

Алгоритм Ахо — Корасик

Пётр Смирнов

ВШЭ

29 ноября 2021 года

Бор как Map

Бор (aka префиксное дерево, aka trie) — реализация интерфейса Set/Map для ключей-строк

```
class Node:
    def __init__(self):
        # Алфавит = {0, 1, ..., 25}
        self.next = [None] * 26
        self.value = None
```

```
root = Node()
```

Бор как Map

Бор (aka префиксное дерево, aka trie) — реализация интерфейса Set/Map для ключей-строк

```
class Node:
    def __init__(self):
        # Алфавит = {0, 1, ..., 25}
        self.next = [None] * 26
        self.value = None

root = Node()

def set(S, value):
    cur = root
    for c in S:
        if cur.next[c] is None:
            cur.next[c] = Node()
        cur = cur.next[c]
    cur.value = value
```

Бор как Map

Бор (aka префиксное дерево, aka trie) — реализация интерфейса Set/Map для ключей-строк

```
class Node:
    def __init__(self):
        # Алфавит = {0, 1, ..., 25}
        self.next = [None] * 26
        self.value = None

root = Node()

def set(S, value):
    cur = root
    for c in S:
        if cur.next[c] is None:
            cur.next[c] = Node()
        cur = cur.next[c]
    cur.value = value

def get(S):
    cur = root
    for c in S:
        if cur.next[c] is None:
            return None
        cur = cur.next[c]
    return cur.value
```

Бор как Set

```
class Node:
    def __init__(self):
        # Алфавит = {0, 1, ..., 25}
        self.next = [None] * 26
        self.value = False
```

```
root = Node()
```

```
def add(S):
    cur = root
    for c in S:
        if cur.next[c] is None:
            cur.next[c] = Node()
        cur = cur.next[c]
    cur.value = True
```

```
def get(S):
    cur = root
    for c in S:
        if cur.next[c] is None:
            return False
        cur = cur.next[c]
    return cur.value
```

Бор как Set

```
class Node:
    def __init__(self):
        # Алфавит = {0, 1, ..., 25}
        self.next = [None] * 26
        self.value = False
```

```
root = Node()
```

```
def add(S):
    cur = root
    for c in S:
        if cur.next[c] is None:
            cur.next[c] = Node()
        cur = cur.next[c]
    cur.value = True
```

```
def get(S):
    cur = root
    for c in S:
        if cur.next[c] is None:
            return False
        cur = cur.next[c]
    return cur.value
```

Вершины с value = True будем называть *терминальными*

Задача

Дано множество (словарь) шаблонов (словарных слов) s_1, \dots, s_k и текст T . Найти вхождение хотя бы одного шаблона в текст.

- Уже получили алгоритм за $\mathcal{O}(|\Sigma| \cdot \sum_i |s_i| + |T| \cdot \max_i |s_i|)$
- Можно хранить переходы в хеш-таблице, тогда получим $\mathcal{O}(\sum_i |s_i| + |T| \cdot \max_i |s_i|)$
- Но хотим быстрее

Задача

Дано множество (словарь) шаблонов (словарных слов) s_1, \dots, s_k и текст T . Найти вхождение хотя бы одного шаблона в текст.

- Уже получили алгоритм за $\mathcal{O}(|\Sigma| \cdot \sum_i |s_i| + |T| \cdot \max_i |s_i|)$
- Можно хранить переходы в хеш-таблице, тогда получим $\mathcal{O}(\sum_i |s_i| + |T| \cdot \max_i |s_i|)$
- Но хотим быстрее
- Алгоритм Ахо — Корасик (Альфред Ахо, Маргарет Корасик, 1975)
- Идея та же, что в КМП: поддерживаем наибольший суффикс уже прочитанного текста, который совпадает с префиксом *какого-нибудь* словарного слова
- Но сначала надо сложить словарь в удобную структуру данных — бор

Алгоритм КМП

```
S = pattern + '$' + text
```

```
k = 0
```

```
# Нумерация символов с единицы
```

```
for i = 1..len(S):
```

```
    while k > 0 and S[i] != S[k + 1]:
```

```
        k = p[k]
```

```
    if S[i] == S[k + 1]:
```

```
        ++k
```

```
    p[i] = k
```

Число k — длина максимального суффикса уже прочитанного текста, совпадающего с префиксом шаблона

ababa \$ abbab

```
for pattern in patterns:  
    add(pattern)
```

```
cur = root
```

```
for c in text:
```

```
    cur = go(cur, c)
```

```
    if ''cur соответствует вхождению'':
```

```
        # обнаружено вхождение
```

cur — вершина бора, соответствующая суффиксу уже прочитанного текста, совпадающему с максимальным префиксом какого-нибудь из шаблонов

Осталось:

- написать функцию go
- понять, когда cur соответствует вхождению

- Суффиксные ссылки — обобщение префикс-функции
- Каждой вершине v бора соответствует строка по пути от корня до v , обозначим её $path(v)$

Определение

Пусть T — бор, v — вершина в нём. *Суффиксная ссылка v ($suf(v)$)* указывает на вершину u бора такую, что:

- $u \neq v$
- $path(u)$ является суффиксом $path(v)$;
- $|path(u)|$ максимальна

Для корня бора суффиксная ссылка не определена, поэтому удобно добавить фиктивный корень.

Наивная реализация *suf* и *go*

```
class Node:
    def __init__(self, parent, char):
        self.next = [None] * 26
        self.value = False
        self.parent = parent
        self.char = char

fake_root = Node(None, '')
root = Node(fake_root, '')
fake_root.next = [root] * 26
```

Наивная реализация *suf* и *go*

```
class Node:
    def __init__(self, parent, char):
        self.next = [None] * 26
        self.value = False
        self.parent = parent
        self.char = char

fake_root = Node(None, '')
root = Node(fake_root, '')
fake_root.next = [root] * 26

def suf(node):
    if node == root:
        return fake_root
    cur = suf(node.parent)
    while cur.next[node.char] is None:
        cur = suf(cur)
    return cur.next[node.char]
```

Наивная реализация *suf* и *go*

```
class Node:
    def __init__(self, parent, char):
        self.next = [None] * 26
        self.value = False
        self.parent = parent
        self.char = char

fake_root = Node(None, '')
root = Node(fake_root, '')
fake_root.next = [root] * 26

def suf(node):
    if node == root:
        return fake_root
    cur = suf(node.parent)
    while cur.next[node.char] is None:
        cur = suf(cur)
    return cur.next[node.char]
```

```
def go(cur, c):
    while cur.next[c] is None:
        cur = suf(cur)
    return cur.next[c]
```

За счёт добавления фиктивного корня код работает и для случая, когда максимальный суффикс пуст.

Наивная реализация *suf* и *go*

```
class Node:
    def __init__(self, parent, char):
        self.next = [None] * 26
        self.value = False
        self.parent = parent
        self.char = char

fake_root = Node(None, '')
root = Node(fake_root, '')
fake_root.next = [root] * 26

def suf(node):
    if node == root:
        return fake_root
    cur = suf(node.parent)
    while cur.next[node.char] is None:
        cur = suf(cur)
    return cur.next[node.char]
```

```
def go(cur, c):
    while cur.next[c] is None:
        cur = suf(cur)
    return cur.next[c]
```

За счёт добавления фиктивного корня код работает и для случая, когда максимальный суффикс пуст.

Корректность — как в префикс-функции.

```
def suf(node):
    if node == root:
        return fake_root
    cur = suf(node.parent)
    while cur.next[node.char] is None:
        cur = suf(cur)
    return cur.next[node.char]

def go(cur, c):
    while cur.next[c] is None:
        cur = suf(cur)
    return cur.next[c]
```

```
def suf(node):
    if node == root:
        return fake_root
    cur = suf(node.parent)
    while cur.next[node.char] is None:
        cur = suf(cur)
    return cur.next[node.char]

def go(cur, c):
    while cur.next[c] is None:
        cur = suf(cur)
    return cur.next[c]
```

Как посчитать *suf*, используя *go*?

```
def suf(node):
    if node == root:
        return fake_root
    cur = suf(node.parent)
    while cur.next[node.char] is None:
        cur = suf(cur)
    return cur.next[node.char]

def go(cur, c):
    while cur.next[c] is None:
        cur = suf(cur)
    return cur.next[c]
```

Как посчитать *suf*, используя *go*?

```
def suf(cur):
    if node == root:
        return fake_root
    return go(suf(node.parent), node.char)
```

go через *suf* и *go*

```
def suf(cur):  
    if node == root:  
        return fake_root  
    return go(suf(node.parent), node.char)  
  
def go(cur, c):  
    while cur.next[c] is None:  
        cur = suf(cur)  
    return cur.next[c]
```

go через *suf* и *go*

```
def suf(cur):  
    if node == root:  
        return fake_root  
    return go(suf(node.parent), node.char)  
  
def go(cur, c):  
    while cur.next[c] is None:  
        cur = suf(cur)  
    return cur.next[c]
```

Напишем *go* рекурсивно:

go через *suf* и *go*

```
def suf(cur):  
    if node == root:  
        return fake_root  
    return go(suf(node.parent), node.char)
```

```
def go(cur, c):  
    while cur.next[c] is None:  
        cur = suf(cur)  
    return cur.next[c]
```

Напишем *go* рекурсивно:

```
def go(cur, c):  
    if cur.next[c] is None:  
        return go(suf(cur), c)  
    return cur.next[c]
```

Добавим мемоизацию:

```
root.suf = fake_root
```

```
def suf(cur):
```

```
    if cur.suf is None:
```

```
        cur.suf = go(suf(node.parent), node.char)
```

```
    return cur.suf
```

```
def go(cur, c):
```

```
    if cur.go[c] is None:
```

```
        if cur.next[c] is None:
```

```
            cur.go[c] = go(suf(cur), c)
```

```
        else:
```

```
            cur.go[c] = cur.next[c]
```

```
    return cur.go[c]
```

Добавим мемоизацию:

```
root.suf = fake_root
def suf(cur):
    if cur.suf is None:
        cur.suf = go(suf(node.parent), node.char)
    return cur.suf

def go(cur, c):
    if cur.go[c] is None:
        if cur.next[c] is None:
            cur.go[c] = go(suf(cur), c)
        else:
            cur.go[c] = cur.next[c]
    return cur.go[c]
```

Корректность: всегда обращаемся к вершинам, которые находятся выше текущей.

- Мемоизацию в Python можно делать с помощью `@functools.cache` (но константа времени работы будет больше)
- Можно пойти другим путём: убрать функции, обойти бор BFS-ом и подсчитывать значения `suf` и `go[]`.

Вернёмся к схеме алгоритма:

```
for pattern in patterns:
    add(pattern)

cur = root
for c in text:
    cur = go(cur, c) # работает за "O(1)"
    if ''cur соответствует вхождению'':
        # обнаружено вхождение
```

Осталось:

- понять, когда cur соответствует вхождению

Определение

$sufpath(v) = \{v, suf(v), suf(suf(v)), \dots, root\}$. Иначе говоря,
 $sufpath(v) = \{v\} \cup sufpath(suf(v))$ для $v \neq root$ и
 $sufpath(root) = \{root\}$.

Определение

$sufpath(v) = \{v, suf(v), suf(suf(v)), \dots, root\}$. Иначе говоря,
 $sufpath(v) = \{v\} \cup sufpath(suf(v))$ для $v \neq root$ и
 $sufpath(root) = \{root\}$.

Утверждение

сир соответствует вхождению \Leftrightarrow на суффиксном пути *сир* есть терминальная вершина.

Определение

$sufpath(v) = \{v, suf(v), suf(suf(v)), \dots, root\}$. Иначе говоря,
 $sufpath(v) = \{v\} \cup sufpath(suf(v))$ для $v \neq root$ и
 $sufpath(root) = \{root\}$.

Утверждение

suf соответствует вхождению \Leftrightarrow на суффиксном пути suf есть терминальная вершина.

Таким образом, достаточно после построения бора для каждой вершины посчитать, есть ли в её суффиксном пути терминальная вершина:

```
v.has_terminal = v.value or suf(v).has_terminal
```

Вхождение словарного слова

Определение

$\text{sufpath}(v) = \{v, \text{suf}(v), \text{suf}(\text{suf}(v)), \dots, \text{root}\}$. Иначе говоря,
 $\text{sufpath}(v) = \{v\} \cup \text{sufpath}(\text{suf}(v))$ для $v \neq \text{root}$ и
 $\text{sufpath}(\text{root}) = \{\text{root}\}$.

Утверждение

сир соответствует вхождению \Leftrightarrow на суффиксном пути *сир* есть терминальная вершина.

Таким образом, достаточно после построения бора для каждой вершины посчитать, есть ли в её суффиксном пути терминальная вершина:

```
v.has_terminal = v.value or suf(v).has_terminal
```

Также бывает полезным посчитать сжатые суффиксные ссылки — ссылки до ближайшей терминальной вершины. Например, чтобы вывести все вхождения всех словарных слов.