DD2458, Problem Solving and Programming Under Pressure

Lecture 3: String matching

Date: 2008-09-15
Scribe(s): Jonas Moberg, Mark Bartish
Lecturer: Douglas Wikström

This lecture was about constructing an efficient data structures and algorithms for string matching. Data structures such as Knuth-Morris-Pratt (KMP) automaton for matching of a pattern in a given string were discussed. Other data structures such as a prefix-tree (a *trie*), suffix trees and suffix arrays were demonstrated as well.

1 String Matching

We often want to search a text for all occurrences of a specific pattern. Let us say we want find all occurrences of the pattern *abab* in the text string *abababccabab*. As you can see the pattern occurs at several positions in the text string. It occurs at positions 1-4, but also at positions 2-6 and positions 9-12 at the end. Finding the starting positions (1, 2 and 9) of the pattern *abab* in the string *abababccabab* is what is often referred to as the string matching problem. In this section we will discuss several algorithm with different running times for solving the string matching problem.

For the rest of this section let us assume that the pattern P is a string of m characters, $(p_1, \ldots, p_m)$, and that the text T is a string of n characters, $(t_1, \ldots, t_n)$ (where $m \leq n$).

1.1 Naive string-matching

The naive algorithm for string matching simply checks if we can find the pattern P starting at character 1 to $n - m$ in T. The outer loop on line 1 in Algorithm 1 is run for each of the $n - m + 1$ possible starting positions of the pattern, and for each of those the inner loop on line 2 is run (at most) m times. The total running time would therefore be $O((n - m + 1)m)$.

Algorithm 1: Naive string-matching.
Input: Pattern $P = (p_1, \ldots, p_m)$ and text $T = (t_1, \ldots, t_n)$.
Output: Positions in T where P start.

```
NaiveMatching(T, P)
(1)   for j ← 1 to n − m
(2)     for i ← 1 to m
(3)       if p_i ≠ t_{i+j−1} then break
(4)       if i ≥ m then output(j)
```

1.2 String matching with finite automata

A problem with the naive algorithm for string-matching is that we may examine each character in the text string several times. Here we will show a way, using automata, in which we examine each character exactly once. The matching time will therefore be $O(n)$. This however requires us to first build a automata which could take a lot of time if we have a large alphabet.

A finite automaton (or a finite state machine) consists of a finite number of states, a finite number of transitions between the states, a start state and a subset of states known as accepting states.

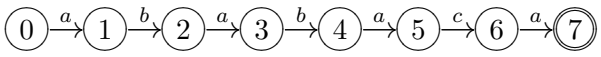

Figure 1: An automaton with states $0, 1, \ldots, 7$ and with transitions labelled a, b and c. Its start state is state 0 and its only accepting state is 7.

A finite automaton can be viewed as deciding whether or not a input string is accepted or not by the automaton. Starting in its start state the automaton reads one character at a time from an input string and follows the transitions labelled with the current character. This is repeated until we reach the end of the string. If we end up in a accepting state the string is said to be accepted by the automaton, otherwise it is said to be rejected.

If we want to use automata for string matching we will need to build one string-matching automaton for each pattern. This has to be done as a preprocessing step before we can search the text string. When we are using automata for matching strings we often want the alphabet to contain all the English letters from a to z. For the automaton to be deterministic we would have to add a transitions for every letter for each state. Doing this would however make the presentation very cumbersome. Instead we will assume that if we encounter a character in the text string that is not in the pattern we have an implicit transition to the start state.

Let us say we want to try and match the string *abacxyzababaca* with the pattern *ababaca* using the automaton in Figure 1. We would then start in state 0 and when we read the first character, a, we would move to state 1. Reading the next character from the input, b, we would move to state 2, and so on. When we reach state 3 we read c from the input and since there is no transition labelled c from state 3 we move implicitly to state 0 and start from the beginning. Since there are no transitions from state 0 for x, y or z we stay there until we read a from the input. The next 7 transitions will take us to the accepting state 7 and we have found a match.

But what if we want to match the string *abababaca* against the same automaton? It clearly matches the pattern at position 3. Reading *ababa* from the input we would move to state 5. There is however no transition from state 5 labelled b. If we go back to state 0 and start over from were we are in the input we will miss the match at position 3. However, because we have read at least *aba* (the last three characters)

and we got a b this could be a new beginning of a match and we can instead go to state 4 and continue matching from there. If we in a similar way have a look at what could happen in the other states we get the automaton in Figure 2.

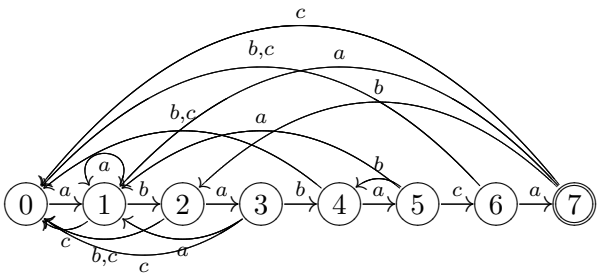

Figure 2: An automaton for the pattern *ababaca*.

As you can see we will have a transition for every character of the alphabet for each state, in this case a, b and c. When the alphabet is large (for example if we are using Unicode strings) and we have a non trivial pattern we would get a dense graph with many edges. In fact, worst case would be $\Omega(n^2)$ edges in total.

1.3 The Knuth-Morris-Pratt algorithm

The problem with the naive way of constructing a finite automaton for string matching was we could get a automaton with a lot of edges. The Knuth-Morris-Pratt algorithm is also based on finite automata, meaning it will also match a string in $O(n)$, and it will need only an auxiliary array that can be precomputed in time $O(m)$.

If we look at Figure 2 we can make an important observation. All transitions going in to a state (except for state 0) is labelled with the same character. This is no coincidence because when we go "backwards" in the pattern (when we follow a transition to a earlier state) we pretend as if we reached the state by reading the "forward" character so the characters going in to a state should be the same (state 0 is special since we stay in 0 until we find the first character of the pattern and we return to 0 when we read something that can not be part of the pattern).

Since the previously read character is always the same when entering a state we can instead see it as if we read from the input only when going "forward". The "backward"-transitions then have to go to a state from which there is a "forward"-transition labelled with the character we chose not consume from the input. A transition not consuming input is usually called a ϵ-transition as ϵ usually stands for the empty string. A ϵ-transition can be thought of as there being the option not to consume a character of the input.

If we look at for example state 5 in the automaton without ϵ-transitions there is a transition to state 4 on b and transition to state 1 on a. We could replace the transition from state 5 to state 4 on b with a ϵ-transition from state 5 to state 3 since the only way to go forward from state 3 is by reading b. This leaves us with having to replace the other transition from state 5. Looking at state 3 we see that we can replace the transition to state 1 on a in the same way. These two changes would

leave us with a ϵ-transition from state 5 to state 3 and a ϵ-transition from state 3 to state 1. Since the ϵ-transitions won't consume input we can follow as many as we like backwards. This means that we can "chain" them together and replace the transition from state 5 on a with the two previously mentioned ϵ-transitions.

Replacing all the "backward"-transitions with chained ϵ-transitions we can reduce the number of transitions in the automaton to only one forward and one backward transition for every state except state 0. Doing so for the automaton in Figure 2 we get the automaton in Figure 3.

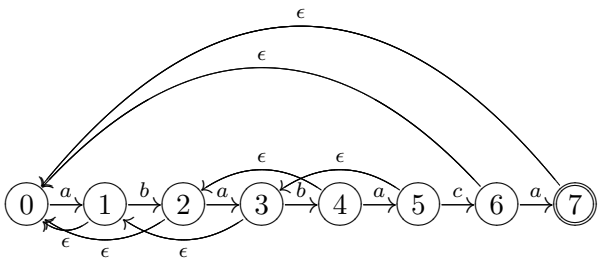

Figure 3: An automaton with ϵ-transitions for the pattern *ababaca*.

One can prove that this automaton we just described is in fact correct and can be built but we will not do so here.

Having only one "backward"-transition for each state means that we can store the automaton in a linked list. However, doing so we have to follow the links of the list (the ϵ-transitions) backwards to find which state to go to. One can prove that the amortized cost for doing this is going to stay the same as if we had a direct link. We won't prove that but the idea is to observe that we can not go further back than we can go forward.

Let us call this list π and let $\pi[i]$ be the end node for a ϵ-transition from state i. If $\pi[i] = j$ then $(p_1, \ldots, p_j)$ will be the longest prefix of $(p_1, \ldots, p_{i-1})$ such that it is also a postfix of $(p_1, \ldots, p_i)$. If we fail to match the pattern we would like to find an as long as possible matching of what we have already read. We want the longest possible prefix which would match. This is exactly what we are doing with the ϵ-transitions. If we can't find a match we move backwards following ϵ-transitions until we find a match. Now we can make the following observation:

- If $(p_1, \ldots, p_j)$ is the longest prefix of $(p_1, \ldots, p_{i-1})$ such that it is also a postfix of $(p_1, \ldots, p_i)$ and
- $(p_1, \ldots, p_l)$ is the longest prefix of $(p_1, \ldots, p_{j-1})$ such that it is also a postfix of $(p_1, \ldots, p_j)$, then
- $(p_1, \ldots, p_l)$ is the *second longest* prefix of $(p_1, \ldots, p_{i-1})$ such that it is also a postfix of $(p_1, \ldots, p_i)$.

The intuition behind building π is that we want to build an as long a prefix as possible to $(p_1, \ldots, p_{i-2})$ such that it is also a postfix of $(p_1, \ldots, p_{i-1})$. We can define π recursively as:

- $\pi[1] = 0$
- $\pi[i], i > 1$ is defined as:
  1. Set $k = i - 1$
  2. Set $k = \pi[k]$
  3. If $p_i = p_{k+1}$, set $\pi[i] = k + 1$
     if $k = 0$, set $\pi[i] = 0$
     otherwise go to step 2.

This recursion is easily translated into to the pseudo code of Algorithm 2.

Algorithm 2: Compute Knuth-Morris-Pratt prefixtable.
Input: Pattern $P = (p_1, \ldots, p_m)$.
Output: Table π such that
$\pi[i] = \max\{k | k < i, (p_1, \ldots, p_k)$ is a postfix of $(p_1, \ldots, p_i)\}$.
PREKMP(P)
(1) $\pi[1] \leftarrow 0$
(2) **for** $i \leftarrow 2$ **to** m
(3) $k \leftarrow \pi[i-1]$
(4) **while** $k > 0$ **and** $p_{k+1} \neq p_i$
(5) $k \leftarrow \pi[k]$
(6) **if** $p_{k+1} = p_i$
(7) $\pi[i] \leftarrow k + 1$
(8) **else**
(9) $\pi[i] \leftarrow 0$

Algorithm 3: Knuth-Morris-Pratt string matching algorithm.
Input: Pattern $P = (p_1, \ldots, p_m)$ and text $T = (t_1, \ldots, t_n)$.
Output: Positions in T where P start.
KMP(T, P)
(1) $\pi \leftarrow$ PREKMP(P)
(2) $j \leftarrow 0$
(3) **for** $i \leftarrow 1$ **to** n
(4) **while** $j > 0$ **and** $p_{j+1} \neq t_i$
(5) $j \leftarrow \pi[j]$
(6) **if** $p_{k+1} = t_i$ **then** $\pi[i] \leftarrow k + 1$
(7) **if** $j = m$
(8) OUTPUT($i - m + 1$)
(9) $j \leftarrow \pi[j]$

Once we have built the automaton using Algorithm 2 the matching of a text string is going to be easy. The pseudo-code for doing the matching is available in Algorithm 3. We scan the n characters of the text from left to right with the loop on line 3. The while-loop on line 4-5 amounts to following the ϵ-transitions backwards as long as we can't find a match for the current input character or until

we reach state 0. At line 6 we check if we have read a character matching some part of the pattern and in that case move forward in the pattern. At line 7 we check if we matched the whole pattern and if so we output the starting position of the match in the text string T. At line 9 we again try to find an as long a prefix of the pattern that we can match with a postfix of our text string.

It necessary to follow the ϵ-transitions backwards even when we found a match because the next match could start in the part we just matched. For example, say we want to match the text *cacacacaca* against the pattern *caca*. The first match would start at position 1 and end at position 4 in the text. The second match however would start at position 2 so we still have to find the longest prefix of the pattern that match the characters we have read.

2 Prefix trees (tries)

A *prefix tree* or *trie* (the name originates from re*trie*val) is basically a tree of nodes where the root is a collection of pointers to other nodes. Each non-root node represent a *state* (like in a KMP-automaton) and each *transition* from one node to another is a character, where a leaf node is an accepting state.

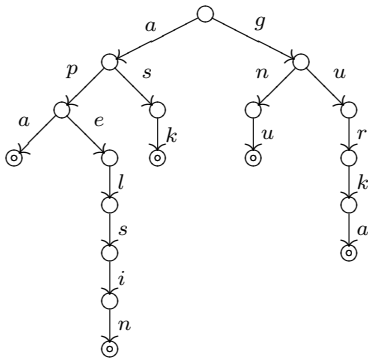

Figure 4: A trie. Each leaf node represents an accepting state (like in KMP automaton). Even non-leaf nodes can be accepting nodes.

This structure is useful for creating so called associative arrays (dictionaries) which contain a set of strings (keywords) for fast searching. Unlike a binary search tree where each node contains a string and up to two nodes, a trie-node is a node with at most as many child nodes as there are letters in the alphabet. To find a keyword in a trie, you start at the root and traverse the tree character by character until you reach a leaf node which contains the keyword.

Insertion of keywords into a trie (building a trie) works as follows:

1. Start with an empty tree, with just a root and no transitions (no pointers)
2. For the first keyword, w_1, you create nodes for every character and link them together from the root. In this tree, the last node (leaf node) is marked as "accepting" node which means that result is found.
3. For the second keyword w_2, you follow the transitions (pointers) from the root by nodes while they match existing characters in w_2 until you come to

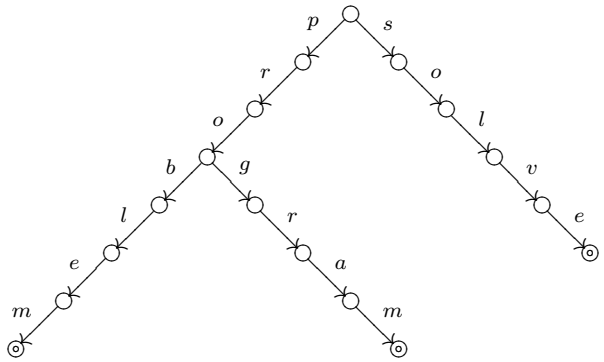

Figure 5: Keywords $\mathbf{P} = \{$**problem**, **program**, **solve**$\}$ in a trie representation.

a node that doesn't have a transition that matches the character in w_2. At that point you create a transition and a node in the same way you did for w_1.

4. And so on for every keyword until the trie is complete.

Algorithm 4: Build trie.
Description: Inserting a keyword into a trie. $g(q, c)$ is a *transition function* between node q, and character c.
Input: An array $\mathbb{W} = \{W_1, \ldots, W_n\}$ of strings, each of length L_i, $i = 1, \ldots, n$
Output: A trie.

```
BuildTrie(𝕎)
(1)   𝕋 ← ∅
(2)   for p ← 1 to n
(3)       q ← root
(4)       for j ← 1 to L_p
(5)           if g(q, W_p[j]) = null then insert(q, W_p[j])
(6)           q ← g(q, W_p[j])
(7)       𝕋 ← 𝕋 ∪ q
```

2.1 Trie as a state machine (automaton) for multiple pattern matching

While a KMP-automaton is a good way to reduce time complexity of string matching from $O(n^2)$ to $O(n)$ for a single pattern matching, what if you have several patterns $\mathbf{P} = \{P_1, \ldots, P_m\}$ to match in a text string S? A naive solution would be to do a matching (using a KMP-automaton maybe) of every pattern $P_1, \ldots, P_m$ which would require $O(nm)$ time where n is text length.

There is a clever way to do this. Consider a state machine like a KMP-automaton but using a tree instead of an array of characters and a jump table.

To find all occurrences of a set of patterns **P** in a string S you just traverse the trie $\mathbb{T}$ by characters in S, once you reach a leaf node, you've got a result. Once you fail to match a character in S to a node transition in $\mathbb{T}$ you follow the **failure**-links (or **failure**-functions) that each node has (even the root node – to itself and leaf nodes – in case there are more occurrences of a pattern to which the current suffix is a prefix). The tricky part here is to set-up the failure function for each node.

1. Set failure pointer of every root-descent node back to root.
2. Use BFS to traverse each node and set their respective failure link to point to a predecessor node in similar way to KMP-automaton.

To search for a match of any keyword in **P** in a text string S use Algorithm 5.

Algorithm 5: Aho-Corasick algorithm.
Description: Searching for all occurrences of all patterns $P \in \mathbf{P}$.
Input: A string S. Note: the patterns to be searched for are already added to the tree.
Output: A set $\mathbb{R} = \{\{r_1, r_2, \ldots, r_{L_1}\}, \ldots, \{r_1, r_2, \ldots, r_{L_m}\}\}$ of result arrays for every pattern $P \in \mathbf{P}$.

```
AhoCorasick(S)
(1)   p ← root
(2)   for i ← 1 to Length(S)
(3)     trans ← null
(4)     while trans = null
(5)       trans ← g(p, S[i])
(6)       if p = root then break
(7)       if trans = null then p ← Failure(p)
(8)     if trans ≠ null then p ← trans
(9)     foreach r ∈ Results(p)
(10)      R_IndexOf(r) ← R_IndexOf(r) ∪ i
(11)  return R
```

For Algorithm 5 we have:

1. Let $\mathbb{R}$ be the set of sets of matches of any $P \in \mathbf{P}$. for example $\mathbb{R}_i$ contains the set of indexes for all matches of P_i.
2. Let *trans* be a transition from a node to another.
3. Let $g(q, c)$ be a transition function from node q to character c.
4. Let Failure(q) be the failure function, a pointer from q that points to a node to go to in case of mismatch
5. Let Results(q) be a set of results – the strings that make a complete match. Each leaf node and non-leaf nodes that represent an accepting stare have a results set.

This Algorithm is known as **Aho-Corasick** multiple pattern matching algorithm and was created by Alfred V. Aho and Margaret J. Corasick.

3 Suffix trees

In its simplest form a suffix tree is simply a trie of the n suffixes of a given string of length n. We could therefore insert all the suffixes of the string in a trie to get a suffix tree. This would require $O(n^2)$ time as we would be inserting n strings each taking $O(n)$ time. The problem with creating a suffix tree this way is that it would require $O(n^2)$ space. In Figure 6 we can see the trie containing the suffixes of the string *bananas*.

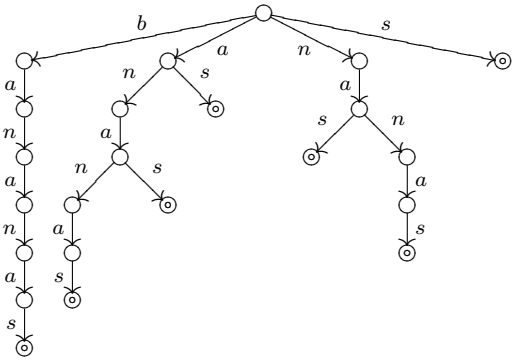

Figure 6: A trie containing the suffixes of the string *bananas*.

Looking at the trie in Figure 6 we can see that only leafs are end nodes. We can therefore reduce the space needed by compressing paths in which every node has only one child. If we do this for the trie in Figure 6 we get the tree in Figure 7.

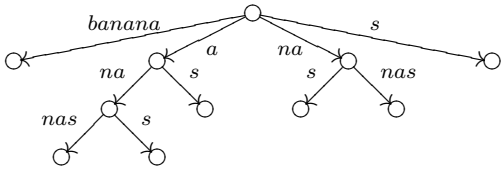

Figure 7: A tree containing the suffixes of the string *bananas* with compressed nodes.

Suffix trees have many uses. For example we can use a suffix tree of a string S of length n to find all occurrences of a string P as a substring S. This can be done in time $O(m)$ if S has length m. Other applications include finding the longest common substring between two strings.

There is a faster algorithm that constructs a suffix tree in $O(n)$ time and space but we won't describe it here.

4 Suffix Arrays

Suffix trees are useful for many things but they can be a little bit tricky to build efficiently in time and space. A naive algorithm to build a suffix tree takes $O(n^2)$ both time and space. Even if there are algorithms to build suffix trees in linear time, they are difficult to implement. An alternative way to create a suffix container data-structure is to use a suffix array. Suffix array is simply an array of all suffixes of a string in an alphabetically sorted order. In Figure 8 we have the suffixes of the string **paper** and it's suffixes sorted.

Figure 8: The suffixes of the string **paper** on left and the suffixes sorted on the right.

To sort such an array makes it possible to do a binary search for a given suffix and find its position in S. In this manner suffix arrays can be used to quickly locate a substring in a string. Finding occurrence of a pattern P in S equals finding a suffix that begins with P. We can also retrieve ith largest suffix of the string by it's index in $O(1)$ time.

Main advantages of a suffix array over a suffix tree is that it's more efficient in terms of space utilization. All we need to create an object for suffix array representation is indexes in the string of all suffixes in sorted order. However sorting such an array using conventional sorting algorithms requires $O(n^2 \log n)$ time, where n = Length(S) (since comparison of every character must be made, but there is a clever way to reduce the cost for each comparison. After sorting by the first letter (and ignoring the rest) we have already made all the comparisons of every character in S.

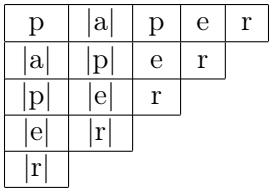

Figure 9: We can observe that the second column contain the same elements as the first except for one element.

This can be used to avoid redundant comparisons. Like this:

1. Sort by the first 2^k characters in each suffix, $k = 1, \ldots, \log n$, and ignore the rest.

2. When we sort by the first 2^k characters:
   The first 2^{k-1} characters already sorted
   The next 2^{k-1} characters are already compared in other suffixes

This reduces time complexity for constructing a suffix array to $O(n(\log n)^2)$. There are more sophisticated ways to further reduce the complexity, but they are beyond the scope of this lecture.

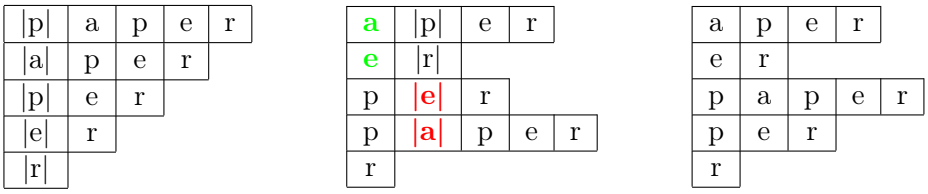

Figure 10: The left figure show the original suffixes before sorting the first column. The middle figure show the result of sorting on the first column. When we want to sort on the second column we ignore the first column and can use the fact that we have already sorted the suffixes **per**, **r**, **er** and **aper** in the first round. We can now use the relative order of these prefixes when sorting the second column. The last figure show the sorted suffix array.