (2')}\\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \overline{\epsilon\text{-DFS}}(\varnothing, \epsilon\text{-DFS}(q_3, \{q_0, q_1\}))) && \text{by eq. (4)}\\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \epsilon\text{-DFS}(q_3, \{q_0, q_1\})) && \text{by eq. (3)}\\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \{q_0, q_1, q_3\} \cup Q')\\
&= \overline{\epsilon\text{-DFS}}(\varnothing, \epsilon\text{-DFS}(q_2, \{q_0, q_1, q_3\} \cup Q')) && \text{by eq. (4)}
\end{aligned}
$$

ϵ-NFA/ϵ-closure/Optimisation (cont)

$$
\begin{aligned}
\epsilon\text{-close}(q_0) &= \epsilon\text{-DFS}(q_2, \{q_0, q_1, q_3\} \cup Q') && \text{by eq. (3)}\\
&= \overline{\epsilon\text{-DFS}}(\{q_3\}, \{q_0, q_1, q_2, q_3\} \cup Q') && \text{by eq. (2')}\\
&= \overline{\epsilon\text{-DFS}}(\varnothing, \epsilon\text{-DFS}(q_3, \{q_0, q_1, q_2, q_3\} \cup Q')) && \text{by eq. (4)}\\
&= \epsilon\text{-DFS}(q_3, \{q_0, q_1, q_2, q_3\} \cup Q') && \text{by eq. (3)}\\
&= \{q_0, q_1, q_2, q_3\} \cup Q' && \text{by eq. (1')}
\end{aligned}
$$

The important thing here is that we did not compute (traverse) several times Q'. Note that some equations can be used in a different order and q can be chosen arbitrarily in equation (4), but the result is always the same.

Extended transition functions for ϵ-NFAs

The ϵ-closure allows to explain how a ϵ-NFA recognises or rejects a given input word.

In (2) we traverse states in *parallel* (consider the union operator), starting from each element in $\delta_E(q, \epsilon)$, whereas in (2') and (6), we traverse them *sequentially* so we can use the information collected (currently reached states) in the previous searches.

ϵ-NFA/ϵ-closure/Optimisation (cont)

Coming back to our example page 149, we find

$$
\begin{aligned}
\epsilon\text{-close}(q_0) &= \epsilon\text{-DFS}(q_0, \varnothing) && q_0 \in Q_E\\
&= \overline{\epsilon\text{-DFS}}(\{q_1, q_2\}, \{q_0\}) && \text{by eq. (2')}\\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \epsilon\text{-DFS}(q_1, \{q_0\})) && \text{by eq. (4)}\\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \overline{\epsilon\text{-DFS}}(\{q_3\}, \{q_0, q_1\})) && \text{by eq. (2')}\\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \overline{\epsilon\text{-DFS}}(\varnothing, \epsilon\text{-DFS}(q_3, \{q_0, q_1\}))) && \text{by eq. (4)}\\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \epsilon\text{-DFS}(q_3, \{q_0, q_1\})) && \text{by eq. (3)}\\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \{q_0, q_1, q_3\} \cup Q')\\
&= \overline{\epsilon\text{-DFS}}(\varnothing, \epsilon\text{-DFS}(q_2, \{q_0, q_1, q_3\} \cup Q')) && \text{by eq. (4)}
\end{aligned}
$$

ϵ-NFA/ϵ-closure/Optimisation (cont)

$$\begin{aligned}
\epsilon\text{-close}(q_0) &= \epsilon\text{-DFS}(q_2, \{q_0, q_1, q_3\} \cup Q') && \text{by eq. (3)}\\
&= \overline{\epsilon\text{-DFS}}(\{q_3\}, \{q_0, q_1, q_2, q_3\} \cup Q') && \text{by eq. (2')}\\
&= \overline{\epsilon\text{-DFS}}(\varnothing, \epsilon\text{-DFS}(q_3, \{q_0, q_1, q_2, q_3\} \cup Q')) && \text{by eq. (4)}\\
&= \epsilon\text{-DFS}(q_3, \{q_0, q_1, q_2, q_3\} \cup Q') && \text{by eq. (3)}\\
&= \{q_0, q_1, q_2, q_3\} \cup Q' && \text{by eq. (1')}
\end{aligned}$$

The important thing here is that we did not compute (traverse) several times Q'. Note that some equations can be used in a different order and q can be chosen arbitrarily in equation (4), but the result is always the same.

Extended transition functions for ϵ-NFAs

The ϵ-closure allows to explain how a ϵ-NFA recognises or rejects a given input word.

In (2) we traverse states in *parallel* (consider the union operator), starting from each element in $\delta_E(q, \epsilon)$, whereas in (2') and (6), we traverse them *sequentially* so we can use the information collected (currently reached states) in the previous searches.

ϵ-NFA/ϵ-closure/Optimisation (cont)

Coming back to our example page 149, we find

$$\begin{aligned}
\epsilon\text{-close}(q_0) &= \epsilon\text{-DFS}(q_0, \varnothing) && q_0 \in Q_E\\
&= \overline{\epsilon\text{-DFS}}(\{q_1, q_2\}, \{q_0\}) && \text{by eq. (2')}\\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \epsilon\text{-DFS}(q_1, \{q_0\})) && \text{by eq. (4)}\\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \overline{\epsilon\text{-DFS}}(\{q_3\}, \{q_0, q_1\})) && \text{by eq. (2')}\\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \overline{\epsilon\text{-DFS}}(\varnothing, \epsilon\text{-DFS}(q_3, \{q_0, q_1\}))) && \text{by eq. (4)}\\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \epsilon\text{-DFS}(q_3, \{q_0, q_1\})) && \text{by eq. (3)}\\
&= \overline{\epsilon\text{-DFS}}(\{q_2\}, \{q_0, q_1, q_3\} \cup Q')\\
&= \overline{\epsilon\text{-DFS}}(\varnothing, \epsilon\text{-DFS}(q_2, \{q_0, q_1, q_3\} \cup Q')) && \text{by eq. (4)}
\end{aligned}$$

ϵ-NFA/ϵ-closure/Optimisation (cont)

$$\begin{aligned}
\epsilon\text{-close}(q_0) &= \epsilon\text{-DFS}(q_2, \{q_0, q_1, q_3\} \cup Q') && \text{by eq. (3)}\\
&= \overline{\epsilon\text{-DFS}}(\{q_3\}, \{q_0, q_1, q_2, q_3\} \cup Q') && \text{by eq. (2')}\\
&= \overline{\epsilon\text{-DFS}}(\varnothing, \epsilon\text{-DFS}(q_3, \{q_0, q_1, q_2, q_3\} \cup Q')) && \text{by eq. (4)}\\
&= \epsilon\text{-DFS}(q_3, \{q_0, q_1, q_2, q_3\} \cup Q') && \text{by eq. (3)}\\
&= \{q_0, q_1, q_2, q_3\} \cup Q' && \text{by eq. (1')}
\end{aligned}$$

The important thing here is that we did not compute (traverse) several times Q'. Note that some equations can be used in a different order and q can be chosen arbitrarily in equation (4), but the result is always the same.

Extended transition functions for ϵ-NFAs

The ϵ-closure allows to explain how a ϵ-NFA recognises or rejects a given input word.

Let $\mathcal{E} = (Q_E, \Sigma, \delta_E, q_0, F_E)$.

We want $\hat{\delta}_E(q, w)$ be the set of states reachable from q along a path whose labels, when concatenated, for the string w. The difference here with NFA's is that several ϵ can be present along this path, despite not contributing to w. For all state $q \in Q_E$, let

$$\hat{\delta}_E(q, \epsilon) = \epsilon\text{-close}(q)$$

$$\hat{\delta}_E(q, wa) = \overline{\epsilon\text{-close}}\left(\bigcup_{p \in \hat{\delta}_E(q,w)} \delta_N(p, a)\right) \qquad \text{for all } a \in \Sigma, w \in \Sigma^*$$

This definition is based on the regular identity $wa = ((w\epsilon^*)a)\epsilon^*$.

Extended transition functions for ϵ-NFAs/Example

Let us consider again the ϵ-NFA recognising natural and decimal numbers, at page 139, and compute the states reached on the input `5.6`:

$$
\begin{aligned}
\hat{\delta}_E(q_0,\epsilon) &= \epsilon\text{-close}(q_0) = \{q_0,q_1\} \\
\hat{\delta}_E(q_0,5) &= \overline{\epsilon\text{-close}}\left(\bigcup_{p\in\hat{\delta}_E(q_0,\epsilon)} \delta_N(p,5)\right) \\
&= \overline{\epsilon\text{-close}}(\delta_N(q_0,5)\cup\delta_N(q_1,5)) = \overline{\epsilon\text{-close}}(\varnothing\cup\{q_1,q_4\}) \\
&= \{q_1,q_3,q_4,q_5\} \\
\hat{\delta}_E(q_0,5.) &= \overline{\epsilon\text{-close}}\left(\bigcup_{p\in\hat{\delta}_N(q_0,5)} \delta_N(p,.)\right) \\
&= \overline{\epsilon\text{-close}}(\delta_N(q_1,.)\cup\delta_N(q_3,.)\cup\delta_N(q_4,.)\cup\delta_N(q_5,.))
\end{aligned}
$$

Extended transition functions for ϵ-NFAs/Example (cont)

$$\begin{aligned}
\hat{\delta}_E(q_0, 5.) &= \overline{\epsilon\text{-close}}(\{q_2\} \cup \varnothing \cup \varnothing \cup \varnothing) = \{q_2\} \\
\hat{\delta}_N(q_0, 5.6) &= \overline{\epsilon\text{-close}}\left(\bigcup_{p \in \hat{\delta}_E(q_0, 5.)} \delta_N(p, 6) \right) \\
&= \overline{\epsilon\text{-close}}(\delta_N(q_2, 6)) \\
&= \overline{\epsilon\text{-close}}(\{q_3\}) \\
&= \{q_3, q_5\} \ni q_5
\end{aligned}$$

Since q_5 is a final state, the string `5.6` is recognised as a number.

Subset construction for ϵ-NFAs

Let us present now how to construct a DFA from a ϵ-NFA such that both recognise the same language.

The method is a variation of the subset construction we presented for NFA: we must take into account the states reachable through ϵ-transitions, with help of ϵ-closures.

Subset construction for ϵ-NFAs (cont)

Assume that $\mathcal{E} = (Q, \Sigma, \delta, q_0, F)$ is an ϵ-NFA. Let us define as follows the equivalent DFA $\mathcal{D} = (Q_D, \Sigma, \delta_D, q_D, F_D)$.

1. Q_D is the set of subsets of Q_E. More precisely, all accessible states of $\mathcal{D}$ are ϵ-closed subsets of Q_E, i.e., sets $Q \subseteq Q_E$ such that $Q = \epsilon\text{-close}(Q)$.
2. $q_D = \epsilon\text{-close}(q_0)$, i.e., we get the start state of $\mathcal{D}$ by ϵ-closing the set made of only the start state of $\mathcal{E}$.
3. F_D is those sets of states that contain at least one final state of $\mathcal{E}$, that is to say $F_D = \{Q \mid Q \in Q_D \text{ and } Q \cap F_E \neq \varnothing\}$.
4. For all $a \in \Sigma$ and $Q \in Q_D$, let $\delta_D(Q, a) = \overline{\epsilon\text{-close}}\left(\bigcup_{q \in Q} \delta_E(q, a)\right)$

Subset construction for ϵ-NFAs/Example

Let us consider again the ϵ-NFA page 139. Its transition table is

$\mathcal{E}$	$+$	$-$	$0,\ldots,9$	$.$	ϵ
$\rightarrow q_0$	$\{q_1\}$	$\{q_1\}$	$\varnothing$	$\varnothing$	$\{q_1\}$
q_1	$\varnothing$	$\varnothing$	$\{q_1, q_4\}$	$\{q_2\}$	$\varnothing$
q_2	$\varnothing$	$\varnothing$	$\{q_3\}$	$\varnothing$	$\varnothing$
q_3	$\varnothing$	$\varnothing$	$\{q_3\}$	$\varnothing$	$\{q_5\}$
q_4	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\{q_3\}$
$\#q_5$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$

Subset construction for ϵ-NFAs/Example (cont)

By applying the subset construction to this ϵ-NFA, we get the table

$\mathcal{D}$	$+$	$-$	$0, \ldots, 9$	.
$\rightarrow\{q_0, q_1\}$	$\{q_1\}$	$\{q_1\}$	$\{q_1, q_3, q_4, q_5\}$	$\{q_2\}$
$\{q_1\}$	$\varnothing$	$\varnothing$	$\{q_1, q_3, q_4, q_5\}$	$\{q_2\}$
$\#\{q_1, q_3, q_4, q_5\}$	$\varnothing$	$\varnothing$	$\{q_1, q_3, q_4, q_5\}$	$\{q_2\}$
$\{q_2\}$	$\varnothing$	$\varnothing$	$\{q_3, q_5\}$	$\varnothing$
$\#\{q_3, q_5\}$	$\varnothing$	$\varnothing$	$\{q_3, q_5\}$	$\varnothing$

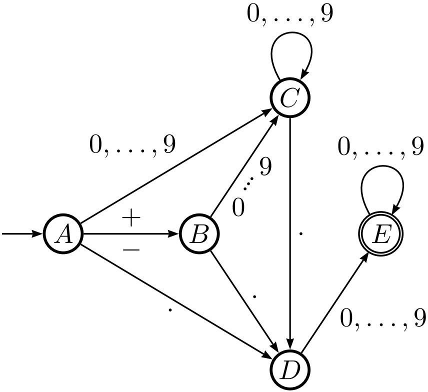
Subset construction for ϵ-NFAs/Example (cont)

Let us rename the states of $\mathcal{D}$ and get rid of the empty sets:

$\mathcal{D}$	$+$	$-$	$0, \ldots, 9$	.
$\rightarrow A$	B	B	C	D
B			C	D
$\#C$			C	D
D			E	
$\#E$			E	

Subset construction for ϵ-NFAs/Example (cont)

The transition diagram of $\mathcal{D}$ is therefore


$0, \ldots, 9$
C
$0, \ldots, 9$
9
0
$0, \ldots, 9$
$+$
$-$
A
B
$.$
E
$.$
$.$
$0, \ldots, 9$
D