Advanced Algorithms: Project

(deadline: May 13th, 2016, 17:00)

Alexandre Francisco and Luís Russo

Last modified: February 26, 2016

This project considers two different problems described in part I and part II, respectively. Both problems contribute equally to the final grade. The delivery deadline for the project code is May 13th, at 17:00. The delivery deadline for the project report is May 15th, at 17:00.

Students should not use existing code for the algorithms described in the project, either from software libraries or other electronic sources.

It is important to read the full description of the project before starting to design and implement solutions.

Students should deliver working implementations, along with a report including the experimental setup, time and space analysis, both theoretical and experimental, of the algorithms implemented and proposed.

1 Part I: Search trees

The first part of this project concerns the implementation of the operations described in the `tsearch` man page for managing binary search trees [3].

The challenge is to implement such operations relying on a treap data structure. Random treaps will be discussed in the classes and are described in detail in section 8.2 of Motwani and Raghavan's book [2].

We recommend to start by implementing those operations relying on a simple binary search tree.

Your report should include a comparison of your different implementations and also a comparison with the implementation of these operations found on glibc. You should also analyse your implementation according to known theoretical results on treaps, discussing experimental issues observed in your experiments.

1.1 Specification

To automatically validate the implementation we use the following conventions. Your implementation should generate a binary that is executed with the following command:

```
./project < in > out
```

The file `in` contains the input commands that we will describe next. The output is stored in a file named `out`. The input and output must respect the specification below precisely. The output file will be validated against an expected result, stored in a file named `check`, with the following command:

```
diff out check
```

This command should produce no output, thus indicating that both files are identical.

The format of the input file is the following. The first line contains a single integer, k, that indicates the number of lines that follow. Each of the following k lines contains a char in $\{A, F, D\}$ and a string, separated by a white space '` `'. The input contains no more data.

The symbols A, F and D denote the operations add string, find string and delete string, respectively. Your program should keep the number of occurrences of each string by using your implementation of above operations for managing binary search trees.

The output should consist of k lines containing the output of each operation, which will be the number of occurrences for the given string, or `NULL` if it does not exist.

1.2 Sample Behaviour

The following example shows the expected `output` for the given `input`. These files are available on the course web page.

input

```
11
A ab234f
A dc3dfa
A ab234f
F ab234f
A dc3dfa
A dc3dfa
A edf123
F dc3dfa
D dc3dfa
F dc3dfa
F edf123
```

output

```
1
1
2
2
2
3
1
3
3
NULL
1
```

2 Part II: Longest substrings

In second part of this project the challenge is to implement the suffix tree algorithm to compute the size of the longest substrings that are common to at least `d` distinct strings. An algorithm for this problem is explained in section 7.6 of Gusfield's book [1].

The algorithm runs in two phases. First it builds a generalized suffix tree, for the set of strings. This is achieved in linear time using Ukkonen's algorithm. In the second step the tree is traversed, possibly with a DFS, and information about the set of distinct strings that contain a certain substring is computed, by performing unions of sets of string identifiers whenever such a set contains at least `d` distinct strings and its string-depth is compared to the longest known value. In fact a slight improvement can be done in this step, because we are interested in the complete table of strings sizes for all the possible values of `d`.

The input will consist of a set of DNA sequences, hence underlying alphabet will be A,C,T,G.

In this project we do not require a naive implementation, but students are advised to implement one for debugging purposes.

2.1 Description

The input will consist of a set $\mathcal{S} = S_1, S_2, \ldots, S_k$, with k strings. The total size of the strings is m. The most naive solution for this problem compares every substring of S_j with every suffix of S_i until there is a mismatch, or one of the strings ends. This algorithm would require $O(m^4)$ time, although this is a rough upper bound. Still it is a good algorithm for debugging.

A more efficient version could use the Boyer-Moore pre-processing, or adapt the Knuth-Morris-Pratt algorithm and obtain $O(m^2)$ time. Using suffix trees we can obtain the better bound of $O(km)$.

Hence the first step must be constructing the generalized suffix tree. To obtain the previous bound this must be done efficiently, in linear time, by using Ukkonen's algorithm. This algorithm is fairly challenging to implement and constitutes a major part of this assignment. Particularly because generalized suffix trees require some extra issues. In implementing this data structure students are advised to use the "sentinel" programming technique. This technique consists in augmenting the data structures to avoid writing more code. The idea is that to avoid producing code that contains many conditional selections, `if` statements mainly, we add extra content in the data structure. An important example for suffix trees is the suffix link of the `root` node. The general definition does not apply to this case, therefore it is formally undefined. For programming purposes it is best to add a `sentinel`, that is the suffix link of the `root`. It should also be possible to `Descend` from this node to the `root` with any letter.

A fair amount of information must be stored at each node. A recommended structure, in C, is the following:

```
typedef struct node* node;

struct node
{
  int Ti;                          /**< The value of i in Ti */
  int head;                        /**< The path-label start
                                      at &(Ti[head]) */
  int sdep;                        /**< String-Depth */
  node child;                      /**< Child */
  node brother;                    /**< brother */
  node slink;                      /**< Suffix link */
  node* hook;                      /**< What keeps this linked? */
};
```

Most fields are explained by the comments. This structure can be used both to represent internal nodes and to represent leaves. For leaves a smaller structure could be used, thus saving overall space, but such optimization is not critical and therefore not recommended. The trickiest field in the previous structure is the `hook`, which is a pointer to a pointer to a `struct`. If you find this confusing you can instead use a doubly linked list for the `brother`, i.e., store two pointers instead of one, and a `father` pointer that points to the node that is the father of this one. This makes inserting nodes into the tree simpler to implement.

The `hook` version requires less code, and space. A node `struct` is pointed to by only one `child` or `brother` pointer, the idea of the `hook` is to keep a pointer to that pointer. With this information it is also simple to update the structure for removals, and moreover the code does not depend on whether the reference was from a `child` pointer or a `brother` pointer. Still the `hook` information must be properly updated, for example if `p->b` becomes the `child` of node `x` we would use the following code :

```
    x->child = p->b;
    p->b->hook = &(x->child);
```

A simple outline of Ukkonen's algorithm is the following, slightly different from section 6.1:

```
j = 0;
while(j <= ni[i])
  {
    while(!DescendQ(p, Ti[i][j]))
      {
        AddLeaf(p, i, j);
        SuffixLink(p);
      }
    Descend(p, Ti[i][j]);
    j++;
  }
```

This code is used to insert text `Ti[i]` into the generalized suffix tree, therefore `Ti[i][j]` represents the character at position `j`, the size of this text is stored in `ni[i]`. For every character `Ti[i][j]` that need to be inserted we create nodes and follow suffix links until we find a position where it is possible to descend by the letter `Ti[i][j]`. The variable `p` represents a point, i.e., a position between two letters of a given branch. The following structure is a possible implementation of this concept:

```
typedef struct point* point;

struct point
{
  node a;                       /**< node above */
  node b;                       /**< node bellow */
  int s;                        /**< String-Depth */
};
```

The `DescendQ` function returns true when it is possible to descend from `p` with `Ti[i][j]`. Then `DescendQ` function actually descends with the given letter. The `AddLeaf` function creates a new leaf and inserts it into the tree, in this process it might also be necessary to create a new internal node. The `SuffixLink` alters the point `p`, so that it points to the suffix link of the current point. This function must implement the skip/count trick of section 6.1.3. An important detail to be mindful about is that the `AddLeaf` function should not alter `p`, the reason for this is that the `SuffixLink` uses the value in the `slink` field, and for recently created nodes this value is not yet available. In fact assigning this value, for a new node, is another crucial detail, it should be set by `SuffixLink` if `p` is an internal node, otherwise it should be set to the next new internal node, created on the next call of `AddLeaf`, this issue is referred in Lemma 6.1.1.

There is also an important trick to implement generalized suffix trees, again focusing on reducing the amount of conditional code. In a generalized suffix tree all terminators should be different symbols. If we where to use real characters this would limit k to $255 - 4$, which is too small, particularly for the larger tests. Moreover handling which characters are terminators, avoiding the DNA letters could be messy. Alternatively we could use `‘\0’` as a terminator and add code so that the comparison of two terminators depended on `Ti` field of the node `struct`. There is a simpler and cleaner approach, we can use a `‘\0’` for all terminators as before, except for the T_i that is currently being inserted into the tree, for that string we use `‘\1’`. Once the string gets inserted into the tree we switch the terminator to `‘\0’` and proceed with the next text. In essence the previous code we showed can be inserted the middle of the following code:

```
i = 0;
while(i < k)
  {
    Ti[i][ni[i]] = ‘\1’;      /**< Force diferent terminators*/

    ...

    Ti[i][ni[i]] = ‘\0’;
    i++;
  }
```

After building the generalized suffix tree the second step is easier. At each node we store a list representing the set of values of `i` for which some suffix `T[i]` corresponds to a descendent of the node. These lists can be computed in the finishing time of a DFS visit to the node, by merging the lists of the children of a node. To merge these lists you can use an array, of size k, that indicates for index `i` if a suffix of `T[i]` is already know to be in the currently merged list. This allows us to avoid duplicating these indexes, but reseting the modified entries back to `false` must be done carefully after each use.

Once we know the size of each list it is possible to search for the node with the largest `sdep` value for which the corresponding list contains at least `d` elements. Note that we are interested in all the k possible values of `d`. By using an array, of size k, it is possible to traverse the suffix tree only once, instead of k times. Thus reducing the time, of this particular operation, from $O(kn)$ to $O(k+n)$.

Finally considering memory allocation it is best to make few calls to `malloc` and allocate big chunks of memory. In a naive implementations these functions can easily occupy 25% of the overall time. All the nodes of the suffix tree can be allocated, after reading the texts, i.e., knowing m, since there can be no more than $2m+1$ nodes. This chunk can be reduced after finishing the tree, but it is not necessary for the memory limits in the mooshak system. The elements of the stacks for the DFS can be implemented with linked lists, likewise it is possible to allocate all in one step. Keeping track of which elements are in use and which are not is straight forward.

2.2 Specification

To automatically validate the index we use the following conventions. The binary is executed with the following command:

```
./project < in > out
```

The file `in` contains the input commands that we will describe next. The output is stored in a file named `out`. The input and output must respect the specification below precisely. The output file will be validated against an expected result, stored in a file named `check`, with the following command:

```
diff out check
```

This command should produce no output, thus indicating that both files are identical.

The format of the input file is the following. The first line contains a single integer, k, i.e., the value indicates the number of strings that follow.

Each of the following k lines starts with a number, m_i. This number indicates the size of the string that follows. Then there is a white space `' '` and finally a string with m_i characters from the `'A'`, `'C'`, `'T'`, `'G'` alphabet.

The input contains no more data.

The output should consist of a single line containing the sizes of the longest substring that exists in at least `d` different strings. For all the possible values of `d` ranging from 2 to k. Every pair of values is separated by a single space `' '`. There should also be a space before the newline character.

2.3 Sample Behaviour

The following examples show the expected `output` for the given `input`. These files are available on the course web page.

input 1

```
4
8 AGAACAGA
6 ATATTG
3 TGG
10 TTGGTGGGCG
```

output 1

```
3 2 1
```

input 2

```
5
4 AACG
1 C
15 GTTCGGCCGACCGAA
4 CGTA
2 GT
```

output 2

```
2 2 1 0
```

input 3

```
5
4 GAAA
1 G
2 AC
1 C
18 CAGGTACAAGGTACCCAT
```

output 3

```
2 1 0 0
```

input 4

```
5
3 GTT
10 TGGTAATTGC
20 CTAAATCTCCCGTAAATATC
4 TTAT
9 ATAAATAAG
```

output 4

```
6 3 2 1
```

input 5

```
5
4 CCGG
18 GGCTGATATCCGGACGCC
4 GACC
7 CCCCCCA
12 CAGCCGGAAGAC
```

output 5

```
5 4 2 2
```

input 6

```
5
22 CACAGGATAGAACGGAACACAT
20 ATCAACACACTGAGCTGCAA
3 GCA
41 CACGGAGCAAACACAAGTCGATGGATCCATGCCTATACGGG
9 AACACCGGA
```

output 6

```
6 6 5 2
```

3 Grading

The final grade will result from the number of points that the students obtain in the mooshak system and the project report.

The mooshak system accepts the C programming language, click on `Help` button for the respective compiler. Projects that do not compile in the mooshak system will be graded 0. Only the code that compiles in the mooshak system will be considered, commented code, or including code in the report will not be considered for evaluation.

Submissions to the mooshak system should consist of a single file. The system identifies the language through the file extension, an extension `.c` means the C language. The compilation process should produce absolutely no errors or warnings, otherwise the file will not compile. The resulting binary should behave exactly as explained in the specification section. Be mindful that `diff` will produce output even if a single character is different, such as a space or a newline.

The report should be delivered in the fenix web page, in PDF format with at most 10 pages A4, fonts of at least 12pt and margins of at least 3cm. The report should contain a brief introduction; any relevant implementation decisions; answers to the questions in this document; extensive experimental results for the algorithms, including performance graphs for all the algorithms and comparison to the projected theoretical bounds, using several families of inputs, measuring both time and space; some brief conclusions.

Reports that are not presented in PDF, will be graded 0. Reports that are not delivered in fenix by the designated deadline will be graded 0. Notice that you can submit the report to fenix several times, you are strongly advised to submit several times and as early as possible. The same happens in the mooshak system, with the same advice. Only the last version is considered for grading purposes, all other submissions are ignored. There will be **no** deadline extensions. Submissions by email will **not** be accepted.

References

[1] Gusfield, D. *Algorithms on strings, trees, and sequences: computer science and computational biology*. Cambridge Univ Press, 1997.

[2] Motwani, R. and Raghavan, P. *Randomized algorithms*. Cambridge Univ Press, 1995.

[3] `TSEARCH(3)`. In *Linux Programmer's Manual*, 2015 (`http://man7.org/linux/man-pages/man3/tsearch.3.html`).