task45_Zhukov_vlad

December 17, 2016

0.0.1 See src.py, test.py files for algorithm code

In [ ]:
```
import src
import test
import matplotlib.pyplot as plt
```

1 Let's test algorithm

See test.py

In [ ]:
```
import unittest
suite = unittest.TestLoader().loadTestsFromTestCase(test.Tests)
unittest.TextTestRunner(verbosity=2).run(suite)
```

1.1 Example of usage

In [ ]:
```
strings = ['cde', 'abc', 'eab', 'fgh', 'ghf', 'hed']
algo = src.superstring4(strings)
print('superstring4: ' + str(algo.solve()))
print ('greedy:       ' + \
       str(src.greedy_min_max_contain_string(strings)))
print ('answer:       ' + \
       str(src.min_max_contain_string(set(strings))))
                         #input must be set of strings
```

In [ ]:
```
from itertools import combinations
from itertools import combinations_with_replacement
from itertools import permutations
from itertools import product

def short_string_test(n_words, words):
    for c in combinations(words, n_words):
        l1 = len(src.greedy_min_max_contain_string(c))
        l2 = len(src.min_max_contain_string(set(c)))
        res.append((1.0 *l1) / l2)
    return res
```

2 Let's "make sure" that aproximation ratio is equal 2 for short strings(a.r.>=2)

It takes some time to find right answers

```
In [ ]: letters = 'abcd'
        tmp = [map(''.join, product(letters, repeat=length))\
                for length in range(1, 4)]
        words = [x for n in tmp for x in n]
        res = []
        sst = short_string_test(3, words)
        print(max(sst))
```

For sentences with two words, where words consist of $1, 2, 3$ letters aproximation ratio is ~<= 1.5

Let's implement test described in: http://www.mimuw.edu.pl/~mucha/teaching/aa2008/ss.pdf (2.2 The greedy algorithm)

```
In [ ]: def generate_worst_test(k):
            return ['a' + 'b' * k, 'b' * k + 'c', 'b' * (k + 1)]
        test = generate_worst_test(4)
        res = list()
        for c in [generate_worst_test(i) for i in range(4, 300)]:
            l1 = src.greedy_min_max_contain_string(c)
            l2 = src.min_max_contain_string(set(c))
            res.append(((1.0 *len(l1)) / len(l2)))
        print (max(res))
```

```
In [ ]: plt.figure(figsize=(10, 5))
        plt.scatter(range(len(res)), res)
        plt.show()
```

So we can see on $\{ab^k, b^kc, b^{k+1}\}$ tests algorithm's aproximation ratio converges to 2. We have a little bit better algorighm than in article(in article assumes that strings can not contain each other) that merges strings in one, if one contains another

3 Testing graph algorithm

```
In [ ]: test = generate_worst_test(4)
        res = list()
        for c in [generate_worst_test(i) for i in range(4, 300)]:
            algo = src.superstring4(c)
            l1 = algo.solve()
            l2 = src.min_max_contain_string(set(c))
            res.append(((1.0 *len(l1)) / len(l2)))
        print (max(res))
```

```
In [ ]: plt.figure(figsize=(10, 5))
        plt.scatter(range(len(res)), res)
        plt.show()
```

Small strings tests

```
In [ ]: letters = 'abcd'
        tmp = [map(''.join, product(letters, repeat=length))\
                for length in range(1, 4)]
        words = [x for n in tmp for x in n]
        res = []
        for c in combinations(words, 3):
                algo = src.superstring4(c)
                l1 = len(algo.solve())
                l2 = len(src.min_max_contain_string(set(c)))
                res.append((1.0 *l1) / l2)
        print(max(res))
```

Worse than greedy

```
In [ ]: res = []
        a = ""
        b = ""
        c = ""
        d = ""
        e = ""
        f = ""
        for k in range(100):
            tst = []
            a = a + "a"
            b = b + "b"
            c = c + "c"
            d = d + "d"
            e = e + "e"
            f = f + "f"
            x = "x"
            tst.append(a + x + b)
            tst.append(b + x + c)
            tst.append(c + x + d)
            tst.append(d + x + e)
            tst.append(e + x + f)
            tst.append(b[:-1] + x + a + x)
            tst.append(c[:-1] + x + b + x)
            tst.append(d[:-1] + x + c + x)
            tst.append(e[:-1] + x + d + x)
            tst.append(f[:-1] + x + e + x)
            algo = src.superstring4(tst)
        #     print(tst)
        #     algo = src.superstring4(tst)
            l1 = len(algo.solve())
        #     print (algo.solve())
            l2 = 9 * k + 1
```

```
        #     l2 = len(src.min_max_contain_string(set(tst)))
            res.append((1.0 * l1) / l2)
```

```
In [ ]: plt.figure(figsize=(10, 5))
        plt.scatter(range(len(res)), res, s=10)
        # plt.ylim((0, 5))
        plt.show()
```

3.0.1 DOT GRAPH

```
In [ ]: strings2 = ['abc', 'bcd', 'daa', 'bcc']
        for i_id, i in enumerate(strings2):
            print ("v" + str(i_id) +" [shape=box, label=\"" + i + "\"]")
        for i_id, i in enumerate(strings2):
            for j_id, j in enumerate(strings2):
                print ("v" + str(i_id) + "->" + "v" + str(j_id) + " [label=\""+ str
```