Advanced Theory of Computation

HomeWork1 Report

Student Name : Jeong Jae-Hwan

Registration Number : 2018 – 26716

1. Program Structure & Baker-Bird Algorithm

(a) Constraints

$\Sigma = \{ a, \dots, z\}, |\Sigma| = 26, m \le n \le 100$

Time Complexity : $O(|\Sigma|m^2 + n)$ – Time Complexity of Baker-Bird Algorithm

Space Complexity : $O(|\Sigma|m^2 + n^2)$ – We have to maintain 1D vector table for space O(n) in Algorithm step. Maintaining only 1D array can be done with interleaving the row-matching step and column matching step(These steps are described in lecture note, so I'll briefly describe it in section(b).

(b) Preprocessing

There are several preprocessing steps for implementing Baker-Bird Algorithm especially for maintaining only 1D array (vector table : $O(n)$), common preprocessing step for both which use $O(n)$ array or $O(n^2)$ array is consists of 3 steps. First, we have to make a automaton for multiple pattern matching (in AC algorithm). And this step has 3 functions named goto function g, failure function f, output function $output$. g denotes the transition which is consist of one state, one symbol, and result state. f denotes the state number that if one alphabet fails to transition in a state, then we have to move the state and check that alphabet on that state. That state is the number of $f(s)$. Last, $output$ denotes that the state has pattern that matches with given input, this output function can be completely computed through the goto and f function. The algorithm of each steps are as follows.

Before constructing goto function, initialize all goto function's value to fail (in code : -1)

Algorithm
Construction of the goto function
Input : given patterns which is divided by their rows. and we treat each row as 1 symbol
Output : g function, output function (which is not completely computed)
Algorithm
$newstate \leftarrow 0$
$for\ i \leftarrow 1\ until\ m\ do\ enter(p_i)$
$for\ all\ a\ s.t\ g(0,a) = fail\ do\ g(0,a) \leftarrow 0$

Procedure enter($a_1 a_2 \dots a_m$):
State $\leftarrow 0$; j $\leftarrow$ 1
while $g\ (state, a_j) \neq$ fail do

$\text{state} \leftarrow g(\text{state}, a_j)$
$j \leftarrow j + 1$
for $p \leftarrow j$ until m do
$\text{newstate} \leftarrow \text{newstate} + 1$
$g(\text{state}, a_p) \leftarrow \text{newstate}$
$\text{state} \leftarrow \text{newstate}$
$\text{output}(\text{state}) \leftarrow \{a_1 a_2 \dots a_m\}$

and the construction of failure function is as follows.

$Algorithm : Construction\ of\ the\ failure\ function$
$Input : goto\ function\ g, output\ function\ output$
$Output : failure\ function\ f\ and\ output\ function\ output$
$queue \leftarrow empty$
$for\ each\ a\ s.t\ g(0,a) = s \neq 0\ do$
$queue \leftarrow queue\ U\ \{s\}$
$f(s) \leftarrow 0$
$while\ queue\ \neq empty\ do$
$let\ r\ be\ the\ next\ state\ in\ queue$
$queue \leftarrow queue - \{r\}$
$for\ each\ a\ s.t\ g(r,a) = s \neq fail\ do$
$queue \leftarrow queue\ U\ \{s\}$
$state \leftarrow f(r)$
$while\ g(state,a) = fail\ do\ state\ \leftarrow f(state)$
$f(s) \leftarrow g(state,a)$
$output(s) \leftarrow output(s)\ U\ output(f(s))$

with this 2 steps, AC automaton can successfully computed. and especially for interleaving row matching steps and column matching steps, we have to maintain $O(n)$ array instead of array R which is $O(n^2)$. for each index in $(\text{row}, \text{column})$ we have to compute also KMP algorithm through column. but the thing we have to sure is that we have to be careful of in case which row matches, but column doesn't matches. in order to do that we have to define the O(n) vector table a. $a[j] = k$ denotes that the pattern's $k-1$ rows match the portion of $\text{text}[i-k+1\ ..\ i-1, j-m+1..j]$. there's a 2 cases for assigning $a[i]$'s value not only 1. (You can know that at the first, all $a[i]$ for $1 \leq i \leq n$ is 1). First one is that if the row-matching and column matching occurs, then we have to configure the a[i]'s value to new one. Not always 1. Because of possibility of exsistence of same rows. Let say $p_1 =$ "aabba" ,$p_2 =$"aaabb", $p_3 =$ "ababa", $p_4 =$ "aabba", $p_5 =$ "aabba". Then we can say one row as one symbol, then the pattern can be simplified as "12312". So think about that

1

2

3

1

2

happens, that a[i]'s value is 6, (means 5 row matches). then we have to continue the matching step, through row and column. then we have configure the value in order to assign a[i]. in this case, we can use KMP scheme[1], called prefix function(longest proper suffix, also prefix). In this case, "12" matches as longest proper suffix, also prefix. We can assign a[i] to 3(note that the definition of a[i]). So we have to preprocess the 12312's prefix function. And the algorithm details are below.

Before describing Algorithm, we must separate failure function in automaton, and this prefix function. So I coded prefix function array as f, failure function as failure_func.

$Input : fix\ a\ row\ pattern\ as\ some\ symbol, and\ then\ make\ a\ pattern\ with\ each\ row\ symbols$
$Output : f\ \ function(longest\ \ suffix, also\ \ prefix)$
$Algorithm$
$f(1) \leftarrow 0$
$k \leftarrow 0$
$for\ \ q\ \ \leftarrow 2\ \ to\ \ m\ \ do$
$while\ \ k > 0\ \ and\ \ P[k+1] \neq P[q]\ \ do$
$k \leftarrow f(k)$
$if\ \ P[k+1] = P[q]\ \ then\ \ k \leftarrow k+1$
$return\ \ f$

Note that the computation time of f function takes $O(m^2)$ because we are dealing with 2D patterns.

There's another cases for considering a[i], when row-matching step occurs but it's not matched with column-wise. See the example

1

2

3

1 (X) -> wrong occurrence in column-wise matching

In this case, we have to set a[i] with some value, these means that

|----------********---------|

|---------x.............--------x|

We have to assign a[i] value as next[i] in KMP scheme[1]. Which means maximum prefix match length of s(shift length in KMP), which means P[1...s-1]=P[s-i+1...i-1] and P[i] $\neq$ P[s], and we repeat this process until match founds. If not then assign 1. Note that s means matches through 1..s-1 matches, and in index s, mismatch occurs. So we don't have to add one for a[i] (because of definition in a[i]). Algorithm of next function is described in below. (I used h for next func)

$Input : fix\ a\ row\ pattern\ as\ same\ symbol, and\ then\ make\ a\ pattern\ with\ each\ row\ symbol$
$Output : shift\ \ function\ \ h$
$Algorithm$
$h(1) \leftarrow 0$

```
t ← 0
  for p ← 2 to p do
    while p > 0 and P[p − 1] ≠ P[t] do t ← h[t]
    t ← t + 1
    if P[p] = P[t] then h[p] ← t
    else then h[p] ← t
return h
```

Note that the computation time of h function takes $O(m^2)$ because we are dealing with 2D patterns. Finally, all preprocessing steps are explained. And we have to check the Baker-Bird algorithm.

(c) Algorithm

Key note is that we have to interleave row-matching step and column matching step, and if we fails with no-matching just assign a[col] to 1. If we matches only with row-matching step, we have to assign a[col] as shift function's value. When if we matches with both row-matching step, we have to assign a[col] as f function's value+1 (look definition of a vector table)

And the Algorithm is in below.

```
Baker − Bird Algorithm
  Input: pattern matching automata(failure func, output, goto), shift function, f function
  Output: location of match occurrence.
    For row ← 1 to n do
      State ← 0
        For column ← 1 to n do
          while g(state, T[row, column]) = fail do
            state ← f(state)
            state ← g(state, T[row, column]
            if(output(state) ≠ 0) ---- pattern found in row, but not match in column
              k ← a[column]
              while (k > 0 and P[k] ≠ output(state)) do
                k ← h(k)
              a[column] ← k + 1
              if k = m ---- pattern found in row and column
                then print row − 1, column − 1
                a[column] ← f[k] + 1
            else
              then a[column] ← 1
```

Now we have to analyze the time complexity and space complexity. Let's have a look at time complexity first. In preprocessing step, We have to compute shift function, f function and pattern matching automata. First shift function takes time for input of patterns, as we treat each row as each symbol we have to compute the whole size of the pattern, so it takes $O(m^2)$ time and also shift function takes also same as $O(m^2)$. In pattern matching automata, there's a 3 algorithms, let's have a look at each one First one is constructing goto function, and I implemented as 2D arrays, one is for node size, and the other one is for alphabet size. So it should take $O(|\Sigma|m^2)$ for

initializing, and we already know that it's other computation part is linearly proportional to the sum of the length of the patterns. (which means $O(m^2)$)[2]. For computing failure function, it's also bounded of length of patterns (which means $O(m^2)$)[2]. So in preprocessing time, we consume $O(|\Sigma|m^2)$. In algorithm part, we spend $O(n)$ for each row, and there's while loop but that can be bounded as $O(n)$ like KMP scheme[1] and we compute this for each column. So it takes $O(n^2)$ time. Totally, time complexity of this algorithm is $O(|\Sigma|m^2 + n^2)$. Let's have a look at space complexity last. In preprocessing step, automata takes $O(|\Sigma|m^2)$, shift function and f function takes $O(m)$ and this is straightforward. Also in algorithm step, we use only 1D vector table $O(n)$. In this case, the total space complexity of baker-bird algorithm is $O(|\Sigma|m^2 + n)$.

(d) Program Structure

The name of source file is hw1.cpp and there's some functions for computing algorithm or preprocessing step. Below list section shows the name of function and it's functionality. Specific descriptions are in code comment section, so if you want to see more details then see the code.

Prefix() : Given patterns, compute the f function with each row as one symbol.

Next() : Given patterns, compute the next function with each row as one symbol.

ComputeNodeSize(): Before constructing goto function, we have to know the character size.

ComputeAutomata(): Constructing goto function, and partially compute output function.

ComputeFailureFunction(): Constructing failure function in automata, and compete output function.

BakerBird(): Given preprocessing steps, compute the Baker-Bird algorithms described above.

---

2. Checker Program

We know that baker-bird algorithm is correct [3], and for reducing time complexity of checker program(compare to naïve matching which takes $O(n^2m^2)$, I applied Baker-Bird algorithm as an checker program. I make the array called answer_array in 2D with size n and n and initialize all elements as 0, and if I find match in a pair of (row, column) then I assign to 1 in that position. And finally I can compare the answer_array with given input(output file for checking). Then this whole procedures are bounded in $O(|\Sigma|m^2 + n^2)$. More details are filled in comment area at code.

3. Program Development Environment

OS : Ubuntu 18.04 LTS

Kernel : Linux 4.15.0-36-generic

CPU : Intel(R) Core(TM) i7-8700 CPU@ 3.20 GHz

Programming Language : C++

Compiler Version : gcc, g++ version : 7.3.0

More easy way to compile : Just stroke “make” command, then you can run it quickly.

4. Example Result

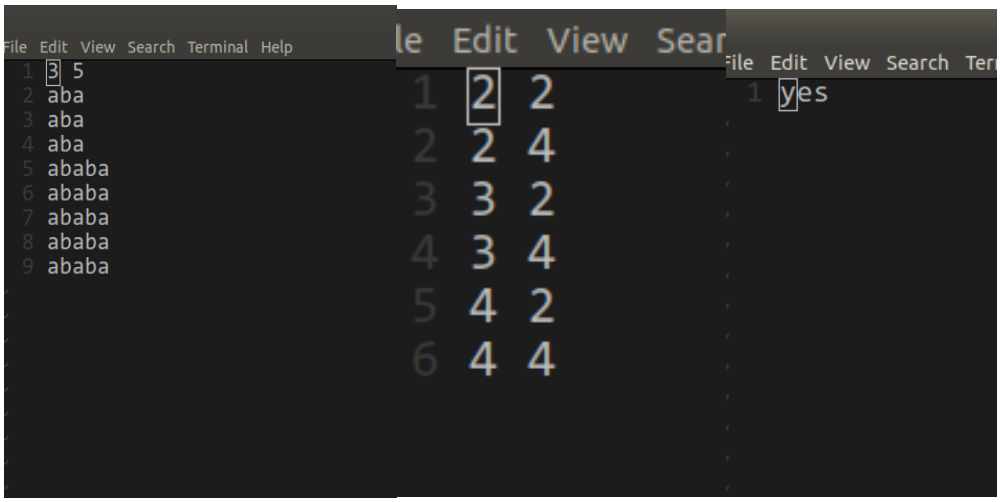
- Usage: All results was correct (in both baker-bird and checker program)

```
./hw1 bb_input.txt bb_output.txt
./checker bb_input.txt bb_output.txt cc_output.txt
```

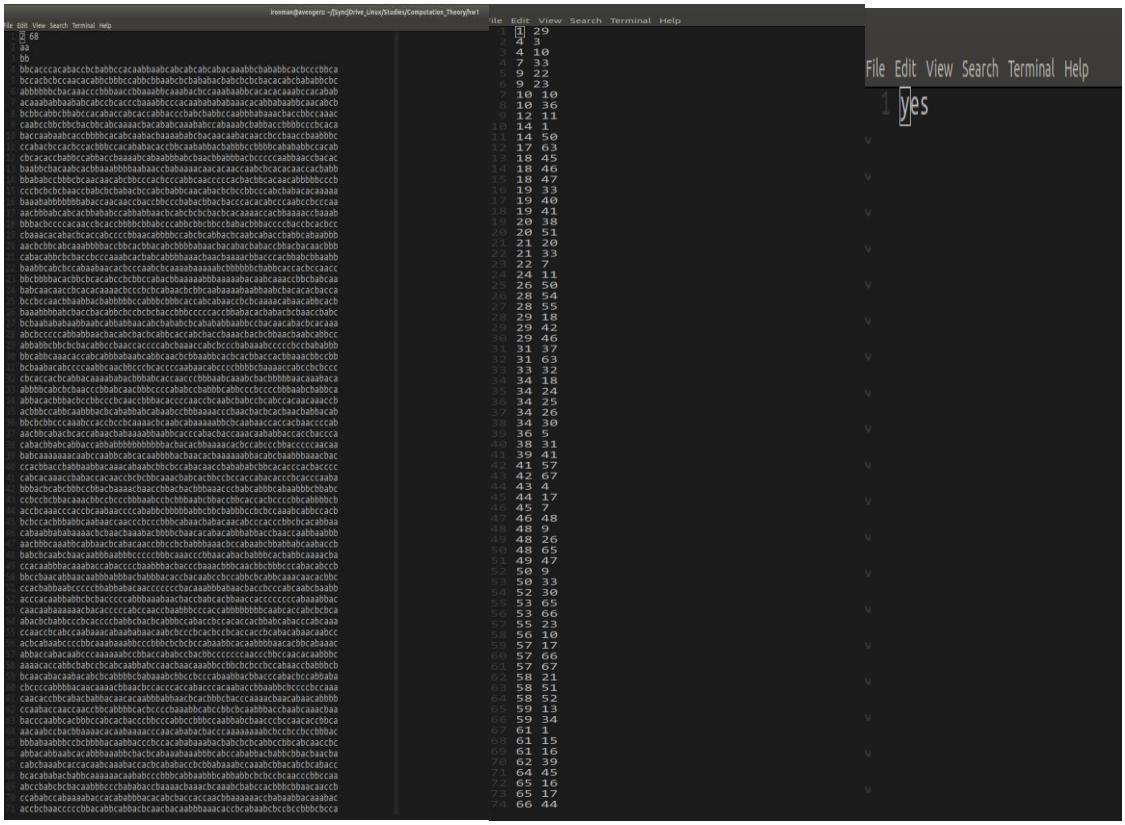
- command : ./hw1 [input] [output]

./checker [input] [output_for_check] [result_of_check]

(a) case 1



(b) case 2



5. Reference

[1] *Knuth, Donald E., James H. Morris, Jr, and Vaughan R. Pratt. "Fast pattern matching in strings." SIAM journal on computing 6.2 (1977): 323-350*

[2] *Aho, Alfred V., and Margaret J. Corasick. "Efficient string matching: an aid to bibliographic search." Communications of the ACM 18.6 (1975): 333-340*

[3] *Bird, Richard S. "Two dimensional pattern matching." Information Processing Letters 6.5 (1977): 168-170*