Advanced Data Structures Final Project

Sergio Rodríguez Guasch

November 27, 2018

Contents

1 Introduction

Pattern matching on strings is one of the most important topics of data structures and algorithms; almost all fields of computer science use it. For example, a program that wants to find all the files that end with `.txt` must use some pattern matching software to compute this result. Also, given that the amount of available data is constantly increasing, the efficiency of these algorithms has become even more important. These algorithms are also famous for This project analyzes, implements, and compares various algorithms for string matching on multiple patterns. The objectives of this project are various: we want to explore the complexity of some of the most popular algorithms, implement them, and obtain some empirical results. Also, we want to see how these algorithms behave in some real data.

2 String Matching on Multiple Patterns

A frequent computational problem is the following: given a string T consisting of n characters, and a list of m strings $p_1, ..., p_m$, find the occurrences of all the strings $p_1, ..., p_m$ in T. In this section we will discuss some algorithms that solve this problem. For convenience, we will call the string T the text, the strings $p_1, ..., p_m$ the patterns and we will denote the length of the longest pattern as $|p|$.

2.1 Brute Force

A first approach could consist in comparing all the contiguous substrings of length $|p_i|$ of T for all the strings p_i. The asymptotic cost of this algorithm is $O(nm|p|)$ in time and makes use of $O(\#\text{matches})$ additional space if we return a vector with the indices of the matches.

2.2 Knuth Morris Pratt (KMP)

This algorithm is based on Deterministic Finite Automata (DFA). We must note that for a string s consisting of l characters, we can build a DFA M of $l+1$ states $q_0, ..., q_l$ where q_0 is the starting state and q_l is the only accepting state, an alphabet Σ, and a transition function $\delta : Q \times \Sigma \to Q$ such that M accepts any string that contains s at least once. We can build it as follows: set $\delta(q_i, s_{i+1}) = q_{i+1}$ for all the states except for the last one. For convenience, we will denote these transitions as match transitions. The rest of the transitions must be set to $\delta(q, c) = f(q, c)$, where f is a function $Q \times \Sigma \to Q$. These transitions will be named failure transitions. In order to get a correct DFA we must define f properly. It is important to note that if we start at q_0 and we follow the match transitions until the ith state we will get the ith prefix of the string s (the ith prefix of a string s is the string $s_1, ..., s_i$). We will say that the ith state represents the ith prefix, and vice-versa. If we are on the state q_i and the current symbol in the input string is c, then we must assign $f(q_i, c)$ to the state that represents the longest prefix of the string $s_1, ..., s_i, c$ that it is also a suffix of it. For example, consider the automata from figure

1 and suppose that we are on the *abab* state, and that our input symbol c is a. We can see that the longest prefix that it is also a suffix of *ababa* is *aba*, so $f(abab, a){=}aba$.
A naive construction of the aforementioned DFA can be done with the following algorithm:

```
for each a = prefix of s in increasing length:
  st = state represented by a
  for each c = character in alphabet:
    w = ac
    for each s = suffix of w in decreasing length:
      if s is a prefix of w:
        delta[st][c] = state represented by s
        break
```

We can see that this algorithm runs in $O(|s|^3|\Sigma|)$ time. Even if we can perform a single-pattern string matching in linear time once we have the automata, the cost of the preprocessing step is still too big.

The Knuth-Morris-Pratt algorithm (KMP) builds a similar data structure as the DFA explained above but in linear time. This data structure also preserves the number of states of the original DFA and its match transitions but, for a given state, the failure transitions will be the same for all the non-match symbols. More precisely, KMP defines the failure function f as follows: set $f(q_0) = q_0$, and set $f(q_i)$ to the state that represents the longest prefix that it is also a suffix of the prefix of s represented by q_i. From now on, we will say that a string that it is both a prefix and a suffix of another string s is a border of s. The most usual implementation of the KMP data structure is the following:

```
s = input string of m characters
f = table of m+1 elements
j = -1
f[0] = -1
for i in 1..m:
  while j >= 0 and s[i] != s[j]: j = f[j]
  ++j
  f[i] = j
output f
```

That is, the most common way to implement the KMP data structure is simply an array f such that $f_i =$ length of the longest border of the ith prefix of s. Given this definition, we must note that f_{f_i} will contain the length of the longest border of the f_ith prefix, which at the same time is the second longest border of the ith prefix. This property allows us to quickly search for the longest border of a prefix by trying to match it against the previously found longest borders in decreasing length (and this is in fact what the inner while loop does). If $s_i = s_j$ then it necessarily means that $s_{i-1} = s_{j-1}$, ...,$s_{i-j} = s_1$ (otherwise, j could not have reached this value because it would have been decreased on previous iterations), that is, the jth prefix of s is a border of the ith prefix of s and, in fact, it is the longest one. We must also note that j only increases one unit after each iteration, so the inner loop will perform $O(|s|)$ iterations. In conclusion, the KMP preprocessing step can be done in $O(|s|)$ time with $O(|s|)$ additional space.

The KMP algorithm can be used for single-pattern string matching by computing the transition table of the string $p\#T$, where p is the pattern, $\#$ is an out-of-alphabet character and T is the text string. We can see that all the occurrences of p in T will end at precisely on the positions on the transition table such that their value is $|p|$ (that is, the length of the longest border is exactly the length of p). Even if this algorithm works, most of the practical implementations first compute the transition table for the pattern p and then find the matches of p in T separately in order to avoid an explicit string concatenation. In any case, the total cost of the KMP algorithm on a single pattern is $O(2|p|+n)$ and $O(m(n+|p|)+m|p|)$ on a multi-pattern string matching instance. Also, it makes use of $O(m|p| + \#\text{matches})$ additional space.

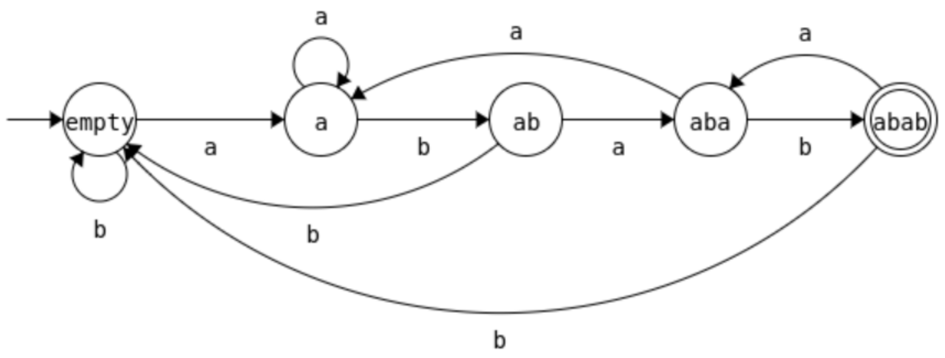

Figure 1: A DFA that accepts the strings that contain the string *abab* at least once. Horizontal arrows are match transitions and curve arrows are failure transitions. This image has been generated with the RACSO online judge (https://racso.lsi.upc.edu) visual editor for DFAs.

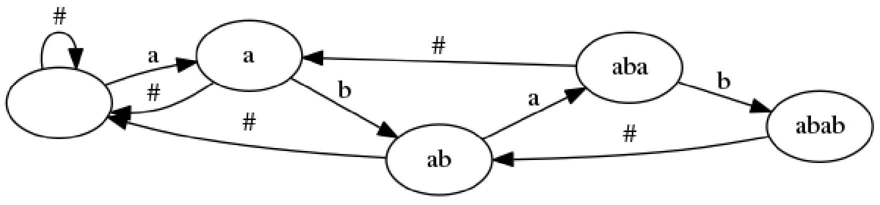

Figure 2: The representation of the transition table generated by the KMP algorithm for the word *abab*. It must be noted how the failure transitions (those labeled as #) differ from the original DFA. For example, if we are on the state *abab* and we read a *b*, then by applying the failure transitions following the KMP algorithm we will need two steps to end up at the starting state.

2.3 Trie

A trie is a rooted directed tree with labels on the edges and a boolean value on its nodes. We will say that a node u represents a string s if by concatenating the labels of the edges of the unique path from the root to u we get the string s. Also, we will say that a node u accepts a string s if u represents s and the boolean value of u is set to true. In figure 3 we can find a visual example of a trie plus some additional explanations of these concepts.

Given a list of m patterns $p_1, ..., p_m$ we can build a trie that accepts exactly them as follows:

```
insert(root, pattern, index):
  if root is empty:
    # a node is a data structure that contains
    # |sigma| pointers to node and a boolean
    # that indicates if this node accepts its
    # representative string or not
    root = new node
  if index == length(pattern):
    root.accepts = true
  else:
    insert(root.children[pattern[index]], pattern, index+1)

main:
  patterns = list of patterns
  trie = empty
  for pattern in pattern:
    insert(trie, pattern, 0)
```

We can use a trie to know if a string s has some pattern p as a prefix by simply following the path indicated by the symbols of s and recording all the states that accept its representative string.

The trie data structure can be built in $O(m|p|)$ and using $O(m|p| + \#\text{matches})$ additional space. Also, we can find all the prefixes of a string s that are also a pattern in $O(|p|)$. However, if what we want is to find all the occurrences of the patterns in some given string, then the cost goes up to $O(n|p| + m|p|)$ plus $O(m|p| + \#\text{matches})$ additional space.

2.4 Aho-Corasick

The Aho-Corasick algorithm [1] can be seen as combination of the KMP algorithm with a trie. More precisely, given the patterns $p_1, ..., p_m$ this algorithm builds a trie that accepts exactly these strings and adds some extra edges to it that can be used in exactly the same way as failure transitions are used in the KMP algorithm. From now on, if a string

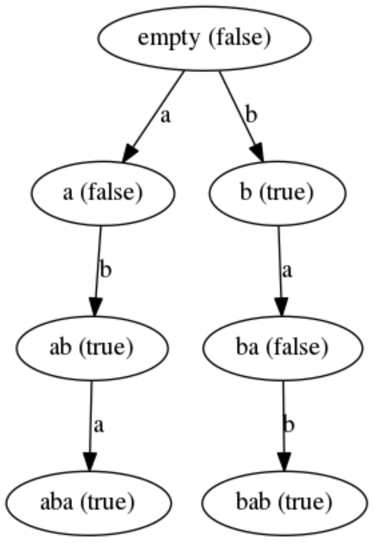

Figure 3: A visual example of a trie. The names of the nodes are precisely the unique strings that these nodes are representing. We can see that the trie accepts the strings *ab*, *aba*, *b*, and *bab*

q is a suffix of a string s and it is also a prefix of some of the strings $p_1, ..., p_m$ we will say that q is a common border of the string s and the strings $p_1, ..., p_m$. For a node u we will have that $f(u)$ leads to the node that represents the longest common border of the string represented by u and the patterns $p_1, ..., p_m$. A very important observation is that if a node u is at distance d from the initial node (or starting state) then $f(u)$ must lead to a node that is at distance less than d (except for the starting state, where $f(q_0)$ leads to q_0). This property allows us to compute the failure transitions by performing a breadth-first search on the trie. If we are on a node u then for an adjacent node v of u we will try to match the string represented by v with some of the common borders of u. As happens with KMP, this definition allows us to search for the longest common border in decreasing order with respect to a given state. Also, the length of the longest common border for a node u that is at distance d is at most one unit greater than the maximum length of the longest common border among all the nodes at distance $d - 1$. These two observations allows us to affirm that we can compute the failure transitions for all the nodes in $O(m|p|)$ (in fact, if each of the m patterns are composed by a different alphabet, the Aho-Corasick algorithm will build m KMP failure tables). As a last remark, we must note that we have defined a trie in such a way that a node accepts only its representative string. In an Aho-Corasick data structure it can happen that a node can be reached by following more than one pattern. This means that we must merge the set of accepting words of the nodes with the set of accepting words of their corresponding failure nodes.

We can use an Aho-Corasick data structure to perform a multi-pattern string matching over a string s as follows: start at the starting node and, for each character c_i, apply

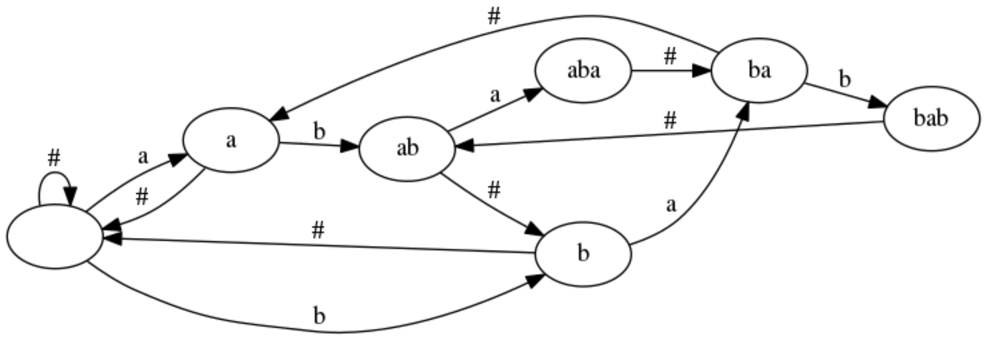

Figure 4: The Aho-Corasick data structure that results from the set of words *a*, *ab*, *aba*, and *bab*. Failure transitions are those labeled as $\#$. Even if not shown on the image, we must note that, for example, the state *aba* must accept both *a* and *aba*

the failure transitions until we reach a node that has an edge labeled as c_i, follow this edge and add the set of the accepted words by the current node to the set of found matches.

The overall cost to perform a multi-pattern string matching with the Aho-Corasick algorithm is $O(n + |p| + m|p|)$ in time and $O(|m|p + \#\text{matches})$ in space.

Pseudocode for the Aho-Corasick algorithm can be found at [1] in the figures labeled as Algorithm 2 and Algorithm 3.

2.5 Observations

As we can observe in table 1, the Aho-Corasick algorithm performs better than the other options on multi-pattern search instances and performs as KMP on single-pattern cases. Also, we can see that the behaviour of the Aho-Corasick algorithm on single-pattern cases is exactly the same as KMP. The same happens with the Brute Force algorithm and the trie.

Brute Force	KMP	Trie	Aho-Corasick
$O(nm\|p\|)$	$O(m(n + \|p\|) + m\|p\|)$	$O(n\|p\| + m\|p\|)$	$O(n + \|p\| + m\|p\|)$

Table 1: The cost of various algorithms on finding the matches of m patterns of maximum length $|p|$ on a string of n characters

We must note that this analysis does not take into account the, possibly big, hidden constants that each algorithm can have. For example, it is more than likely that the trie implementation runs significantly slower than the brute-force algorithm on single-pattern instances, even if the asymptotic analysis leads us to the same cost. Also, some of these constants can be due to factors as (lack of) cache locality or compiler ability to optimize some particular constructions. Given this situation, we conclude that some empirical experiments must be done.

3 Experiments

This section has three objectives: verify that the algorithms behave according to the theoretical analysis we have performed before, measure some extra factors as the possible hidden constants in our implementations, and test these algorithms on some real data.

3.1 Implementation

In order to be able to perform empirical experiments we have implemented all of the algorithms explained above in C++. The source code (an some additional scripts) can be found at `https://github.com/srgrr/AhoCorasick`. The required dependencies are `g++` 4.6 or higher, `python` 2.7 or higher, `python-pylab`, and `python-wget`. If all dependencies are satisfied, then the command `python perform_all_experiments.py` should automatically compile the sources and run all the experiments. Also, the implementations include some basic tests for the algorithms. These tests can be run with `sudo ./install_gtest.sh` and `python run_all_tests.py`. All the experiments also check that the generated outputs by the different algorithms match. Our implementations have slight differences with our previous theoretical analysis. For example, our KMP algorithm first builds the failure table of the pattern and then runs the following code to find the matches on a given text:

```
std::vector< int > KMP::_find_matches(std::vector<int>& prefix_table,
                                      std::string& pattern,
                                      std::string& text) {
  std::vector< int > ret;
  int n = int(text.size());
  int l = 0, r = 0;
  while(r<n) {
    while(l>=0 && text[r] != pattern[l]) l = prefix_table[l];
    ++l, ++r;
    if(l == int(pattern.size())) {
      l = prefix_table[l];
      ret.push_back(r - int(pattern.size()));
    }
  }
  return ret;
}
```

We can observe that this code is equivalent to running the Aho-Corasick matching algorithm on a single-pattern instance: we first go to a state that has a match transition defined for our current character by applying the failure function f and then apply its match transition. If we are on the last state, then we add a match to our match list.

Another difference between the theoretical analysis and the practical implementations are the nodes of the trie and the Aho-Corasick data structure. In our implementation our nodes have always $|\Sigma|$ pointers to nodes (and, since we are considering bytes, this means that each node will have 256 pointers). This affects both algorithms in terms of space: we will need $O(|\Sigma|m|p| + \#\text{matches})$ instead of $O(m|p| + \#\text{matches})$ additional space. Also, in our Aho-Corasick breadth-first traversal we will iterate over all the neighbors of all nodes (and check if their pointers are null or not to know if there is a transition defined). This means that our Aho-Corasick implementation cost is $O(n+|p|+|\Sigma|m|p|)$. We made this decision because this kind of implementations are usually faster than, for example, having a hash map of symbols and nodes. Also, we considered that, in practical instances, this extra space is perfectly affordable.

A more dangerous decision was to implement the set of accepting words of an Aho-Corasick data structure naively. For example, if we build an Aho-Corasick data structure for the set of words a, aa, aaa then we will have that the state that represents the word aaa will have a, aa, and aaa in its set of accepting words. This means that we can potentially waste $O(m^2)$ additional time and space on these sets. However, we consider that this case is very specific and not very common on real data.

3.2 Synthetic Experiments

These experiments are designed to analyze how the algorithms performance behave on the following scenarios:

1. When the text length increases
2. When the pattern length increases
3. When the number of patterns increases

For all of these cases the pattern(s) will be strings of the form a^iba^j and the text will consist of a string of the form a^n. This scenario makes the bruteforce and trie algorithm to maximize its iterations, and also it is not a very favorable case for KMP and Aho-Corasick algorithms. Graphical results for these experiments can be found in figures 5, 6, and 7. These plots summarize the evolution of required time when a specific parameter is varied for the algorithms explained in the previous section. As we can see, Aho-Corasick and KMP behave similarly on single-pattern instances, and the same happens with brute force and the trie. Also, it seems that the additional overhead introduced by more complex data structures as the trie and the Aho-Corasick graph is not negligible at all.

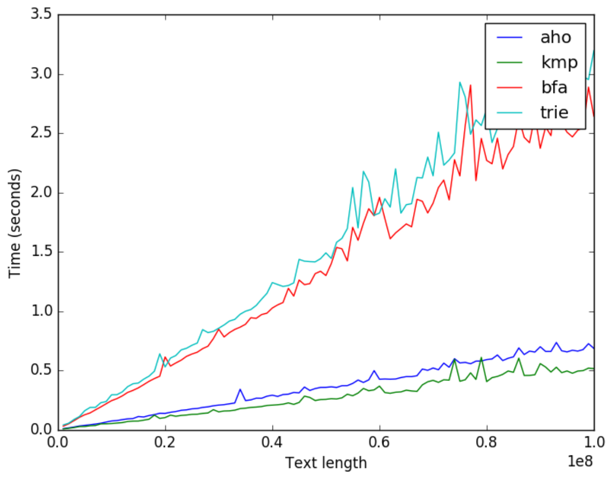

Figure 5: Evolution of the required computation time on a single-pattern instance when the text length increases. As we can observe, KMP and Aho-Corasick behave similarly, and the same happens with the brute-force and the trie. These observations are coherent with the asymptotic costs exposed in table 1. The plot also shows signs of overhead on the Aho-Corasick and the trie algorithms

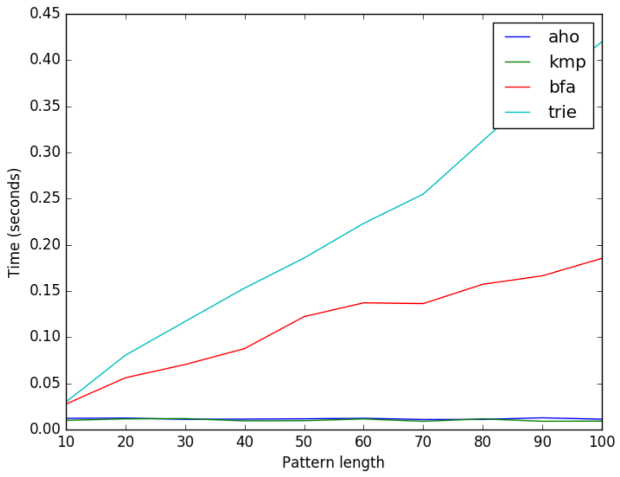

Figure 6: Evolution of the required computation time on a single-pattern instance when the pattern length increases. As a remark, the text was fixed and consisted of 10^6 characters. Given that the longest pattern length was 100, we can consider this length negligible with respect to the text length

3.3 Experiments With Real Data

In this second section our implementations are tested with real data. More precisely, our search algorithms are run on the concatenation of all the Shakespeare's writings as the

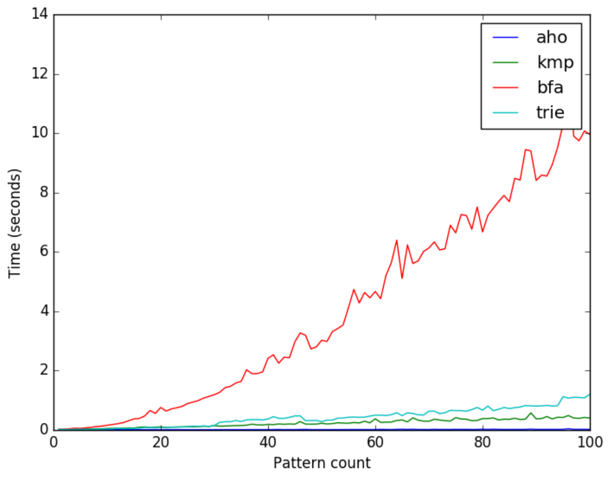

Figure 7: Evolution of the required computation time on a multi-pattern instance when the number of patterns increases. It must be noted that the ith pattern forces the bruteforce algorithm to perform i iterations, so this variable does not contribute linearly to the running time

text and the most 10000 used english words (with no swear words) as the patterns. In table 2 we can see the results. As expected, the great amount of patterns dominated the running time of the Brute-Force and the KMP algorithm, leading to enormous costs. Also, given that the patterns are not very long the extra iterations that the trie must perform are not very significant.

Brute Force	KMP	Trie	Aho-Corasick
187.622s	186.904s	1.239s	1.383s

Table 2: Elapsed time for our four algorithms on finding the occurrences of the most used 10000 english words on all the Shakespeare's writings

4 Conclusions

With these experiments we have shown that the algorithms behave as supposed, and that overhead is important.

Also, we must note that KMP and Aho-Corasick are totally equivalent on single-pattern instances, but Aho-Corasick has shown to be slower than KMP on our empirical instances. This is probably due to the fact that KMP uses arrays and Aho-Corasick uses pointers for its data structures, leading to a significant loss of data locality. A possible future work could consist in getting rid of the pointers and implementing the Aho-Corasick implicit graph in a more compact form. For example, we could first build the data structure with pointers and then convert it to an adjacency list. Another possible future work can be, as mentioned in [1], to convert the Aho-Corasick data structure to a DFA.

This project has shown something that is somehow intuitive: real-life data is usually easier than synthetic datasets. Even if, as shown in table 1, Aho-Corasick is the algorithm that performs better on multiple pattern string matching instances, it was enough to run a trie on this kind of data to get the result in a reasonable small amount of time. The reasons are two: patterns are short, and misses are usually early. Also, it must be taken into account that the conceptual complexity of these algorithms is high (even if the authors of [1] remarked that they developed the Aho-Corasick algorithm to give programmers alternatives to the even harder regular expression based algorithms).

References

[1] A. V. Aho and M. J. Corasick, “Efficient string matching: An aid to bibliographic search,” *Commun. ACM*, vol. 18, pp. 333–340, June 1975.

[2] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson, *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd ed., 2001.