TÉCNICO LISBOA

DEI
Departamento de Engenharia Informática

Advanced Algorithms

2016/2017

Project

Matching and Linking

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 19th, at 17:00. The delivery deadline for the project report is May 19th, 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 the solution.

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

1 Part I: Text Searching

1.1 Overview

In this project students will implement and compare several online algorithms for matching small patterns against a large texts. This problem is recurrent in computer science and specially in bioinformatic applications, therefore we will use a reference database of DNA sequences.

The input will consist of several DNA sequences. A reference sequence T and several search patterns P. Since the sequences are DNA the underlying alphabet will be A,C,T, G.

The first algorithm will simply be the naive algorithm, which tests P at every possible location. We will then implement the Knuth-Morris-Pratt and Boyer Moore.

1.2 Knuth-Morris-Pratt

Since the naive algorithm may require $O(nm)$ time, where n is the size of T and m is the size of P, it is important to find more efficient algorithms. You should find examples where this bad performance occurs, these examples will be useful for the experimental validation.

For this purpose we will implement the Knuth-Morris-Pratt algorithm. This algorithm starts by computing the π table which for every i store the size of the longest suffix of $P[..i]$ that is also a proper prefix. Note that this processing step must be computed in $O(m)$, but the technique is similar to the general algorithm. In general the KMP algorithm keeps track of the size of the longest prefix of P that matches a suffix of $T[..i]$. At each step the value of i is incremented, in which case the size of the prefix of P can increase by 1, or be decreased using the π table. This is determined by comparing $T[i]$, with a given letters, or letters of P. Both reference books [1, 2] contain detailed explanations of this algorithm.

For this algorithm we want to output the locations of all the occurrences of P in T. Moreover we, also, want to count how many times a letter of P is compared with a letter of T, note that you should not count comparisons when pre-processing P. We want to count both the comparisons that extend the prefix of P and those that lead to accesses to the table π. What is the relation between this value and n? Validate this relation experimentally.

1.3 Boyer-Moore

The Boyer-Moore is another “efficient” algorithm for exact string matching. This algorithm is in fact a combination of heuristics, a proper combination of which yields an $O(m + n)$ time algorithm. These heuristics can be adapted for the kind of text in question. The proof of this bound is intricate and beyond the scope of this course.

The algorithm matches P against T and after a match of a fail P is shifted to the right, P starts in the beginning of T and finishes at the end. However contrary to the naive algorithm or KMP the letters in P are compared from right to left, i.e., the first letter to compare is the last letter of P. A match is found when all the letters of P, up to $P[0]$, match the corresponding letters of T.

The algorithm obtains its good performance by using “large” shifts. There are two heuristics for shifting, the bad character rule and the good suffix rule.

The bad character rule, uses a table L, which for every letter c of Σ stores the position of the rightmost occurrence of c in P. Whenever a letter c of T fails to match a letter of P we use the table L to shift P so that the rightmost c of P is now aligned with the c in T. Notice that this might mean that the pattern was shifted backwards, i.e., to the left, in that we case we instead shift P one position to the right. This rule can be improved to the extended bad character rule that stores all the positions of every letter in P, this rule fixes the previous problem in a more efficient way, using essentially the same time and space requirements. For the purposes of this project we will implement only the “simple” rule.

Another rule the algorithm also uses is the (strong) good suffix rule, it is similar in spirit both to KMP and to the previous rule. For every suffix $P[i..]$, of size ℓ, we store the location k of the right most occurrence of the same string in P, i.e., $P[i..] = P[k..k + \ell - 1]$ and $P[i..] \neq P[k'..k' + \ell - 1]$ for any $k' > k$. If this was the only restriction then it would be a weak good suffix rule. To be a strong good suffix rule we further require that k is the beginning of P, i.e., $k = 0$, or that $P[k - 1] \neq P[i - 1]$. This means that in the strong rule k is the rightmost occurrence of the suffix $P[i..]$ that cannot be extended to the left. Hence the restriction on k' must also be updated.

Whenever P need to be shifted the Boyer-Moore algorithms computes both these shifts and chooses the largest one. For further details on this algorithm see Gusfield's book [2]. Notice that pre-processing of the bad character rule is quite straight forward, but the strong good suffix is more intricate, in fact it requires the Z algorithm.

With these two rules it is possible to guarantee that the Boyer-Moore algorithm runs in $O(m + n)$ time. Provided that P does not occur in T. Present an example where this algorithm requires $O(mn)$ time. To fix this bad performance it is possible use the Galil rule, which avoids having to math the pattern all the way to the end. We will not implement this rule, you should show experimental proving that this bad performance occurs in practice.

Like the KMP algorithm the output should contain the locations of all the occurrences of P in T. Moreover we, also, want to count how many times a letter of P is compared with a letter of T, note that you should not count comparisons when pre-processing P. What is the relation between this value and n, when P does not occur in T? Validate this relation experimentally. Why is the BM algorithm claimed to be sub-linear?

1.4 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 bellow 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.

Each operation is issued in a separate line, it begins by a letter that identifies it and is followed by a sequence of argument or options.

T followed by single space `' '` followed by the string T. This strings consists of characters `A`, `C`, `G`, `T` and ends with a, single, newline character `'\n'`. This commands specified the text that will be used for the sub-sequent search. Note that more than one `T` command may be issued, in which case the previous text is replaced by the new one. Only one text is considered. Note that the size of the text is not specified, and it might be large. Therefore you should use appropriate dynamic memory techniques to increase the necessary space.

N followed by single space `' '` followed by the string P. Upon receiving this command the program should do a naive online search for P in T.

At least one *T* must have been issued before this command. The input files and test files verify this restriction, which therefore does not need to be checked. For each occurrence this command should print its position followed by a space `' '`. After printing all the occurrence positions the command issues a newline `'\n'`.

K , same as the `N` command, but using the KMP algorithm. Moreover the output should include a second line containing the number of character comparisons, a space `' '` and a newline `'\n'`. For example for `printf` the format is `"%d \n"`.

B , same as the `B` command, but using the BM algorithm, for the occurrences and the number of comparisons.

X , terminates the program without producing output

1.5 Sample Behaviour

The following examples show the expect `output` for the given `input`. These files are available on the course webpage.

input 1

```
T TCGCAGGGCG
N TC
K TC
B TC
X
```

output 1

```
0
0
11
0
7
```

input 2

```
T AAAAAAAAAA
N AAA
K AAA
B AAA
X
```

output 2

```
0 1 2 3 4 5 6 7
0 1 2 3 4 5 6 7
17
0 1 2 3 4 5 6 7
24
```

input 3

```
T AGGTACCCAT
K CA
X
```

output 3

```
7
13
```

input 4

```
T AAAAAAAAAA
K AAA
X
```

output 4

```
0 1 2 3 4 5 6 7
17
```

input 5

```
T GCCCAAAGAC
B CA
X
```

output 5

```
3
9
```

input 6

```
T AAAAAAAAAA
B AAA
X
```

output 6

```
0 1 2 3 4 5 6 7
24
```

2 Part II: Connectivity in Forests

2.1 Overview

In this problem we aim to maintain the connectivity information of an underlying forest. The forest consists of a set of vertexes linked by edges, such that no set of edges forms a cycle. The objective is to support the following operations:

- `Link(u,v)`, that adds an edge linking the node `u` to the node `v`. If such an edge already exists or this insertion would create a cycle then the operation has no effect.
- `Cut(u,v)`, that removes the edge linking the node `u` to the node `v`, if such an edge exists. If the edge does not exist this operation has no effect.
- `ConnectedQ(u,v)`, which returns true if there is a connection from `u` to `v`. If such a connection does not exist it returns false. A connection may consist of a single edge, or a sequence of edges, provided that it links `u` to `v`

All theses operations should be computed in $O(\log n)$ time, where n is the number of vertexes, by using the Link-Cut tree data structure. The following structure is recommended to represent the tree.

```
struct LCT { /** Represents a node in the LCT. */
  LCT left; /** Child */
  LCT right; /** Child */
  LCT* hook; /** General Parent Pointer. */
  int sum; /** The size of this sub-tree.
  Negative values are used to indicate that
left and right pointers should be swapped
on the sub-tree. */
};
```

The `left` and `right` pointers are used to represent the pointers in the auxiliary splay trees. The `hook` pointer is used to represent the father of the node. If the node contains a parent within the auxiliary tree than it represents that parent, otherwise it represents a path parent. To distinguish between both cases it is possible to check weather the node pointed by `hook` is pointed back by the the corresponding `right` or `left` pointers. If that is not the case then it corresponds to a path parent pointer. The `hook` value is `NULL` if the corresponding path parent pointer does not exist. To avoid checking both `right` and `left` pointers the `hook` field may point directly to the corresponding pointer, hence the extra `*` in the declaration.

The `sum` field can be used to store the sub-tree size, but it is not essential for the current project. In fact this field can be used simply as a bit. When this bit is true it means that the tree is conceptually reversed. Meaning that the order of the nodes is from right to left instead of left to right. This reversed state applies to all the nodes in the tree, not only the current structure. This property is relevant for the `reRoot` operation, which changes the root of the represented tree. The following API is a recommended implementation.

- `LCT allocLCT(int V);`, allocates space for a Link-Cut tree with `V` vertexes.
- `void freeLCT(LCT t);`, frees the Link-Cut tree structure.
- `void access(LCT t, int v);`, makes the path from the root to `v` selected.
- `void reRoot(LCT t, int v);`, makes `v` the root of the represented tree.
- `int connectedQ(LCT t, int u, int v);`, returns true if there is path from `u` to `v` and false otherwise.
- `void link(LCT t, int r, int v);`, adds the edge from `r` to `v` to the represented tree. Before linking the node `r` is made the root of its represented tree.
- `void cut(LCT t, int r, int v);`, removes the edge `(r, v)` from the represented tree. The node `r` becomes the root of the resulting sub-tree.

Naturally it may be necessary to implement further auxiliary functions. The `connectedQ` function can be computed by applying `reRoot` to `u` and the computing `access` on `v`, if `u` and `v` end up in the same preferred path then the two are connected, otherwise they are not.

2.2 Specification

To automatically validate the Link-Cut tree 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 bellow 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 first line contains a single number, indicating the number of vertexes, `n`. The vertexes are numbered from 1 to n. The following lines contain operations each operation is issued in a separate line, it begins by a letter that identifies it and is followed by a sequence of argument or options.

L followed by single space `' '` followed by vertex number `u` followed by single space `' '` followed by vertex number `v`. This operation calls the link procedure that adds the edge `(u,v)` to the represented tree.

C followed by single space `' '` followed by vertex number `u` followed by single space `' '` followed by vertex number `v`. This operation calls the cut procedure that removes the edge `(u,v)` from the represented tree.

Q followed by single space `' '` followed by vertex number `u` followed by single space `' '` followed by vertex number `v`. This operation calls the `connectedQ` procedure that verifies if the vertexes `u` and `v` are connected. This procedure outputs `'T'` if the vertexes are connected and `'F'` otherwise. A newline is outputted either way.

X terminates the program without producing output

2.3 Sample Behaviour

The following examples show the expect `output` for the given `input`. These files are available on the course webpage.

input 1

```
4
L 1 2
L 2 3
Q 4 2
C 3 2
L 3 2
Q 2 3
```

output 1

```
F
T
```

input 2

```
4
L 1 2
L 2 3
L 3 4
C 4 3
L 4 3
Q 4 2
C 1 2
L 1 4
Q 4 1
```

output 2

```
T
T
```

input 3

```
4
L 1 2
L 2 3
L 3 4
C 1 2
L 1 2
Q 4 3
C 4 3
L 4 3
```

```
Q 4 2
C 1 2
L 1 4
Q 2 4
C 2 1
L 4 1
Q 2 3
```

output 3

```
T
T
T
T
```

input 4

```
4
L 1 2
L 2 3
L 3 4
C 2 1
L 2 1
Q 4 3
Q 4 2
C 2 3
L 2 3
Q 3 2
C 3 2
L 4 1
Q 2 3
C 1 2
L 1 3
Q 3 1
C 2 3
L 2 4
Q 4 1
C 1 2
L 1 2
Q 4 1
Q 3 1
C 4 3
L 4 1
Q 3 4
```

output 4

```
T
T
T
```

```
T
T
T
T
T
F
```

input 5

```
10
L 1 2
L 2 3
L 3 4
L 4 5
L 5 6
C 6 5
L 6 5
Q 8 2
Q 3 4
C 1 2
L 1 4
Q 5 3
```

output 5

```
F
T
T
```

input 6

```
20
L 1 2
L 2 3
L 3 4
L 4 5
L 5 6
L 6 7
L 7 8
L 8 9
L 9 10
L 10 11
Q 19 10
C 8 9
L 12 9
C 8 7
L 15 4
Q 18 8
Q 14 16
Q 12 1
```

```
C 4 3
L 17 3
Q 14 5
Q 10 13
C 3 4
L 3 7
Q 7 12
C 11 10
L 11 10
Q 2 11
Q 15 20
```

output 6

```
F
F
F
F
F
F
F
F
F
```

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 several programming languages, click on `Help` button for the a list of the languages and the respective compilers. 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. To determine the other extension check mooshak's `Help`. 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 webpage, in PDF format with at most 6 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 sheet; extensive experimental results for the algorithms, including performance graphs for all the algorithms and comparison to the projected theoretical bounds, using several families of strings, 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] Cormen, T. and Leiserson, C. and Rivest, R. and Stein, C. *Introduction to algorithms*. The Massachusetts Institute of Technology, 2nd Edition, 2001.

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

[3] Sleator, D.D., and R.E. Tarjan, 'Self-adjusting binary trees," Proc. 15th Ann. ACM Symp. Theory Comput. STOC-83, pp. 235–245; also in final form in *J. ACM* **32** (1985), pp. 652–686.