Raport z laboratorium 1

Filip Nikolow

23 marca 2021

1 Cel laboratorium

Celem laboratorium było zapoznanie się z drzewami suffiksowymi - trie oraz skompresowanymi, a także z metodami ich konstrukcji.

2 Realizacja

Zaimplementowałem budowę drzew suffiksowych w trzech wersjach:

- Drzewo trie (pamięć: $O(n^2)$, obliczenia: $O(n^2)$)
- Skompresowane drzewo suffiksowe z wykorzystaniem tylko procedury slow_find do znajdywania głów (pamięć: $O(n)$, obliczenia: $O(n^2)$)
- Po długiej walce: algorytm McCreighta (pamięć: $O(n)$, obliczenia: $O(n)$)

Przy implementacji algorytmu McCreighta tegoroczny wykład był znacznie bardziej pomocny niż zeszłoroczny (być może jest to też kwestia tego, że już przed wtorkowym wykładem sporo nad tym algorytmem myślałem). Kluczowy natomiast do doszlifowania algorytmu i usunięcia bugów był niezawodny MIMUW :) (smurf.mimuw.edu.pl).

3 Realizacja poszczególnych poleceń

3.1 Polecenia 1-2

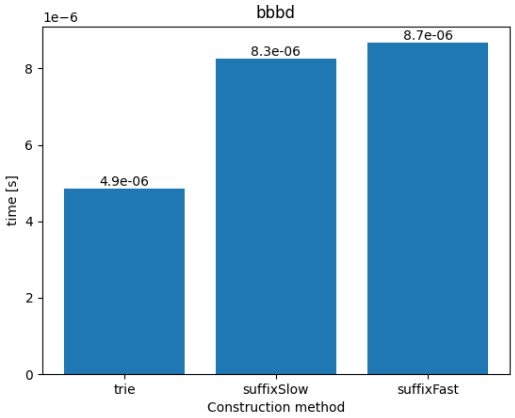
Łańcuchy 1-4 posiadają unikalny marker, natomiast do fragmentu ustawy na koniec dodałem znak Unicode "F000".

3.2 Polecenia 3-5

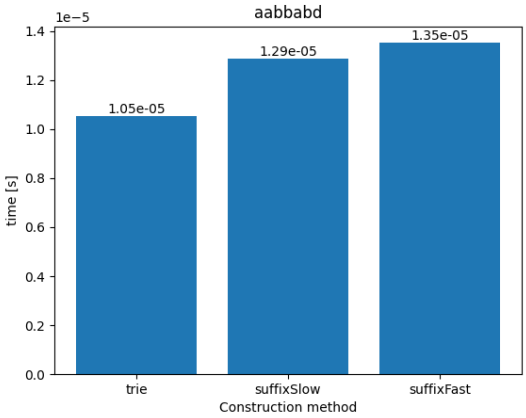
Zaimplementowałem algorytmy wyszczególnione w sekcji 2. Poprawność utworzonych struktur testowałem głównie wizualnie, poprzez porównanie drzewa trie z oczekiwanym, później porównanie drzewa trie z drzewem skompresowanym policzonym slow_findem, a następnie porównałem wyniki algorytmu ze slow_findem z algorytmem McCreighta. Nie wykryłem różnic - obrazki przedstawiające wyliczone drzewa dla przykładów z podzadania 1 znajdują się w sekcji "Drzewa". Dodatkowo załączam cały użyty kod na końcu raportu.

3.3 Testy porównujące szybkość działania algorytmów - polecenie 6

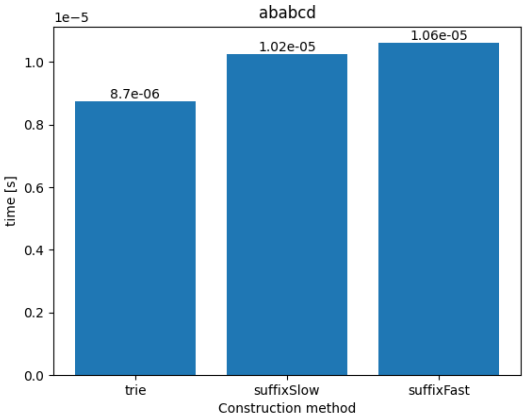
Poniżej załączam pomiary czasów działania każdego z trzech algorytmów: trie, algorytm ze slow_findem (suffixSlow), algorytm McCreighta (suffixFast). Testy przeprowadziłem na każdym z pięciu wymienionych w podzadaniu 1 tekstów. Każdy pomiar powtórzyłem 100-krotnie, aby zmniejszyć losowość przy krótkich tekstach.



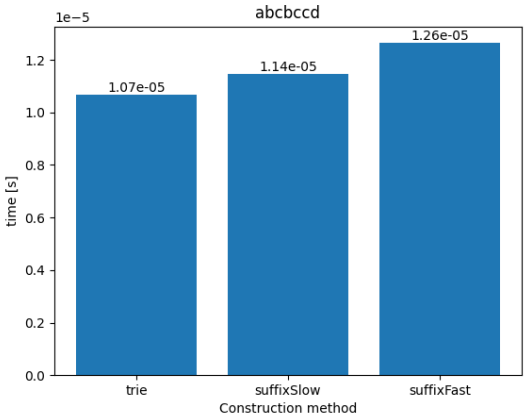
Rys. 1



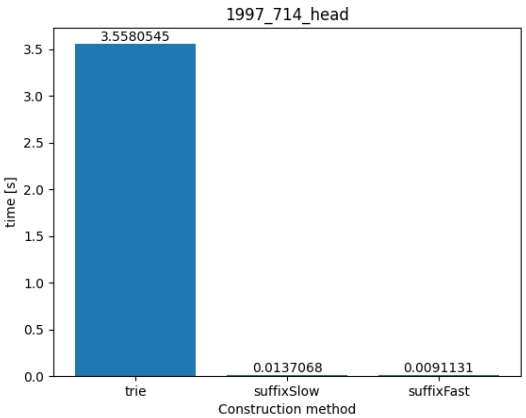
Rys. 2



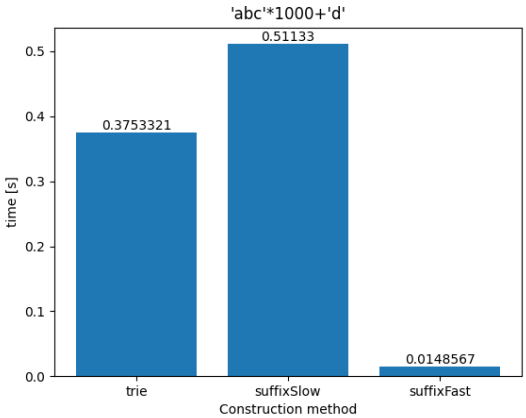
Rys. 3



Rys. 4



Rys. 5



Rys. 6

Rysunki od 1 do 4 pokazują, że na tak krótkich testach mała stała w algorytmie budującym trie jest bardziej znacząca niż dobra złożoność obliczeniowa i algorytm budujący trie wygrywa, natomiast algorytm McCreighta, właśnie przez większą stałą, jest ostatni. Na rysunku 5 kwadratowa złożoność algorytmu budującego trie jest już bardzo widoczna, natomiast ciekawe jest to, że sufixSlow tylko nieznacznie przegrywa z McCreightem - trie oprócz wysokiej złożoności ma także duży narzut związany z tworzeniem bardzo wielu obiektów (Node'ów), ale mimo to

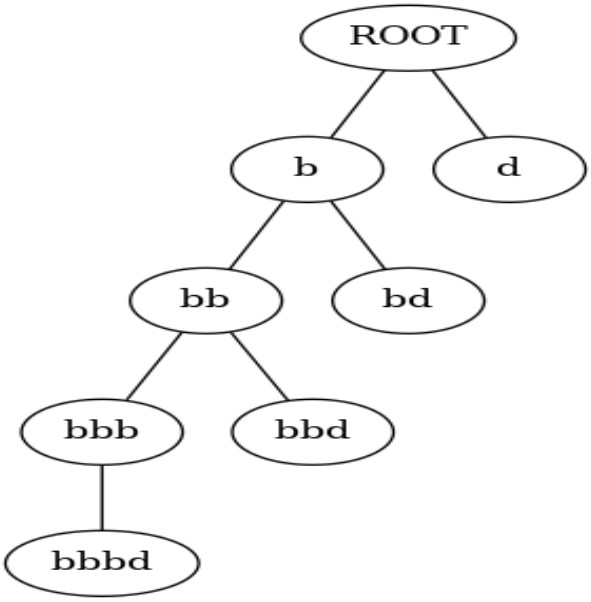
ewidentnie przykład ustawy jest korzystny dla funkcji slow_find (dodatkowo pewnie optymalizacje operacji na stringach w Pythonie pomagają) i linki w McCreighcie nie dają wiele - pomimo to McCreight jest szybszy mimo większej stałej. Aby sprawdzić czy faktycznie suffixSlow ma kwadratową złożoność (a zaimplementowany McCreight liniową) przeprowadziłem dodatkowy test z rysunku 6 - tam już wyniki są takie jakich można oczekiwać od strony teoretycznej :).

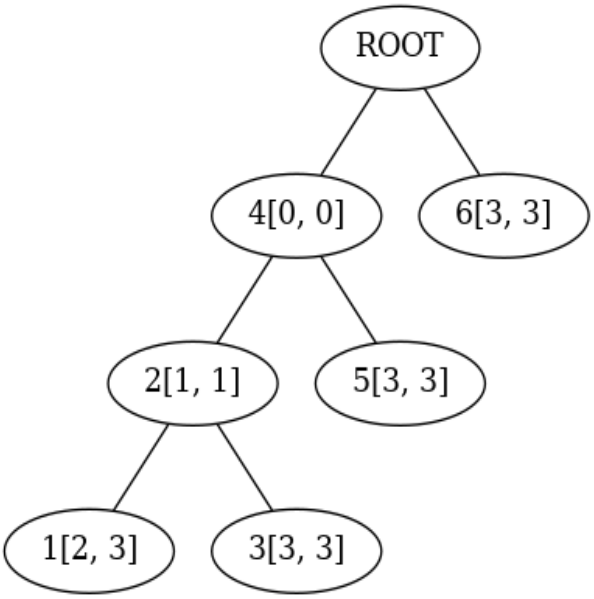
4 Szczegóły techniczne platformy testowej

Wszystkie pomiary czasów zostały przeprowadzone na maszynie wirtualnej VirtualBox z systemem Ubuntu 20. Używana wersja języka Python to 3.8.5. Użyty procesor to AMD Ryzen 3900X.

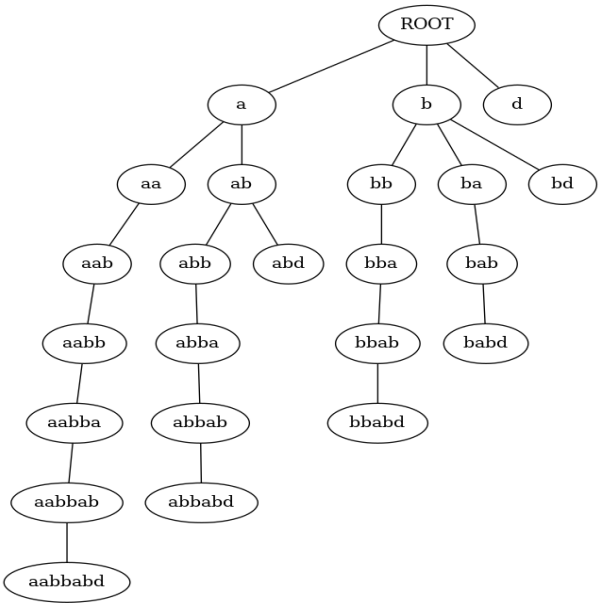
5 Drzewa

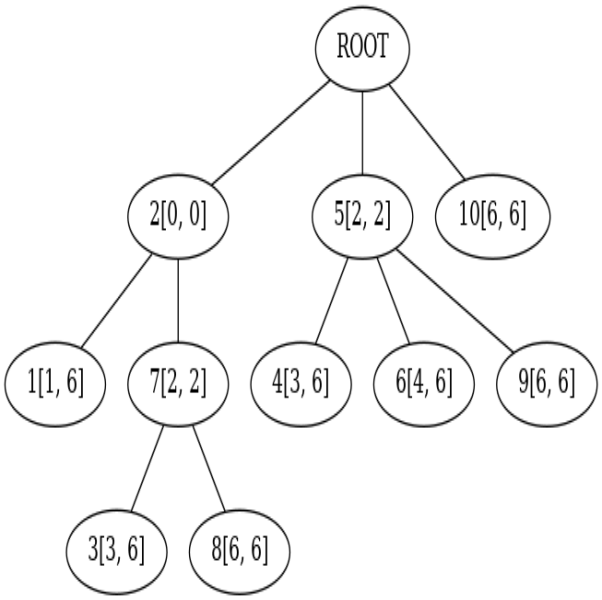
5.1 bbbd



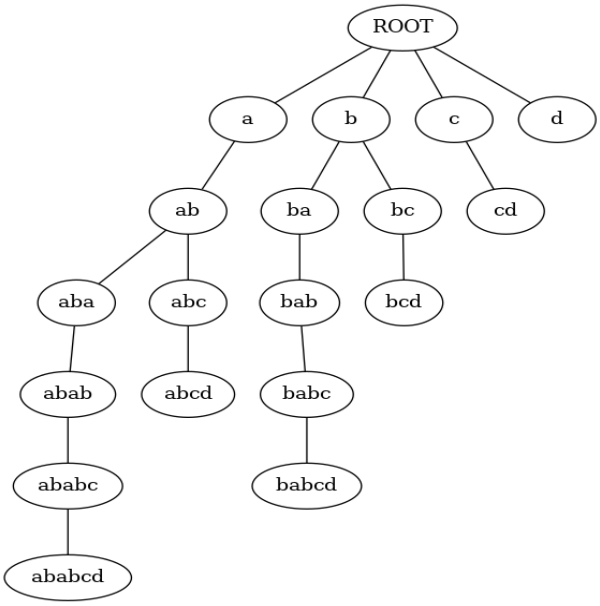


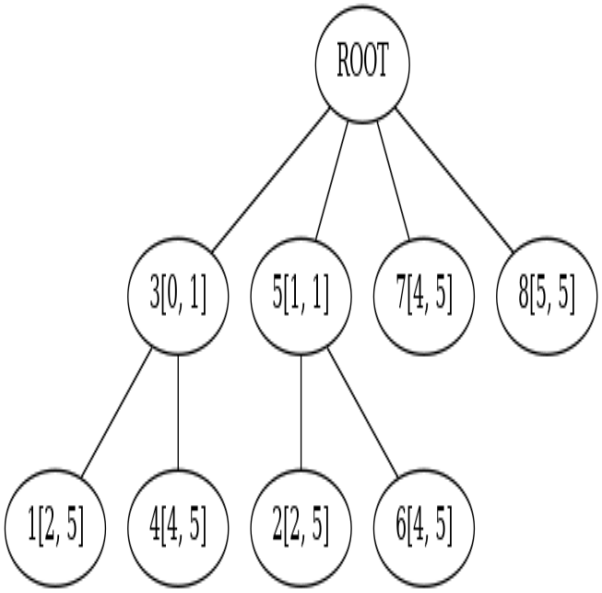
5.2 aabbabd



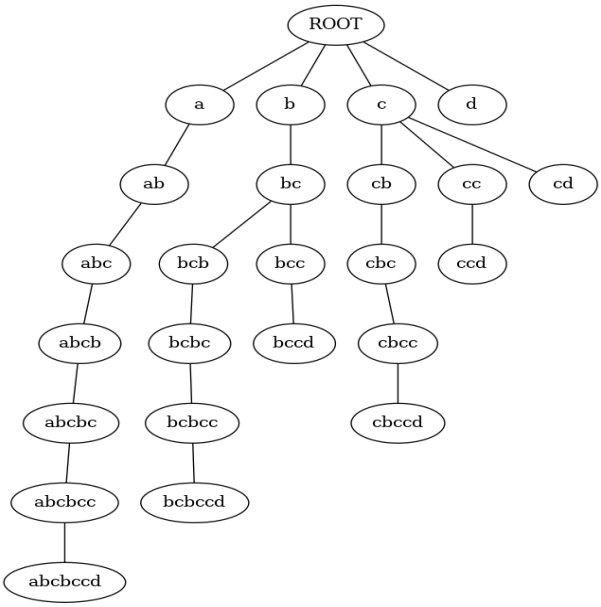


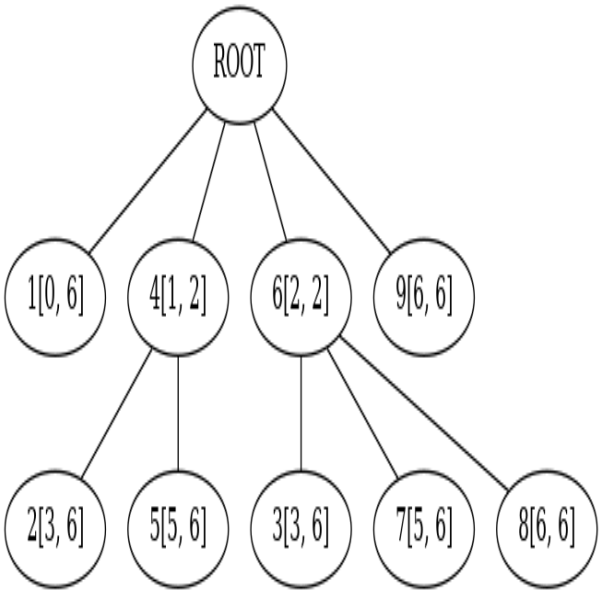
5.3 ababcd





5.4 abcdccd





6 Kod programów

6.1 Trie

```
1  import pydot
2  import tempfile
3  from PIL import Image
4
5
6  class Node:
7
8      def __init__(self):
9          self.children = dict()
10
11
12 def create_trie(text):
13     trie = Trie()
14     for i in range(len(text)):
15         trie.insert(text[i:])
16     return trie
17
18
19 class Trie:
20
21     def __init__(self):
22         self.root = Node()
23
24     def pretty_print(self, name=None, display=True):
25
26         def dfs_helper(node, label, graph):
27             for c in node.children.keys():
28                 graph.add_edge(pydot.Edge(label, label + c))
```

```
                dfs_helper(node.children[c], label + c, graph)

        graph = pydot.Dot(graph_type='graph')
        for c in self.root.children.keys():
            graph.add_edge(pydot.Edge("ROOT", c))
            dfs_helper(self.root.children[c], c, graph)

        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()

    def insert(self, string):
        p = self.root
        for i, c in enumerate(string):
            if c in p.children:
                p = p.children[c]
            else:
                break
        for j in range(i, len(string)):
            c = string[j]
            p.children[c] = Node()
            p = p.children[c]

    def search(self, string):
        p = self.root
        for c in string:
            if c in p.children:
                p = p.children[c]
            else:
                return False
        return True


def create_treeplots(textlist):
    for text in textlist:
        T = create_trie(text)
        T.pretty_print(text, False)


if __name__ == '__main__':
    create_treeplots(["bbbd", "aabbabd", "ababcd", "abcbccd"])
    # T = create_trie("aabbabbc")
    # print(T.search("tki"))
    # print(T.search("tkitek"))
    # print(T.search("krotkitekst"))
    # print(T.search("otko"))
    # print(T.search("otkt"))
    # T.pretty_print()
```

6.2 suffixSlow oraz McCreight

```
import pydot
import tempfile
from PIL import Image

ind = 0


class Node:

    def __init__(self, label):
        global ind
        self.ind = ind
        # print(ind)
        ind += 1
        self.label = label
        self.letter = None
        self.link = None
        self.parent = None
        self.children = dict()

    def connect(self, child, key):
        self.children[key] = child
        child.parent = self
        child.letter = key
        return child

    def length(self):
        return self.label[1] - self.label[0] + 1


class suffixTree:

    def __init__(self, text):
        self.text = text
        self.root = Node(None)
        self.root.parent = self.root
        self.root.link = self.root

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

        def dfs_helper(node, graph):
            for c in node.children.values():
                graph.add_edge(
                    pydot.Edge(str(node.ind) + str(node.label),
                               str(c.ind) + str(c.label)))
                dfs_helper(c, graph)

        graph = pydot.Dot(graph_type='graph')
```

```
        for c in self.root.children.values():
            graph.add_edge(pydot.Edge("ROOT", str(c.ind) + str(c.label)))
            dfs_helper(c, graph)
        if name is None:
            fout = tempfile.NamedTemporaryFile(suffix=".png")
            name = fout.name
        else:
            name += "_tree.png"
        graph.write(name, format="png")
        if display:
            Image.open(name).show()

    def split(self, node, key, index):
        child = node.children[key]
        new_node = Node([child.label[0], index])
        child.label[0] = index + 1
        new_node.connect(child, self.text[index + 1])
        node.connect(new_node, key)
        return new_node

    def fast_find(self, i, j, node):
        # print("FAST")
        if i > j:
            return node
        child = node.children[self.text[i]]
        if (j - i + 1 > child.length()):
            return self.fast_find(i + child.length(), j, child)
        elif (j - i + 1 == child.length()):
            return child
        else:
            return self.split(node, self.text[i], child.label[0] + j - i)

    def slow_find(self, i, node):
        # print("SLOW")
        key = self.text[i]
        if key not in node.children:
            return (node, i)
        child = node.children[key]
        start, end = child.label
        while start <= end:
            if self.text[i] != self.text[start]:
                return (self.split(node, key, start - 1), i)
            i += 1
            start += 1
        return self.slow_find(i, child)

    def find(self, pattern, i, node):
        key = pattern[i]
        if key not in node.children:
            return False
        child = node.children[key]
        start, end = child.label
        while start <= end and i < len(pattern):
```

```
            if pattern[i] != self.text[start]:
                return False
            i += 1
            start += 1
        if i == len(pattern):
            return True
        return self.find(pattern, i, child)

    def mc_creight(self):
        m = len(self.text)
        last_head = head = self.root
        leaf = self.root.connect(Node([0, m - 1]), self.text[0])
        for i in range(1, m):
            p = last_head.label
            q = leaf.label
            if last_head == self.root:
                head, j = self.slow_find(q[0] + 1, self.root)
            else:
                u = last_head.parent
                if u == self.root:
                    v = self.fast_find(p[0] + 1, p[1], self.root)
                    j = q[0]
                else:
                    # print("LINK")
                    v = self.fast_find(p[0], p[1], u.link)
                    j = q[0]
                if len(v.children) == 1:
                    head = v
                else:
                    head, j = self.slow_find(q[0], v)
                last_head.link = v
            # print(j, m - 1)
            leaf = head.connect(Node([j, m - 1]), self.text[j])
            last_head = head

    def mc_creight_slow(self):
        m = len(self.text)
        for i in range(m):
            head, j = self.slow_find(i, self.root)
            head.connect(Node([j, m - 1]), self.text[j])


def suffixTreeFastWrapper(text):
    T = suffixTree(text)
    T.mc_creight()


def suffixTreeSlowWrapper(text):
    T = suffixTree(text)
    T.mc_creight_slow()


def create_treeplots(textlist):
```

```
    for text in textlist:
        global ind
        ind = 0
        T = suffixTree(text)
        T.mc_creight()
        T.pretty_print(text, False)


if __name__ == '__main__':
    create_treeplots(["bbbd", "aabbabd", "ababcd", "abcbccd"])
    # with open("ustawa.txt", "r") as f:
    #     T = suffixTree(f.read() + "\0")
    #     T.mc_creight()
    #     # T.pretty_print()
    #     print(T.find("dochodów2137", 0, T.root))
```

6.3 Testowanie, generowanie wykresów itp.

```
from mc_creight_final_attempt import suffixTreeFastWrapper, suffixTreeSlowWrapper
from trie import create_trie
from time import time
import matplotlib.pyplot as plt


def benchmark(funclist,
              datlist,
              labels,
              titles,
              axis_labels=None,
              repeats=None,
              save_to_file=False):
    if repeats is None:
        repeats = 1
    for dat, title in zip(datlist, titles):
        times = dict()
        for func, label in zip(funclist, labels):
            for _ in range(repeats):
                t1 = time()
                func(dat)
                t2 = time()
                times[label] = times.get(label, 0) + (t2 - t1) / repeats
        plt.bar(times.keys(), times.values())
        plt.title(title)
        for i, v in enumerate(times.values()):
            plt.annotate(str(round(v, 7)), xy=(i, v), ha='center', va='bottom')
        if axis_labels is not None:
            plt.xlabel(axis_labels['x'])
            plt.ylabel(axis_labels['y'])

        if save_to_file:
            plt.savefig(title + ".png")
```

```
        plt.show()


if __name__ == '__main__':
    with open("1997_714_head.txt") as ustawa:
        funclist = [create_trie, suffixTreeSlowWrapper, suffixTreeFastWrapper]
        datlist = [
            "bbbd", "aabbabd", "ababcd", "abcbccd",
            ustawa.read()[:2000] + "\uF000", "abc" * 1000 + "d"
        ]
        labels = ["trie", "suffixSlow", "suffixFast"]
        titles = [s for s in datlist]
        titles[-2] = "1997_714_head"
        titles[-1] = "\'abc\'*1000+'d'"
        axis_labels = {'x': 'Construction method', 'y': 'time [s]'}
        benchmark(funclist, datlist, labels, titles, axis_labels, 100, True)
```