String Matching Algorithms

Esad Akar

String matching is the problem of searching and locating a string pattern ‘B’ in another string ‘A’. Algorithms that solve this problem usually have an input of two strings needed to be compared and an output of some sort of a representation of where pattern ‘B’ is in ‘A’. Return values might be a character pointer or a count of characters from the start of string ‘A’ the algorithm steppes over to get to the first instance of string ‘B’ in ‘A’.

Examples of where string matching is used includes text-editors, websites, file name searching, where a user needs to quickly find instances of a certain word in a large pile of words. Programs that parse text, such as compilers, regular expression engines, database look-ups, initial program argument processing are some examples of program (non-user) usage of string matching.

As with most algorithms, there exists a naïve approach of solving a problem and a more complex but efficient way. In this case, 2 different algorithms along with their theoretical and actual run times will be analyzed.

Algorithm	Worst Case O Running Time
Naïve classical approach	O(nm)
Finite-state automaton based search (1)	O(km) – preprocessing, O(n) – matching where m: |’A’|, n: |’B’|, k: |distinct characters in m|

```
Naive Find (string A, string B)                  | Let n : A.length, m: B.length
for i = 0 to A.length                            | = O(n)
  for j = i to A.length and k = 0 to B.length    | = O(mn)
    if (A[i] != B[k])
      break;
  if (k == B.length)
    return i;
return -1;                                       | O(mn)
```

```
Finite-State-Find (string A, string B)           | Let n : A.length, m: B.length, k: d
table[ASCII] = -1
distinct[B.length]
d = 0

for i = 0 to B.length                            | = O(m)
  if (table[pat[i]] == -1)
     distinct[d] = pat[i]
     table[pat[i]]

states[B.length][d]
for i = 0 to B.length                            | = O(m)
  state[i][0] = i
  for j = 0 to d                                 | = O(mk)
    if (distinct[j] == pat[i])
```

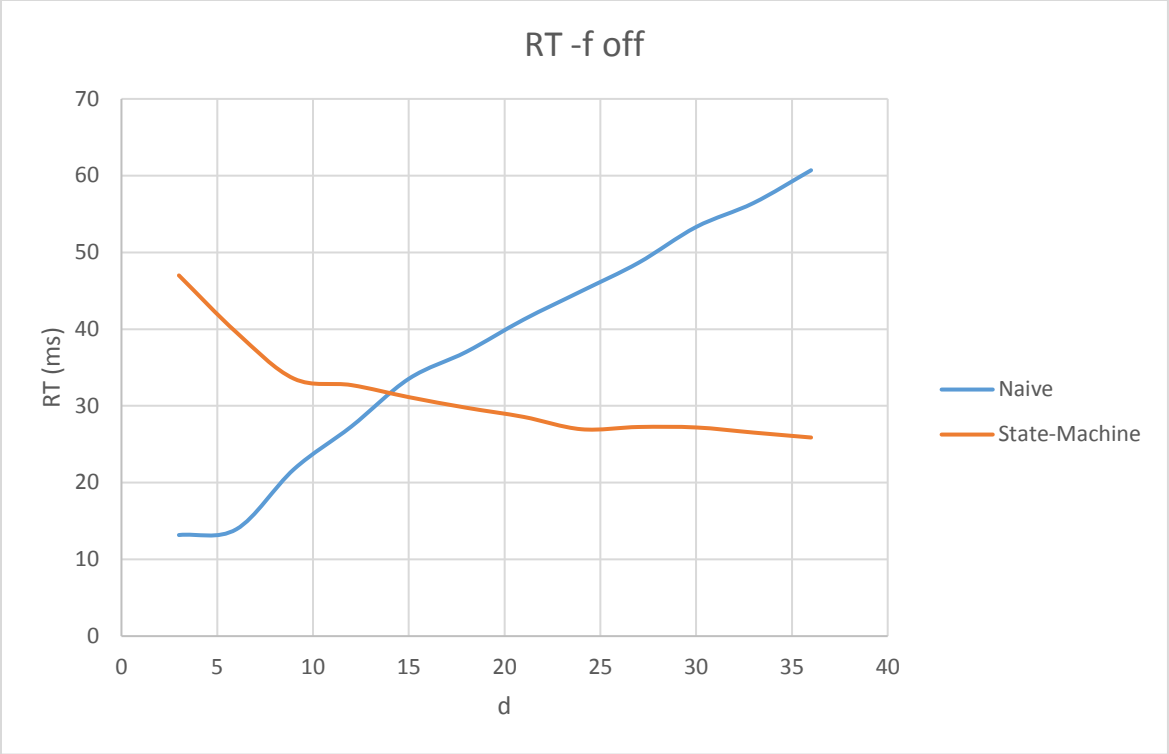
```
    state[i][j] = i + 1

state = 0
for i = 0 to A.length                          = O(n)
  state = states[state][table[str[i]]]
  if (state == B.length)
    return i - B.length
return -1                                      = O(n + mk)
```

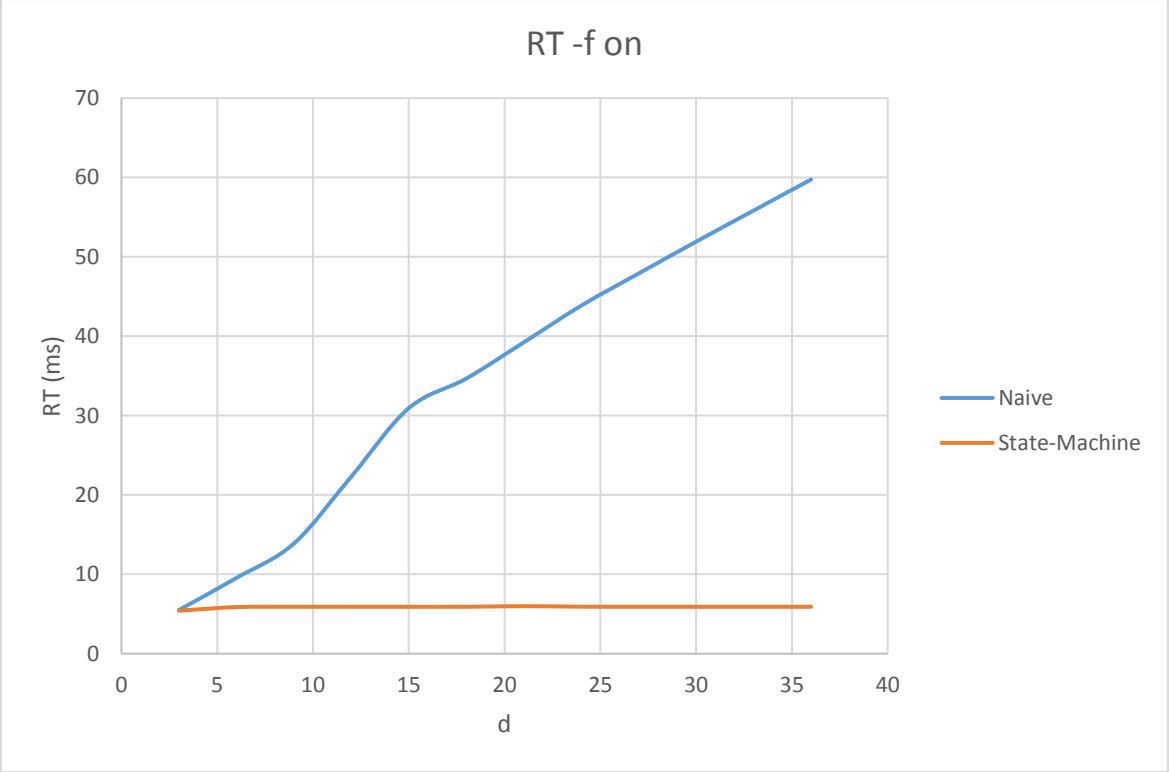
The implementation of each algorithm will stop at the first instance of string 'B' and return its index position in 'A'. They will return -1 if 'B' is not found. For each algorithm, there will be 12, timed runs and each run will contain a loop that will keep calling the algorithm starting from the last index position of 'B' in 'A' until it returns -1. In other words, all instances of 'B' will be searched for in 'A'. The total time will be divided by 12 to get an average result.

To be able to produce any tangible data to compare each algorithm, a large string 'A' is needed, and for that purpose the original plan was to use a large book and use the literature within to test the algorithms. That idea, however, quickly came to a halt as I realized string 'B' was too short in each test as it had to be a regular word in the English language. I tried using full sentences from the book but sentences rarely repeat. Another issue is, even when string 'B' is of fair size, only the few initial characters of 'B' are tested as there are very few words that start off with many of same characters and end with few different ones (ex: computed, computer). To test the worst-case scenario, here is a better idea: a create_test function is used that produces binary string of 'd - 1' repeating 0's followed by a single '1' which is then assigned to string 'B'. String 'A' is a collection of the same pattern whose sum is 1,000,000 characters. So as 'd' get larger, number of patterns decrease in 'A' as the sum of characters remains constant at 1,000,000.

To summarize the test, here is how the program works. A user in the console calls the program as such: *find -d 3. '*3' is the 'd' parameter. Create_Test creates this pattern '001' and repeats until there are a total of 1,000,000 characters in 'A' (ex: '001001001001…'). If 'd' was '4', the repeating pattern would be '0001'. Once a run is finished and timed for each algorithm, the next 'd' multiple is used, and another run is timed with the new 'd'. This repeats for 12 times. If the original 'd' value was '3', d would become '6', '9', '12'… over the iteration of the test.

Another flag that can be passed is -f (ex: *find -d 3 -f)*. This creates the same repeating pattern above, '001' in 'A', but an extra '0' is appended for string 'B' which produces '0001'. Obviously, this pattern does not exist in 'A'. This is used to time how each algorithm behaves when 'B' is not in 'A'





As stated above, |n| is constant at 1,000,000 characters. Here is table with all the data.

Naïve -f off

d	Emprical RT	Theoritcal RT	Const c
3	13.17	3E+06	4.39E-06
6	13.93	6E+06	2.32E-06
9	21.74	9E+06	2.42E-06
12	27.31	1E+07	2.28E-06
15	33.53	2E+07	2.24E-06
18	37.04	2E+07	2.06E-06
21	41.25	2E+07	1.96E-06
24	44.94	2E+07	1.87E-06
27	48.66	3E+07	1.80E-06
30	53.31	3E+07	1.78E-06
33	56.44	3E+07	1.71E-06
36	60.7	4E+07	1.69E-06

State-Machine -f off

d	Emprical RT	Theoritcal RT	Const c
3	47	1E+06	4.70E-05
6	39.58	1E+06	3.96E-05
9	33.55	1E+06	3.35E-05
12	32.71	1E+06	3.27E-05
15	31.14	1E+06	3.11E-05
18	29.76	1E+06	2.98E-05
21	28.56	1E+06	2.86E-05
24	26.96	1E+06	2.70E-05
27	27.25	1E+06	2.72E-05
30	27.19	1E+06	2.72E-05
33	26.52	1E+06	2.65E-05
36	25.88	1E+06	2.59E-05

Naïve -f on

d	Emprical RT	Theoritcal RT	Const c
3	5.5	3E+06	1.83E-06
6	9.54	6E+06	1.59E-06
9	13.88	9E+06	1.54E-06
12	22.32	1E+07	1.86E-06
15	30.94	2E+07	2.06E-06
18	34.66	2E+07	1.93E-06
21	39.22	2E+07	1.87E-06
24	43.85	2E+07	1.83E-06
27	47.9	3E+07	1.77E-06
30	51.91	3E+07	1.73E-06
33	55.83	3E+07	1.69E-06
36	59.74	4E+07	1.66E-06

State- Machine -f on

d	Emprical RT	Theoritcal RT	Const c
3	5.42	1E+06	5.42E-06
6	5.87	1E+06	5.87E-06
9	5.91	1E+06	5.91E-06
12	5.91	1E+06	5.91E-06
15	5.91	1E+06	5.91E-06
18	5.91	1E+06	5.91E-06
21	5.99	1E+06	5.99E-06
24	5.92	1E+06	5.92E-06
27	5.91	1E+06	5.91E-06
30	5.91	1E+06	5.91E-06
33	5.91	1E+06	5.91E-06
36	5.91	1E+06	5.91E-06

Let's analyze the case when -f is off. When 'd' is small such as 3, the pattern is '001' so there must be 1,000,000 / 3 repeating patters, so each algorithm must run 1,000,000 / 3 times to get though the entirety of 'A'. This is beneficial for the Naïve Find algorithm as O(nm) feels like it is O(n). The Finite-State approach struggles and performs worse than Naïve until d approaches 15. Because Finite-State has a preprocessing stage, when d is small, there are many calls to the algorithm and in each one preprocessing must be done. This isn't true for Naïve, and it can start its search immediately. When d becomes large, Naïve this time begins performing worse, as its O(nm) running time starts weighing it down.

In the case when -f is on, both algorithms return -1 in very first call. In other words, the algorithms only run once to get through 'A', whereas, in the other case, they would return at each find. Thus, Finite-State does it preprocessing once, and the algorithm is left to run its O(n) part. The run time for Naïve find, on the other hand, gets bigger and bigger as 'd' get bigger. The graphs paint a clear picture: when you need to find all instances of something, use Naïve approach, else, use Finite-State.

References:

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2014). Introduction to Algorithms. Cambridge, MA: The MIT Press. Chapter 32