Raport z laboratorium 6

Filip Nikolow

27 maja 2021

1 Cel laboratorium

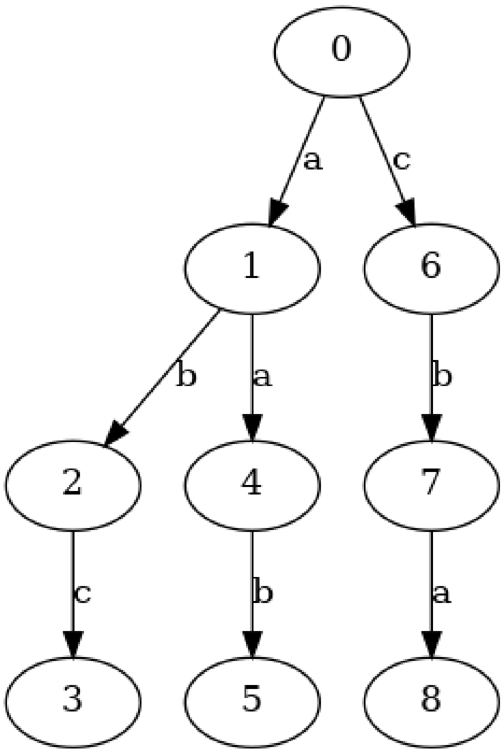
Celem laboratorium była implementacja i testy algorytmu wyszukiwania wzorca 2d.

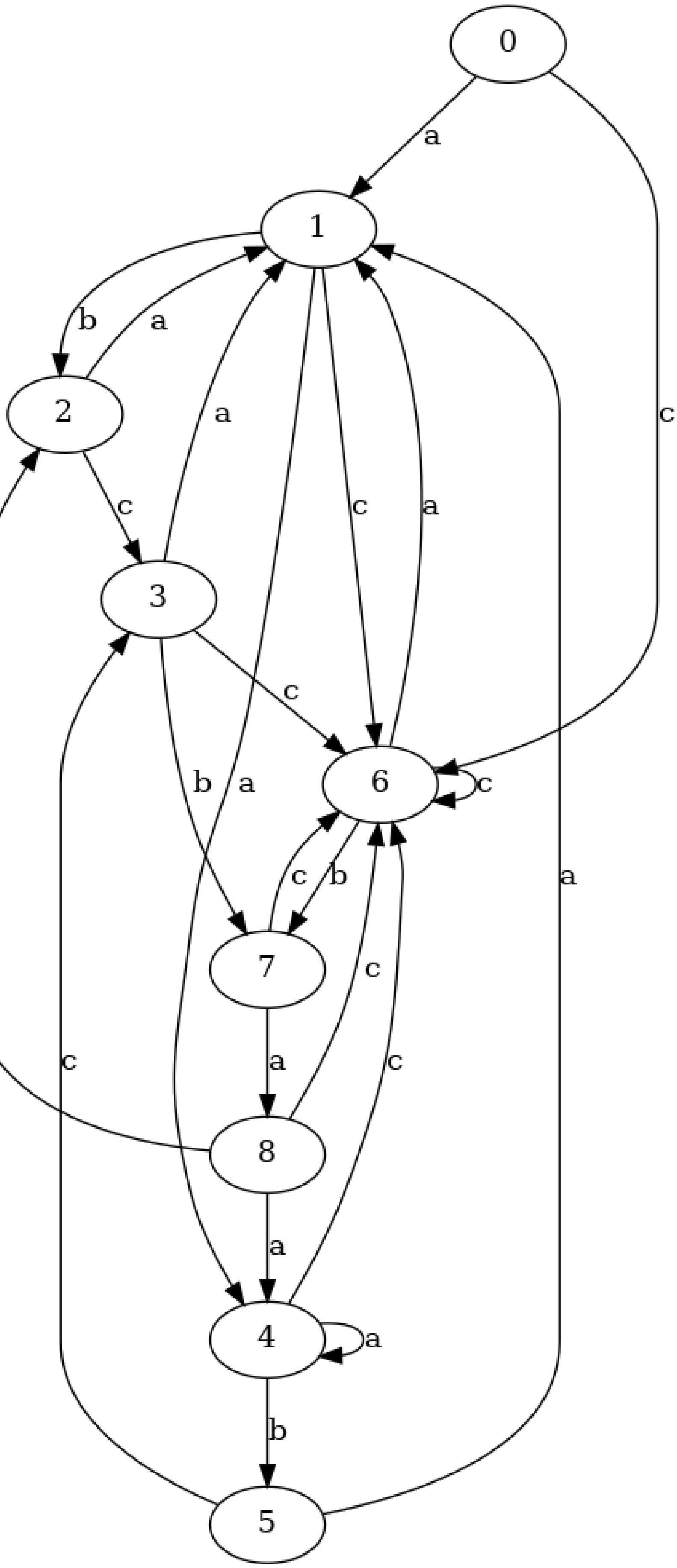
2 Realizacja poszczególnych poleceń

Kod do każdego z poleceń załączam na końcu sprawozdania.

2.1 Polecenie 1

Zaimplementowałem algorytm wyszukiwania wzorca 2d bazujący na automacie omawianym na ćwiczeniach. Poniżej załączam dwa zdjęcia: pierwsze to drzewo trie dla omawianego na zajęciach przykładowego wzorca, drugie to automat zbudowany na bazie tego drzewa (automat zupełny, bez pokazanej krawędzi domyślnej do stanu początkowego).




0
1
2
3
4
5
6
7
8
a
b
c

2.2 Polecenie 2

Użyłem zaimplementowanego algorytmu aby znaleźć wszystkie wystąpienia takich samych liter na takich samych pozycjach (za literę uznałem znak spełniający warunek .isalpha()). W poniższym wyniku działania programu każda krotka jest w formacie (linia, kolumna) i oznacza lewy górny róg wzorca:

```
'A': []
'B': []
'C': []
'D': []
'E': []
'F': []
'G': []
'I': []
'L': []
'M': []
'N': []
'O': []
'P': []
'Q': []
'R': []
'S': []
'T': []
'U': []
'V': []
'W': []
'X': []
'a': [(64, 2), (37, 4), (20, 6), (56, 11), (52, 12), (53, 12), (64,
14), (76, 21), (64, 22), (59, 24), (3, 30), (65, 35), (69, 35), (57,
36), (58, 36), (79, 37), (77, 42), (53, 48), (31, 50), (78, 59), (5,
60), (77, 61), (6, 63), (33, 66), (28, 69), (31, 73), (76, 74), (0,
82)]
'b': []
'c': [(41, 0), (68, 0), (13, 10), (82, 41), (10, 45), (3, 54)]
'd': [(37, 19)]
'e': [(10, 1), (14, 2), (24, 3), (17, 6), (76, 6), (77, 6), (80, 6),
(1, 8), (20, 10), (40, 11), (81, 14), (69, 15), (67, 17), (72, 23),
(40, 26), (18, 27), (73, 27), (51, 31), (42, 36), (29, 38), (71, 38),
(15, 43), (29, 43), (68, 46), (82, 47), (37, 48), (42, 48), (70, 49),
(47, 50), (58, 50), (46, 52), (22, 53), (57, 54), (58, 54), (41, 57),
(21, 61), (0, 63), (10, 64), (7, 65), (24, 65), (78, 65), (63, 66),
(28, 67), (65, 69), (66, 72), (28, 73), (59, 73), (4, 77)]
'f': [(77, 1), (30, 59)]
'g': []
'h': [(27, 2), (37, 2), (73, 12), (56, 31)]
'i': [(31, 0), (1, 5), (73, 13), (77, 13), (55, 17), (31, 31), (44,
33), (8, 37), (60, 45), (68, 51), (19, 55), (9, 60), (52, 69)]
'j': []
'k': []
'l': [(33, 45), (53, 45), (46, 61), (28, 72), (41, 77)]
'm': [(44, 0), (16, 5), (34, 40), (34, 60), (28, 70)]
'n': [(31, 1), (1, 9), (56, 13), (35, 18), (64, 29), (51, 32), (54,
```

```
33), (67, 35), (19, 37), (67, 40), (14, 54), (20, 56), (67, 57), (21,
62), (0, 83)]
'o': [(41, 1), (53, 1), (50, 2), (52, 8), (79, 10), (33, 11), (27,
17), (28, 17), (33, 26), (10, 27), (32, 34), (6, 38), (7, 38), (71,
42), (58, 45), (81, 52), (44, 55), (30, 58), (15, 60), (5, 66), (4,
75)]
'p': [(41, 18), (28, 71)]
'q': []
'r': [(1, 4), (52, 5), (33, 10), (7, 13), (17, 14), (15, 18), (69,
22), (43, 25), (67, 29), (60, 30), (33, 37), (47, 37), (6, 39), (62,
39), (55, 40), (46, 42), (6, 50), (19, 54), (20, 54), (28, 65), (31,
70)]
's': [(54, 0), (49, 14), (8, 21), (71, 24), (79, 24), (37, 34), (45,
34), (67, 37), (70, 41), (46, 44), (28, 45), (4, 49), (52, 53), (29,
56), (30, 56), (3, 57), (9, 58), (3, 63), (40, 63)]
't': [(37, 0), (50, 0), (16, 3), (71, 3), (72, 3), (23, 4), (24, 4),
(69, 5), (1, 6), (0, 7), (1, 7), (22, 8), (35, 10), (72, 10), (54,
11), (15, 12), (4, 14), (30, 16), (77, 22), (4, 23), (28, 23), (46,
24), (7, 29), (27, 31), (19, 33), (51, 33), (59, 33), (3, 37), (41,
45), (58, 49), (28, 52), (55, 54), (13, 55), (61, 56), (72, 59), (52,
61), (67, 71), (41, 73), (8, 75), (59, 75), (58, 78)]
'u': []
'v': []
'w': [(1, 3), (21, 70)]
'x': [(28, 68)]
'y': [(44, 5)]
'z': []
```

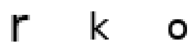
2.3 Polecenie 3

Załączam wynik działania, format jak w podpkt. 2:

```
th: []
t h: [(37, 0)]
```

2.4 Polecenie 4

Dokonałem wyszukiwania trzech wzorców:



Wyniki prezentuje jako haystack.png w którym znalezione wzorce są w negatywie, a dodatkowo wypisuje ilość znalezionych wzorców i ilość która powinna zostać znaleziona:

```
Found 298 occurences of r.png
Should find 339
Found 21 occurences of k.png
Should find 22
Found 310 occurences of o.png
Should find 369
```

2.4.1 r

One of the simplest and natural types of information representation is by means of written texts. This type of data is characterized by the fact that it can be written down as a long sequence of characters. Such linear a sequence is called a text. The texts are central in "word processing" systems, which provide facilities for the manipulation of texts. Such systems usually process objects that are quite large. For example, this book probably contains more than a million characters. Text algorithms occur in many areas of science and information processing. Many text editors and programming languages have facilities for processing texts. In biology, text algorithms arise in the study of molecular sequences. The complexity of text algorithms is also one of the central and most studied problems in theoretical computer science. It could be said that it is the domain in which practice and theory are very close to each other.

The basic textual problem in stringology is called pattern matching. It is used to access information and, no doubt, at this moment many computers are solving this problem as a frequently used operation in some application system. Pattern matching is comparable in this sense to sorting, or to basic arithmetic operations.

Consider the problem of a reader of the French dictionary " Grand Larousse," who wants all entries related to the name "Marie-Curie-Sklodowska." This is an example of a pattern matching problem, or string matching. In this case, the name "Marie-Curie-Sklodowska" is the pattern. Generally we may want to find a string called a pattern of length m inside a text of length n, where n is greater than m. The pattern can be described in a more complex way to denote a set of strings and not just a single word. In many cases n is very large. In genetics the pattern can correspond to a gene that can be very long; in image

The search of words or patterns in static texts is quite a different question than the previous pattern-matching mechanism. Dictionaries, for example, are organized in order to speed up the access to entries. Another example of the same question is given by indexes. Technical books often contain an index of chosen terms that gives pointers to parts of the text related to words in the index. The algorithms involved in the creation of an index form a specific group. The use of dictionaries or lexicons is often related to natural language processing. Lexicons of programming languages are small, and their representation is not a difficult problem during the development of a compiler. To the contrary, English contains approximately 100,000 words, and even twice that if inflected forms are considered. In French, inflected forms produce more than 700,000 words. The representation of lexicons of this size makes the problem a bit more challenging.

A simple use of dictionaries is illustrated by spelling checkers. The UNIX command, spell, reports the words in its input that are not stored in the lexicon. This rough approach does not yield a pertinent checker, but, practically, it helps to find typing errors. The lexicon used by spell contains approximately 70,000 entries stored within less than 60 kilobytes of random-access memory. Quick access to lexicons is a necessary condition for producing good parsers. The data structure useful for such access is called an index. In our book indexes correspond to data structures representing all factors of a given (presumably long) text. We consider problems related to the construction of such structures: suffix trees, directed acyclic word graphs, factor automata, suffix arrays. The PAT tool developed at the NOED Center (Waterloo, Canada) is an implementation of one of these structures tailored to work on large texts. There are several applications that effectively require some understanding of phrases in natural languages, such as data retrieval systems, interactive software, and character recognition.

An image scanner is a kind of photocopier. It is used to give a digitized version of an image. When the image is a page of text, the natural output of the scanner must be in a digital form available to a text editor. The transformation of a digitized image of a text into a usual computer representation of the text is realized by an Optical Character Recognition (OCR). Scanning a text with an OCR can be 50 times faster than retyping the text on a keyboard. Thus, OCR softwares are likely to become more common. But they still suffer from a high degree of imprecision. The average rate of error in the recognition of characters is approximately one percent. Even if this may happen to be rather small, this means that scanning a book produces approximately one error per line. This is compared with the usually very high quality of texts checked by specialists. Technical improvements on the hardware can help eliminate certain kinds of errors occurring on scanned texts in printed forms. But this cannot alleviate the problem associated with recognizing texts in printed forms. Reduction of the number of errors can thus only be achieved by considering the context of the characters, which assumes some understanding of the structure of the text. Image processing is related to the problem of two-dimensional pattern matching. Another related problem is the data structure for all subimages, which is discussed in this book in the context of the dictionary of basic factors.

The theoretical approach to the representation of lexicons is either by means of trees or finite state automata. It appears that both approaches are equally efficient. This shows the practical importance of the automata theoretic approach to text problems. At LITP (Paris) and IGM (Marne-la-Vallee) we have shown that the use of automata to represent lexicons is particularly efficient. Experiments have been done on a 700,000 word lexicon of LADL (Paris). The representation supports direct access to any word of the lexicon and takes only 300 kilobytes of random-access memory.

2.4.2 k

One of the simplest and n a t u r a l types of information representation is by means of written texts. This type of d a t a is characterized by t h e fact t h a t it can be written down as a long sequence of characters. Such linear a sequence is called a text. T h e texts are central in "word processing" systems, which provide facilities for t h e manipulation of texts. Such systems usually process objects t h a t are quite large. For example, this book probably contains more t h a n a million characters. Text algorithms occur in many areas of science and information processing. Many text editors and programming languages have facilities for processing texts. In biology, text algorithms arise in the study of molecular sequences. T h e complexity of text algorithms is also one of t h e central and most studied problems in theoretical computer science. It could be said t h a t it is the domain in which practice and theory are very close to each other.
T h e basic textual problem in stringology is called pattern matching. It is used to access information and, no doubt, at this moment m a n y computers are solving this problem as a frequently used operation in some application system. P a t t e r n matching is comparable in this sense to sorting, or to basic arithmetic operations.
Consider the problem of a reader of the French dictionary " G r a n d Larousse," who wants all entries related to the n a m e "Marie-Curie-Sklodowska." This is a n example of a p a t t e r n matching problem, or string matching. In this case, the n a m e "Marie-Curie-Sklodowska" is the p a t t e r n . Generally we m a y want to find a string called a pattern of length m inside a text of length n, where n is greater t h a n m. T h e p a t t e r n can be described in a more complex way to denote a set of strings and not just a single word. In many cases n is very large. In genetics the p a t t e r n can correspond to a gene t h a t can be very long; in image

T h e search of words or p a t t e r n s in static texts is quite a different question t h a n the previous pattern-matching mechanism. Dictionaries, for example, are organized in order to speed u p the access to entries. Another example of the same question is given by indexes. Technical books often contain a n index of chosen terms t h a t gives pointers to p a r t s of the text related to words in the index. T h e algorithms involved in the creation of an index form a specific group. T h e use of dictionaries or lexicons is often related t o n a t u r a l language processing. Lexicons of programming languages are small, and their representation is not a difficult problem during the development of a compiler. To the contrary, English contains approximately 100,000 words, a n d even twice t h a t if inflected forms are considered. In French, inflected forms produce more t h a n 700,000 words. T h e representation of lexicons of this size makes the problem a bit more challenging.
A simple use of dictionaries is illustrated by spelling checkers. T h e UNIX command, spell, reports the words in its input t h a t are not stored in the lexicon. This rough approach does not yield a pertinent checker, b u t , practically, it helps to find typing errors. T h e lexicon used by spell contains approximately 70,000 entries stored within less t h a n 60 kilobytes of random-access memory. Quick access to lexicons is a necessary condition for producing good parsers. T h e d a t a structure useful for such access is called an index. In our book indexes correspond to d a t a structures representing all factors of a given (presumably long) text. We consider problems related to the construction of such structures: suffix t r e e s , d i r e c t e d a c y c l i c w o r d g r a p h s , f a c t o r a u - t o m a t a , suffix arrays. T h e PAT
tool developed at the N O E D Center
(Waterloo, Canada) is an implementation of one of these structures tailored to work on large texts. There are several applications t h a t effectively require some understanding of phrases in n a t u r a l languages, such as d a t a retrieval systems, interactive software, a n d character recognition.
An image scanner is a kind of photocopier. It is used to give a digitized version of a n image. W h e n the image is a page of text, the n a t u r a l o u t p u t of the scanner must be in a digital form available to a text editor. T h e transformation of a digitized image of a text into a usual computer representation of the text is realized by a n Optical Character Recognition (O C R) . Scanning a text with an O C R can be 50 times faster t h a n retyping the text on a keyboard. T h u s , O C R softwares are likely to become more common. B u t they still suffer from a high degree of imprecision. T h e average rate of error in the recognition of characters is approximately one percent. Even if this may h a p p e n to be rather small, this means t h a t scanning a book produces approximately one error per line. This is compared with the usually very high quality of texts checked by specialists. Technical improvements on the hardware can help eliminate certain kinds of errors occurring on scanned texts in printed forms. But this cannot alleviate the problem associated with recognizing texts in printed forms. Reduction of the number of errors can thus only be achieved by considering the context of the characters, which assumes some understanding of the structure of the text. Image processing is related to the problem of two-dimensional pattern matching. Another related problem is the data structure for all subimages, which is discussed in this book in the context of the dictionary of basic factors.
The theoretical approach to the representation of lexicons is either by means of trees or finite state automata. It appears that both approaches are equally efficient. This shows the practical importance of the automata theoretic approach to text problems. At LITP (Paris) and IGM (Marne-la-Vallee) we have shown that the use of automata to represent lexicons is particularly efficient. Experiments have been done on a 700,000 word lexicon of LADL (Paris). The representation supports direct access to any word of the lexicon and takes only 300 kilobytes of random-access memory.

2.4.3 o

One of the simplest and n a t u r a l types of information representation is by means of written texts. This type of d a t a is characterized by t h e fact t h a t it can be written down as a long sequence of characters. Such linear a sequence is called a text. T h e texts are central in "word processing" systems, which provide facilities for t h e manipulation of texts. Such systems usually process objects t h a t are quite large. For example, this book probably contains more t h a n a million characters. Text algorithms occur in many areas of science and information processing. Many text editors and programming languages have facilities for processing texts. In biology, text algorithms arise in the study of molecular sequences. T h e complexity of text algorithms is also one of t h e central and most studied problems in theoretical computer science. It could be said t h a t it is the domain in which practice and theory are very close to each other.

T h e basic textual problem in stringology is called pattern matching. It is used to access information and, no doubt, at this moment m a n y computers are solving this problem as a frequently used operation in some application system. P a t t e r n matching is comparable in this sense to sorting, or to basic arithmetic operations.

Consider the problem of a reader of the French dictionary " G r a n d Larousse," who wants all entries related to the n a m e "Marie-Curie-Sklodowska." This is a n example of a p a t t e r n matching problem, or string matching. In this case, the n a m e "Marie-Curie-Sklodowska" is the p a t t e r n . Generally we m a y want to find a string called a pattern of length m inside a text of length n, where n is greater t h a n m. T h e p a t t e r n can be described in a more complex way to denote a set of strings and not just a single word. In many cases n is very large. In genetics the p a t t e r n can correspond to a gene t h a t can be very long; in image

T h e search of words or p a t t e r n s in static texts is quite a different question t h a n the previous pattern-matching mechanism. Dictionaries, for example, are organized in order to speed u p the access to entries. Another example of the same question is given by indexes. Technical books often contain a n index of chosen terms t h a t gives pointers to p a r t s of the text related to words in the index. T h e algorithms involved in the creation of an index form a specific group. T h e use of dictionaries or lexicons is often related t o n a t u r a l language processing. Lexicons of programming languages are small, and their representation is not a difficult problem during the development of a compiler. To the contrary, English contains approximately 100,000 words, a n d even twice t h a t if inflected forms are considered. In French, inflected forms produce more t h a n 700,000 words. T h e representation of lexicons of this size makes the problem a bit more challenging.

A simple use of dictionaries is illustrated by spelling checkers. T h e UNIX command, spell, reports the words in its input t h a t are not stored in the lexicon. This rough approach does not yield a pertinent checker, b u t , practically, it helps to find typing errors. T h e lexicon used by spell contains approximately 70,000 entries stored within less t h a n 60 kilobytes of random-access memory. Quick access to lexicons is a necessary condition for producing good parsers. T h e d a t a structure useful for such access is called an index. In our book indexes correspond to d a t a structures representing all factors of a given (presumably long) text. We consider problems related to the construction of such structures: suffix t r e e s , d i r e c t e d a c y c l i c w o r d g r a p h s , f a c t o r a u t o m a t a , suffix arrays. T h e PAT tool developed at the N O E D Center (Waterloo, Canada) is an implementation of one of these structures tailored to work on large texts. There are several applications t h a t effectively require some understanding of phrases in n a t u r a l languages, such as d a t a retrieval systems, interactive software, a n d character recognition.

An image scanner is a kind of photocopier. It is used to give a digitized version of a n image. W h e n the image is a page of text, the n a t u r a l o u t p u t of the scanner must be in a digital form available to a text editor. T h e transformation of a digitized image of a text into a usual computer representation of the text is realized by a n Optical Character Recognition (O C R) . Scanning a text with an O C R can be 50 times faster t h a n retyping the text on a keyboard. T h u s , O C R softwares are likely to become more common. B u t they still suffer from a high degree of imprecision. T h e average rate of error in the recognition of characters is approximately one percent. Even if this may h a p p e n to be rather small, this means t h a t scanning a book produces approximately one error per line. This is compared with the usually very high quality of texts checked by specialists. Technical improvements on the hardware can help eliminate certain kinds of errors occurring on scanned texts in printed forms. But this cannot alleviate the problem associated with recognizing texts in printed forms. Reduction of the number of errors can thus only be achieved by considering the context of the characters, which assumes some understanding of the structure of the text. Image processing is related to the problem of two-dimensional pattern matching. Another related problem is the data structure for all subimages, which is discussed in this book in the context of the dictionary of basic factors.

The theoretical approach to the representation of lexicons is either by means of trees or finite state automata. It appears that both approaches are equally efficient. This shows the practical importance of the automata theoretic approach to text problems. At LITP (Paris) and IGM (Marne-la-Vallee) we have shown that the use of automata to represent lexicons is particularly efficient. Experiments have been done on a 700,000 word lexicon of LADL (Paris). The representation supports direct access to any word of the lexicon and takes only 300 kilobytes of random-access memory.

2.5 Polecenie 5

Podobnie jak w poleceniu 4, prezentuje wynik ze wzorcem w negatywie:

One of the simplest and n a t u r a l types of information representation is by means
of written texts. This type of d a t a is characterized by t h e fact t h a t it can
be written down as a long sequence of characters. Such linear a sequence
is called a text. T h e texts are central in "word processing" systems, which
provide facilities for t h e manipulation of texts. Such systems usually process
objects t h a t are quite large. For example, this book probably contains more
t h a n a million characters. Text algorithms occur in many areas of science and
information processing. Many text editors and programming languages have
facilities for processing texts. In biology, text algorithms arise in the study
of molecular sequences. T h e complexity of text algorithms is also one of t h e
central and most studied problems in theoretical computer science. It could
be said t h a t it is the domain in which practice and theory are very close to
each other.
T h e basic textual problem in stringology is called pattern matching. It is
used to access information and, no doubt, at this moment m a n y computers
are solving this problem as a frequently used operation in some application
system. P a t t e r n matching is comparable in this sense to sorting, or to basic
arithmetic operations.
Consider the problem of a reader of the French dictionary " G r a n d Larousse,"
who wants all entries related to the n a m e "Marie-Curie-Sklodowska." This is
a n example of a p a t t e r n matching problem, or string matching. In this case,
the n a m e "Marie-Curie-Sklodowska" is the p a t t e r n . Generally we m a y want to
find a string called a pattern of length m inside a text of length n, where n is
greater t h a n m. T h e p a t t e r n can be described in a more complex way to denote
a set of strings and not just a single word. In many cases n is very large. In
genetics the p a t t e r n can correspond to a gene t h a t can be very long; in image

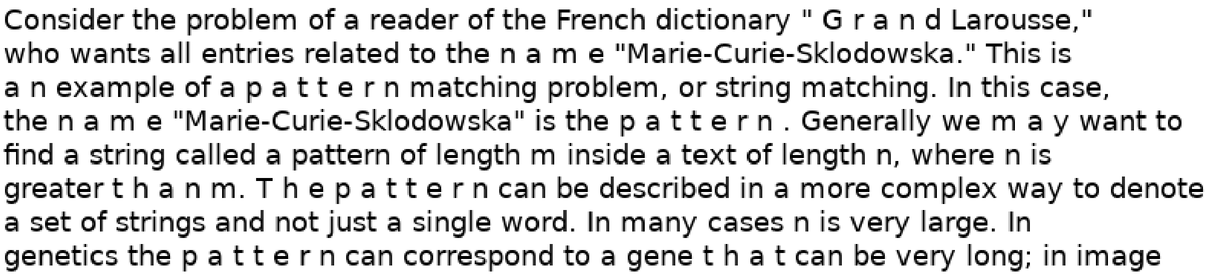
T h e search of words or p a t t e r n s in static texts is quite a different question
t h a n the previous pattern-matching mechanism. Dictionaries, for example,
are organized in order to speed u p the access to entries. Another example
of the same question is given by indexes. Technical books often contain a n
index of chosen terms t h a t gives pointers to p a r t s of the text related to words
in the index. T h e algorithms involved in the creation of an index form a
specific group. T h e use of dictionaries or lexicons is often related t o n a t u r a l
language processing. Lexicons of programming languages are small, and their
representation is not a difficult problem during the development of a compiler.
To the contrary, English contains approximately 100,000 words, a n d even twice
t h a t if inflected forms are considered. In French, inflected forms produce more
t h a n 700,000 words. T h e representation of lexicons of this size makes the
problem a bit more challenging.
A simple use of dictionaries is illustrated by spelling checkers. T h e UNIX
command, spell, reports the words in its input t h a t are not stored in the lexi-
con. This rough approach does not yield a pertinent checker, b u t , practically,
it helps to find typing errors. T h e lexicon used by spell contains approxi-
mately 70,000 entries stored within less t h a n 60 kilobytes of random-access
memory. Quick access to lexicons is a necessary condition for producing good
parsers. T h e d a t a structure useful for such access is called an index. In our
book indexes correspond to d a t a structures representing all factors of a given
(presumably long) text. We consider problems related to the construction of
such structures: suffix t r e e s , d i r e c t e d a c y c l i c w o r d g r a p h s , f a c t o r a u -
t o m a t a , suffix arrays. T h e PAT
tool developed at the N O E D Center
(Waterloo, Canada) is an implementation of one of these structures tailored
to work on large texts. There are several applications t h a t effectively require
some understanding of phrases in n a t u r a l languages, such as d a t a retrieval
systems, interactive software, a n d character recognition.
An image scanner is a kind of photocopier. It is used to give a digitized
version of a n image. W h e n the image is a page of text, the n a t u r a l o u t p u t of the
scanner must be in a digital form available to a text editor. T h e transformation
of a digitized image of a text into a usual computer representation of the text
is realized by a n Optical Character Recognition (O C R) . Scanning a text with
an O C R can be 50 times faster t h a n retyping the text on a keyboard. T h u s ,
O C R softwares are likely to become more common. B u t they still suffer from
a high degree of imprecision. T h e average rate of error in the recognition of
characters is approximately one percent. Even if this may h a p p e n to be rather
small, this means t h a t scanning a book produces approximately one error per
line. This is compared with the usually very high quality of texts checked
by specialists. Technical improvements on the hardware can help eliminate
certain kinds of errors occurring on scanned texts in printed forms. But this
cannot alleviate the problem associated with recognizing texts in printed forms.
Reduction of the number of errors can thus only be achieved by considering the
context of the characters, which assumes some understanding of the structure
of the text. Image processing is related to the problem of two-dimensional
pattern matching. Another related problem is the data structure for all
subimages, which is discussed in this book in the context of the dictionary
of basic factors.
The theoretical approach to the representation of lexicons is either by means
of trees or finite state automata. It appears that both approaches are equally
efficient. This shows the practical importance of the automata theoretic
approach to text problems. At LITP (Paris) and IGM (Marne-la-Vallee)
we have shown that the use of automata to represent lexicons is particularly
efficient. Experiments have been done on a 700,000 word lexicon of LADL
(Paris). The representation supports direct access to any word of the lexicon
and takes only 300 kilobytes of random-access memory.

2.6 Polecenie 6

Poniżej załączam wzorce dla których przeprowadziłem pomiary, po kolei: small.png, medium.png, big.png.

Rys. 1: small.png

Rys. 2: medium.png



Rys. 3: big.png

Poniżej załączam pomiary. Pomiary każdego przypadku przeprowadziłem 10-krotnie i wyciągnąłem z nich średnią:

```
=========Small pattern=========
Preprocessing time: 0.004592970500016236
Searching time: 1.0887323621998803
=========Medium pattern=========
Preprocessing time: 0.1500062629002059
Searching time: 0.9383168405998731
=========Big pattern=========
Preprocessing time: 2.218663644499975
Searching time: 0.9668237744997896
```

2.7 Polecenie 7

Załączam pomiary, podobnie jak w zadaniu 6, zostały powtórzone 10 razy, a w załączonych wynikach znajduje się średni czas wykonania.

```
=========Divided into 2 parts=========
Searching time: 0.9772503497999423
=========Divided into 4 parts=========
Searching time: 0.9691310410998994
=========Divided into 8 parts=========
Searching time: 0.9755429782999272
```

3 Kod

```python
from PIL import Image
import numpy as np
import pydot
import tempfile
from queue import Queue
from timeit import default_timer as timer


class Node:
    state_counter = 0

    def __init__(self, parent=None):
        self.state = Node.state_counter
        Node.state_counter += 1

        self.parent = parent
        self.children = dict()
        self.fail_link = parent

    # finds next state, using fail links if necessary
    def next_state(self, key):
        if self.parent == self:
            return self.children.get(key, self)
        return self.children.get(key, self.fail_link.next_state(key))

    def pretty_print(self, name=None, display=True):

        def dfs_helper(node, graph, visited):
            visited.add(node)
            for k, v in node.children.items():
                graph.add_edge(pydot.Edge(str(node.state), str(v.state), label=str(k)))
                if v not in visited:
                    dfs_helper(v, graph, visited)

        graph = pydot.Dot(graph_type='digraph')
        dfs_helper(self, graph, set())

        if name is None:
            fout = tempfile.NamedTemporaryFile(suffix=".png")
            name = fout.name
        else:
            name += "_trie.png"
        graph.write(name, format="png")
        if display:
            Image.open(name).show()


class PatternSearcher2d:
    neutral = '#'

```

```
    def __init__(self, pattern):
        self.pattern = self.convert_text(pattern, self.neutral)
        self.trie_root, self.accepting_states = None, None
        self.text = None

    def preprocess(self):
        self.trie_root, self.accepting_states = self.aho_corasick()
        return self

    def load_text(self, text):
        self.text = self.convert_text(text, self.neutral)
        return self

    # Neutral is a char non existent in the file we are going to scan
    # Only important when supplying a list of lines
    @staticmethod
    def convert_text(text, neutral='#'):
        if isinstance(text, list):   # Assumes text is a list of lines
            n = len(text)
            m = max([len(line) for line in text])
            arr = np.ndarray((n, m), dtype=object)
            for i in range(n):
                for j in range(m):
                    if j < len(text[i]):
                        arr[i, j] = text[i][j]
                    else:
                        arr[i, j] = neutral
            text = arr
        elif not isinstance(text, (np.ndarray)):
            Exception("Bad text format, has to be either list of lines or np.ndarray")
        # Assumes text is a numpy color array of shape (n,m,color_data)
        # where color_data is a list (so non_hashable)
        else:
            arr = np.ndarray(text.shape[:2], dtype=object)
            for i in range(text.shape[0]):
                for j in range(text.shape[1]):
                    arr[i, j] = tuple(text[i, j, :])   # 'Making' list hashable
            text = arr
        return text

    # pattern must be of type np.ndarray
    def aho_corasick(self, pattern=None):
        if pattern is None:
            pattern = self.pattern
        (n, m) = pattern.shape[:2]
        # Creating a simple trie
        trie = Node()
        trie.parent = trie
        trie.fail_link = trie
        accepting_states = []
        alphabet = set()
        for i in range(n):
            p = trie
```

```python
            for j in range(m):
                key = pattern[i, j]
                if key not in p.children:
                    alphabet.add(key)
                    p.children[key] = Node(p)
                p = p.children[key]
            accepting_states.append(p.state)
        # trie.pretty_print("trie")
        # Creating fail links
        # Starting bfs from nodes with dist = 2, (dist={0,1} fail links are already correct)
        Q = Queue()
        [Q.put((k, v)) for c in trie.children.values() for k, v in c.children.items()]
        while not Q.empty():
            k, v = Q.get()
            v.fail_link = v.parent.fail_link.next_state(k)
            for k2, v2 in v.children.items():
                Q.put((k2, v2))

        # Determinization of the automaton
        Q = Queue()
        Q.put(trie)
        while not Q.empty():
            v = Q.get()
            for u in v.children.values():
                Q.put(u)
            for k, u in v.fail_link.children.items():
                if k not in v.children:
                    v.children[k] = u

        # trie.pretty_print("automaton")
        return trie, accepting_states

    def search(self):
        n, m = self.text.shape[:2]
        pn, pm = self.pattern.shape[:2]
        states = np.zeros(self.text.shape)
        for i in range(n):
            node = self.trie_root
            for j in range(m):
                node = node.children.get(self.text[i, j], self.trie_root)
                states[i, j] = node.state
        pattern_vert = self.convert_text([self.accepting_states])
        root, accepting_state = self.aho_corasick(pattern_vert)
        accepting_state = accepting_state[0]
        found_coords = []
        for j in range(m):
            node = root
            for i in range(n):
                node = node.children.get(states[i, j], root)
                if node.state == accepting_state:
                    found_coords.append((i - pn + 1, j - pm + 1))
        return found_coords

```

```
    @staticmethod
    def search_wrapper(text, pattern):
        return PatternSearcher2d(pattern).preprocess().load_text(text).search()


def find_same_letter_pos(text):
    alphabet = set()
    for line in text:
        for letter in line:
            alphabet.add(letter)
    res = dict()
    for letter in alphabet:
        if letter.isalpha():
            res[letter] = PatternSearcher2d.search_wrapper(text, [letter, letter])
    for k, v in sorted(res.items()):
        print(repr(k) + ":", v)


def flip_colors(im, where, size=(20, 20)):
    im = im.copy()
    for x, y in where:
        for i in range(size[0]):
            for j in range(size[1]):
                for k in range(3):
                    im[x + i, y + j, k] = 255 - im[x + i, y + j, k]
    return Image.fromarray(im.astype(np.uint8))


def bench(text, pattern, reps=10):
    P = PatternSearcher2d(pattern)
    P.load_text(text)

    t1 = timer()
    for _ in range(reps):
        P.preprocess()
    t2 = timer()
    print("Preprocessing time:", (t2 - t1) / reps)

    t1 = timer()
    for _ in range(reps):
        P.search()
    t2 = timer()
    print("Searching time:", (t2 - t1) / reps)


def divide_and_search(image, pattern, parts, reps=10):
    P = PatternSearcher2d(pattern)
    P.load_text(image)
    P.preprocess()

    size = image.shape[0]
    parts = [
        PatternSearcher2d.convert_text(image[i * size // parts:(i + 1) * size // parts, :])
```

```
        for i in range(parts)
    ]

    t1 = timer()
    for _ in range(reps):
        for part in parts:
            P.text = part
            P.search()
    t2 = timer()
    print("Searching time:", (t2 - t1) / reps)


if __name__ == '__main__':
    im = np.asarray(Image.open('../sources/haystack.png'))
    # print(PatternSearcher2d.search_wrapper(['aaabcd', 'eeaab', 'ppcba'],
    #                                         ["abc", "aab", "cba"]))
    with open("../sources/haystack.txt", "r") as f:
        text = f.readlines()
        # Task2
        find_same_letter_pos(text)
        # Task3
        print("th:", PatternSearcher2d.search_wrapper(text, ['th', 'th']))
        print("t h:", PatternSearcher2d.search_wrapper(text, ['t h', 't h']))
        # Task4
        r = np.asarray(Image.open('../patterns/r.png'))
        k = np.asarray(Image.open('../patterns/k.png'))
        o = np.asarray(Image.open('../patterns/o.png'))
        where = PatternSearcher2d.search_wrapper(im, r)
        flip_colors(im, where, r.shape).show()
        print("Found", len(where), "occurences of r.png")
        print("Should find", len(PatternSearcher2d.search_wrapper(text, ["r"])))
        where = PatternSearcher2d.search_wrapper(im, k)
        flip_colors(im, where, k.shape).show()
        print("Found", len(where), "occurences of k.png")
        print("Should find", len(PatternSearcher2d.search_wrapper(text, ["k"])))
        where = PatternSearcher2d.search_wrapper(im, o)
        flip_colors(im, where, o.shape).show()
        print("Found", len(where), "occurences of o.png")
        print("Should find", len(PatternSearcher2d.search_wrapper(text, ["o"])))
        # Task5
        pattern = np.asarray(Image.open('../patterns/pattern.png'))
        where = PatternSearcher2d.search_wrapper(im, pattern)
        flip_colors(np.asarray(Image.open('../sources/haystack.png')),
                    where,
                    size=pattern.shape).show()
        # Task6
        print("=========Small pattern=========")
        bench(im, np.asarray(Image.open("../patterns/small.png")), 10)
        print("=========Medium pattern=========")
        bench(im, np.asarray(Image.open("../patterns/medium.png")), 10)
        print("=========Big pattern=========")
        bench(im, np.asarray(Image.open("../patterns/big.png")), 10)
        # Task7
```

```python
        for s in [2, 4, 8]:
            print("=========Divided into", s, "parts=========")
            divide_and_search(im, np.asarray(Image.open("../patterns/medium.png")), s)
```