Problem Sirrom - solution

Proposed by Pătcaş Csaba

1 Overview

The problem is known in the literature as the Shortest Common Superstring (SCS), which is currently known to be NP-complete. But considering the soft limits for n we can give a solution with certain optimizations, which works in the time limit.

The solution is composed of two parts. First, we must determine the longest suffix-prefix for each pair of words. Then we must find the longest Hamiltonian path in the corresponding graph. Each of these two parts can be done using trivial and efficient methods.

2 Finding the longest suffix-prefix

This part can be done trivially in $O(l1 \cdot l2)$, where $l1$ is the length of the first word, and $l2$ is the length of the second word.

The best method is based on the partial match table calculation part of the Knuth-Morris-Pratt algorithm (which also inspired the problem title :)). We must calculate the table for the concatenation string of the two words (in inverse order) and return the last element of the table. Using this approach the desired number can be calculated in $O(l1 + l2)$ time and space for a pair of words, yielding to a total complexity of $O(n^2 \cdot l)$, where l is the double of the average length of the words. In the worst case this means $\frac{15 \cdot 14}{2} \cdot 20000 = 2.100.000$ basic operations, which is absolutely fast enough for the first phase.

Because a word isn't considered a prefix or suffix of itself, there are a number of special cases, that needs to be treated:

- If one of the two words has a single character, the return value is surely 0.

- To solve cases where the first word is a prefix of the second, such as "xxx" and "xxxaaa" we will start the calculation of the table for the string "xxxaaaxxx" from the second character of the first word, which is the 8. character of the concatenated string.

- To solve cases where the second word is a suffix of the first, such as "aaaxxx" and "xxx" we will simply overwrite the the last character of the second word with a dummy character (#), so we will get "xx#aaaxxx" for the concatenated string in this case.

3 Finding the longest Hamiltonian path

Of course, this is the part where the NP complete property comes from. The brute force method uses the backtracking technique to generate all the possible permutations of n elements and calculates the length of the corresponding path for each of them. Thus, this method yields to a complexity of $O(n!)$.

The backtracking can be highly optimized as follows. Store in a vector all the values calculated in the first phase and sort it in decreasing order than let $overlaps_i = overlaps_i + overlaps_{i-1}$ for each i. Why does this help us? If we are at a certain level in the backtracking, we know the length of the path constructed so far and the number of edges that needs to be added until a complete Hamiltonian path. The i^{th} element of the *overlaps* vector will contain the maximal sum we can get from i edges. Thus, adding it to the current path length, we get the maximal length of the final path that can be achieved in optimal circumstances. Starting from this remark, we can optimize the backtracking by stepping to the next level, only if there is a chance to find a better solution, than the best found so far.

Another possibility to solve this part is to generate a big number of random permutations and choose the one, which gives the longest Hamiltonian path. This method can be also optimized using the observations from the last paragraph.

The proposed solution uses the matrix C which is defined as follows:

$$C_{i,j} = \texttt{the length of the longest path that ends in node } i \texttt{ and contains the nodes represented by the bits of } j$$

This matrix can be easily built using the dynamic programming or memoization methods and the following relations:

$$C_{i,j} = \begin{cases} 0 & j = 2^i \text{ or } j = 0 \\ \max C_{k,j-2^i} + A_{k,i}, \forall k : \text{the } k^{th} \text{ bit of } j \text{ is set} & \text{otherwise} \end{cases}$$

We noted with A is the adjacency matrix of the graph. The final solution will be given by $\max_{i=1}^{n} C_{i,2^n-1}$.

4 Test data

In order to facilitate the construction of a powerful test data set, the implementation of the suboptimal versions described above was needed.

The *sirrom.cpp* and *sirrom_c.cpp* programs solve the problem in the optimal way, the first one using STL types and function, while the latter using mainly standard C types and functions (this version being faster, as seen on the table below). The *sirrom_slow1.cpp* and *sirrom_slow2.cpp* programs use the trivial suffix-prefix construction algorithm, the only difference being the order in which they check whether a possible optimal solution is a match. The implementation of the optimized backtracking can be found in the *sirrom_back.cpp* file, while the probabilistic method is implemented in *sirrom_rnd.cpp*.

The testing was done on a machine with AMD Barton 2500 processor, 768 Megabytes RAM memory and Windows XP Pro operation system. All programs were run in Visual Studio 2005 using Release mode.

Case	n	Solution	Remark
1	6	17	Hand-crafted test with some special cases
2	14	16	All possible configurations with length ≤ 3
3	15	149931	Randomly generated binary strings with length 10000 and probability 0.33 for letter "a"
4	15	149908	Randomly generated binary strings with length 10000 and probability 0.25 for letter "a"
5	15	10014	Strings of 10000 "a" letters
6	15	149986	Strings of 9999 equal letters followed by a single different character
7	15	80014	4999 "a" followed by one "b" then 5000 "a"
8	15	150000	9999 "a" and one "b"
9	15	150000	One "b" then 9999 "a"
10	15	149812	Randomly generated binary strings with length 10000 and probability 0.1 for letter "a"
11	15	15	Strings of one character
12	0	0	Empty dictionary
13	1	18	A single word
14	1	1	A single character
15	15	71	alpha beta charlie ...

Running times

Remark: The probabilistic method generated 500.000 permutations for each case. The suboptimal solutions found are given in the parenthesis.

Case	sirrom	sirrom_c	sirrom_slow1	sirrom_slow2	sirrom_back	sirrom_rnd
1	0.000s	0.000s	0.265s	0.172s	0.000s	0.828s
2	0.078s	0.063s	0.281s	0.266s	4.063s	2.234s (17)
3	0.609s	0.469s	0.516s	0.500s	TLE (149934)	2.641s (149940)
4	0.625s	0.454s	0.500s	0.532s	TLE (149922)	2.516s (149923)
5	0.579s	0.438s	0.469s	0.516s	0.141s	1.641s
6	0.641s	0.438s	0.469s	0.485s	0.141s	1.703s
7	0.594s	0.438s	8.187s	8.156s	0.156s	1.625s
8	0.656s	0.375s	TLE	0.500s	0.156s	1.641s
9	0.531s	0.438s	0.484s	TLE	0.125s	1.609
10	0.610s	0.454s	0.547s	0.532s	TLE (149827)	2.484s (149835)
11	0.235s	0.282s	0.453s	0.484s	0.000s	1.547s
12	0.000s	0.000s	0.204s	0.204s	0.000s	0.016s
13	0.000s	0.000s	0.188s	0.172s	0.000s	0.047s
14	0.000s	0.000s	0.188s	0.188s	0.000s	0.047s
15	0.219s	0.266s	0.453s	0.469s	TLE (72)	1.438s