Approximating the shortest superstring & recap

Algorithms on Strings

Paweł Gawrychowski

July 17, 2013

Outline

In all problems considered so far we were looking for polynomial algorithms. Well, in fact anything larger than linear was frowned upon. Alas, real world ain't that nice.
Consider the following question. There is an unknown sequence that we would like to learn. Using a process known as shotgun sequencing, we extract a bunch of its (probably short) substrings.

```
AGGTGGTC

AGGTG...
....GGTC
.GGTG...
```

Given the substrings, how do we reconstruct the sequence?

Shortest superstring

Input: a collection of strings $s_1, s_2, \ldots, s_n \in \Sigma^+$ Output: shortest word w such that all s_i are substrings of w.

Why shortest?

Wishful thinking. Doesn't really have to be our original string, but with enough data we hope the answer to be unique that way.

Simplifying assumption

No s_i is a substring of some other s_j.

Do you see why?

Now the sad thing is that the problem is NP-hard, so we can't hope to solve it efficiently. So the story is over?

Not quite. There are two ways around:

- relax the requirement that the algorithm should work in polynomial time,
- relax the requirement that the algorithm should produce the shortest solution.

Now the sad thing is that the problem is NP-hard, so we can't hope to solve it efficiently. So the story is over?

Not quite. There are two ways around:

- relax the requirement that the algorithm should work in polynomial time,
- relax the requirement that the algorithm should produce the shortest solution.

Now the sad thing is that the problem is NP-hard, so we can't hope to solve it efficiently. So the story is over?

Not quite. There are two ways around:

- relax the requirement that the algorithm should work in polynomial time,
- relax the requirement that the algorithm should produce the shortest solution.

Shortest superstring can be solved in $\mathcal{O}(2^n n^2)$ time and $\mathcal{O}(2^n n)$ space. The space can be improved to polynomial.

Now the sad thing is that the problem is NP-hard, so we can't hope to solve it efficiently. So the story is over?

Not quite. There are two ways around:

- relax the requirement that the algorithm should work in polynomial time,
- relax the requirement that the algorithm should produce the shortest solution.

Today's lecture.

Approximation algorithm

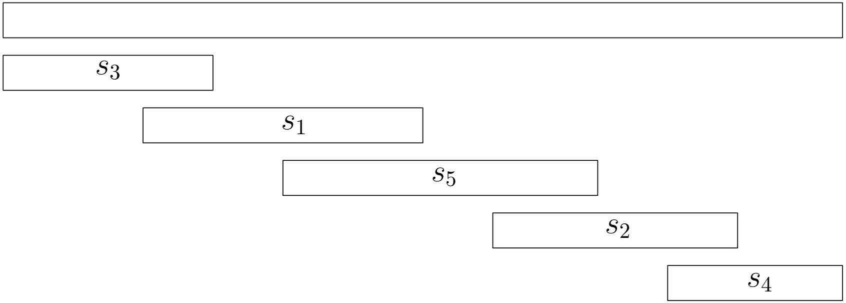
We are interested in constructing an algorithm which will find (in polynomial time) a superstring of length at most $c * OPT$, where OPT is the shortest possible superstring. We call such procedure a c-approximation algorithm.

Large c is boring and useless. For instance, $c = n$ is very simple to achieve. So, we would like c to be a constant, and hopefully a small one.

Theorem

There is a (polynomial time) 4-approximation algorithm for shortest superstring.

First we need some insight into the structure of the optimal solution. Take the shortest superstring, and mark the first occurrence of each s_i.



$$OPT = \sum_{i} |s_i| - \sum_{i>1} |\text{overlap}(s_{j_{i-1}}, s_{j_i},)|$$

What is overlap(s, t)? It is the longest suffix of s that is also a prefix of t.

What is overlap($\texttt{aabaa}, \texttt{baaba}$)?

We will also need to know what is prefix(s, t). It is the prefix of s left after removing the suffix corresponding to overlap(s, t).

What is prefix($\texttt{aabaa}, \texttt{baaba}$)?

Let's first look at a very natural method. As long as we have more than two strings, choose s_i and s_j with the longest overlap(s_i, s_j), and replace the two strings with their merge prefix$(s_i, s_j)s_j$. This is known as the Greedy Algorithm.

Blum, Jiang, Li, Tromp, and Yannakakis 1991

Greedy is a 4-approximation algorithm.

Kaplan and Shafrir 2005

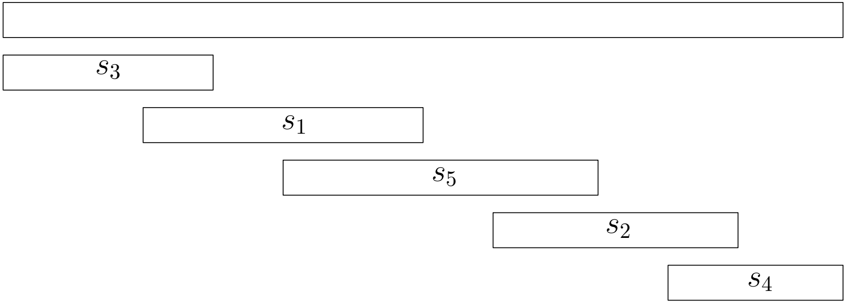
Well, Greedy is actually a 3.5-approximation algorithm.

Conjecture

Greedy is a 2-approximation algorithm.

Prove this and you will be very famous ☺

We will analyse an algorithm which is maybe less natural, but easier to reason about. First we need to look at *OPT* in a slightly way.

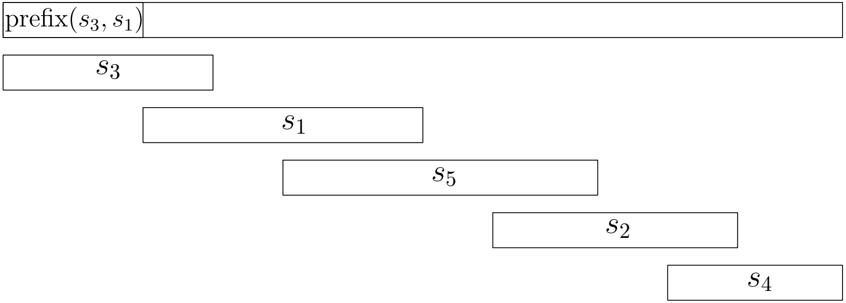


This is not very symmetric, so we additionally observe that

$$s_4 = \text{prefix}(s_4, s_3)\text{overlap}(s_4, s_3)$$

.

We will analyse an algorithm which is maybe less natural, but easier to reason about. First we need to look at *OPT* in a slightly way.

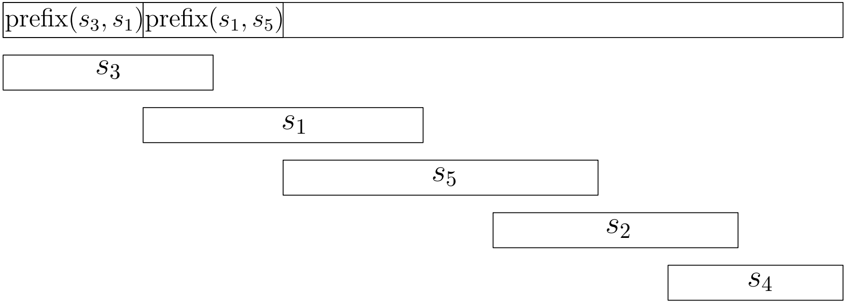


This is not very symmetric, so we additionally observe that

$$s_4 = \text{prefix}(s_4, s_3)\text{overlap}(s_4, s_3)$$

.

We will analyse an algorithm which is maybe less natural, but easier to reason about. First we need to look at *OPT* in a slightly way.

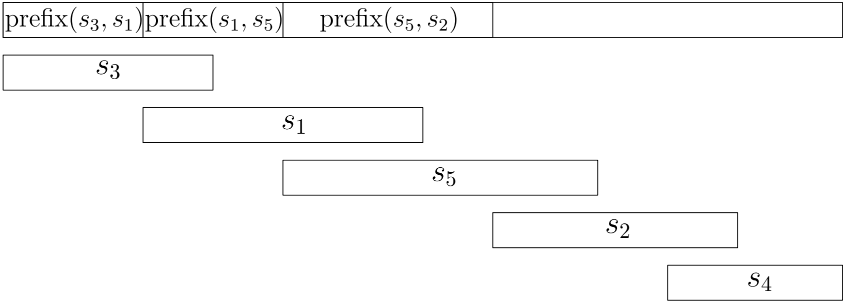


This is not very symmetric, so we additionally observe that

$$s_4 = \text{prefix}(s_4, s_3)\text{overlap}(s_4, s_3)$$

.

We will analyse an algorithm which is maybe less natural, but easier to reason about. First we need to look at *OPT* in a slightly way.

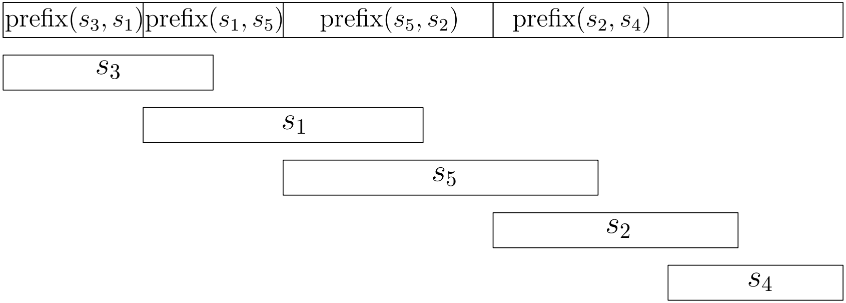


This is not very symmetric, so we additionally observe that

$$s_4 = \text{prefix}(s_4, s_3)\text{overlap}(s_4, s_3)$$

.

We will analyse an algorithm which is maybe less natural, but easier to reason about. First we need to look at *OPT* in a slightly way.

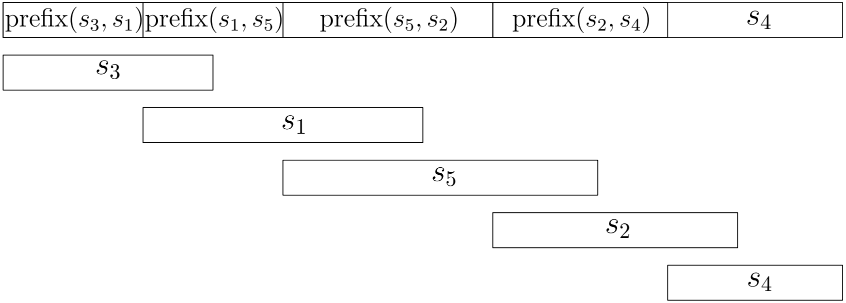


This is not very symmetric, so we additionally observe that

$$s_4 = \text{prefix}(s_4, s_3)\text{overlap}(s_4, s_3)$$

.

We will analyse an algorithm which is maybe less natural, but easier to reason about. First we need to look at *OPT* in a slightly way.



This is not very symmetric, so we additionally observe that

$$s_4 = \text{prefix}(s_4, s_3)\text{overlap}(s_4, s_3)$$

.

Lemma

The optimal solution $j_1, j_2, \ldots, j_n$ has cost

$$|\text{overlap}(s_{j_n}, s_{j_1})| + \sum_i |\text{prefix}(s_{j_i}, s_{j_{i+1}})|$$

where $j_{n+1} = j_1$.

Prefix graph

Create one node for each string s_i. Add an edge $s_i \rightarrow s_j$ with cost $|\text{prefix}(s_i, s_j)|$.

Crucial observation

The sum corresponds to the cost of a travelling salesman tour in the prefix graph. So, the cheapest such tour is a lower bound on *OPT*.

So, maybe we could construct the prefix graph, find the cheapest tour there, and then use it to construct a short superstring? It would result in a 2-approximation algorithm.

Not really: finding the cheapest travelling salesman tour is NP-hard. Nevertheless, there is something that we can find efficiently: cheapest cycle cover, which is simply a collection of cycles such that each node is on exactly one cycle.

Lemma

Cheapest cycle cover can be found in polynomial time.

See the whiteboard.

But now the problem is that we have a collection of possibly more than one cycle. How can we construct a superstring?

$w(C)$ and $\sigma(C)$

Take a cycle $C = s_{j_1} \to s_{j_2} \to \ldots \to s_{j_k}$. Construct the word:

$$w(C) = \text{prefix}(s_{j_1}, s_{j_2})\text{prefix}(s_{j_2}, s_{j_3}) \ldots \text{prefix}(s_{j_k}, s_{j_1})$$

and finally the word:

$$\sigma(C) = w(C)s_{j_1}$$

If your cycle cover is $\mathcal{C} = \{C_1, C_2, \ldots, C_\ell\}$, concatenate all $\sigma(C_i)$ to get a superstring.

Let r_i be the first string on the cycle C_i. Then the total cost is the sum of all $|\sigma(C_i)|$, which is $\sum_i |w(C_i)|$ plus $\sum_i |r_i|$. The first part is really the cost of the cycle cover, so at most OPT. We only have bound $\sum_i |r_i|$.

For this we look at OPT again. All r_i occur in the shortest superstring, so:

$$OPT \geq |r_1| - \text{overlap}(r_1, r_2) + |r_2| - \text{overlap}(r_2, r_3) + |r_3| + \ldots + |r_\ell|$$

Why? See the whiteboard.

So, now we have that:

$$\sum_i |r_i| \le OPT + \sum_{i>1} \text{overlap}(r_{i-1}, r_i)$$

hence to upperbound $\sum_i |r_i|$ we only have to upperbound $\sum_{i>1} \text{overlap}(r_{i-1}, r_i)$!

Lemma

Let C and C' be two cycles in the cheapest cycle cover, and r, r' their first strings. Then:

$$\text{overlap}(r, r') < |w(C)| + |w(C')|$$

Now using the above inequality:

$$\sum_i |r_i| \le OPT + \sum_{i>1} |w(C_{i-1})| + |w(C_i)| \le OPT + 2\sum_i |w(C_i)| \le 3OPT$$

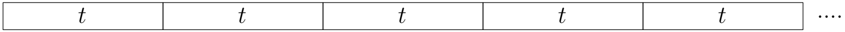
Lemma

Let C and C' be two cycles in the cheapest cycle cover, and r, r' their first strings. Then:

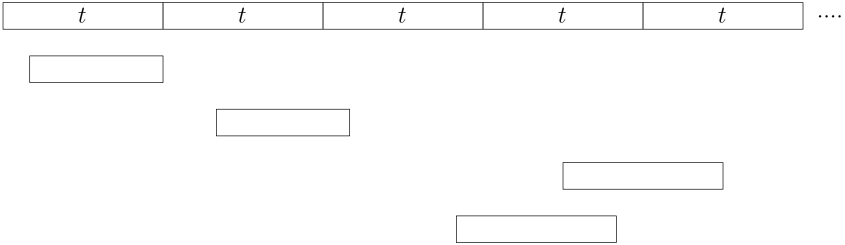
$$\text{overlap}(r, r') < |w(C)| + |w(C')|$$

Assume that $\text{overlap}(r, r') \geq |w(C)| + |w(C')|$. r begins $w^\infty(C)$ and r' begins $w^\infty(C')$, so we can choose a rotation α of $w(C)$ and a rotation α' of $w(C')$ such that $\text{overlap}(r, r')$ is a prefix of α^∞ and α'^∞ at the same time. But then from the periodicity lemma α and α' are actually both powers of the same word t, where $|t| \leq |w(C)|, |w(C')|$.
It follows that all strings on C and C' are substrings of t^∞, as they were substrings of $w^\infty(C)$ or $w^\infty(C')$. Then we claim that we can actually construct one cycle of cost at most $|t|$ containing all of them, which would decrease the total cost by at least $|w(C')| > 0$.

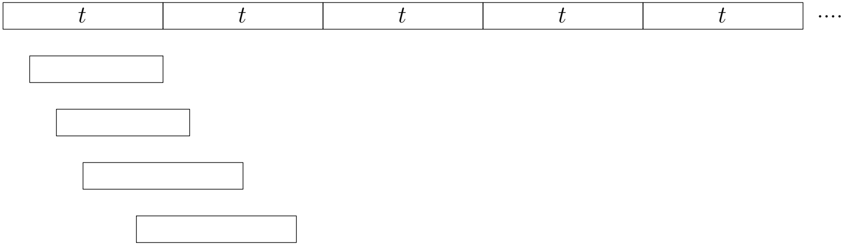
So, the only remaining part is to show that if we have a collection of strings which all occur in t^{∞}, there is a cycle of cost $|t|$ in the prefix graph containing all of them.



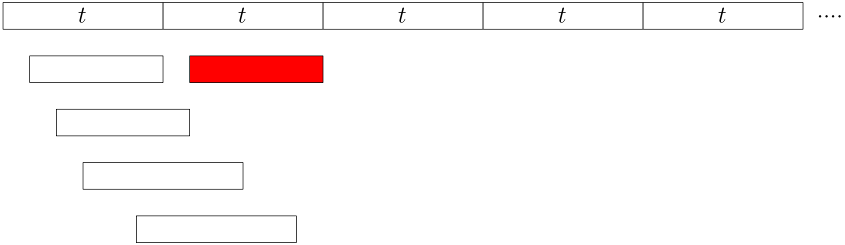
So, the only remaining part is to show that if we have a collection of strings which all occur in t^∞, there is a cycle of cost $|t|$ in the prefix graph containing all of them.



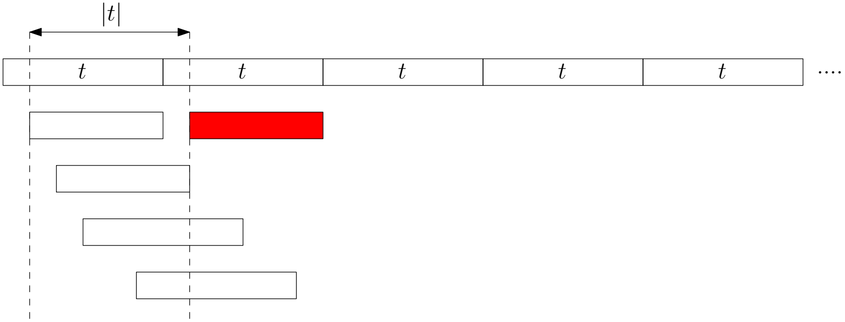
So, the only remaining part is to show that if we have a collection of strings which all occur in t^∞, there is a cycle of cost $|t|$ in the prefix graph containing all of them.



So, the only remaining part is to show that if we have a collection of strings which all occur in t^∞, there is a cycle of cost $|t|$ in the prefix graph containing all of them.



So, the only remaining part is to show that if we have a collection of strings which all occur in t^∞, there is a cycle of cost $|t|$ in the prefix graph containing all of them.



So, we get 4-approximation. Can we do better?

Yes!

3-approximation is possible.

Sweedyk 1999

2.5-approximation is possible.

...and the story is not over yet!

Mucha 2013

$2\frac{11}{23}$-approximation is possible.

The road so far...