Efficient implementation of Unicode string pattern matching automata in Java

Janne Nieminen

Report A/2005/2

ISBN 951-781-270-1
ISSN 0787-6416

UNIVERSITY OF KUOPIO
Department of Computer Science

P.O.Box 1627, FIN-70211 Kuopio, FINLAND

Efficient implementation of Unicode string pattern matching automata in Java

Janne Nieminen
University of Kuopio
Department of Computer Science

November 8, 2005

Abstract

We study different efficient implementations of an Aho-Corasick pattern matching automaton when searching for patterns in Unicode text. Much of the previous research has been based on the assumption of a relatively small alphabet, for example the 7-bit ASCII. Our aim is to examine the differences in performance arising from the use of a large alphabet, like the 16-bit Unicode that is widely used today. The main concern is the representation of the transition function of the pattern matching automaton. We examine and compare array, linked list, hashing, balanced tree, perfect hashing and triple-array representations. For perfect hashing, we present an algorithm that constructs the hash tables in expected linear time and linear space.

We implemented the Aho-Corasick automaton in Java using the different transition function representations, and we evaluate their performance. Triple-array performed best in our experiments, with perfect hashing, hashing and balanced tree coming next. We discovered that the array implementation has a slow preprocessing time when using the Unicode alphabet. It seems that the use of a large alphabet can slow down the preprocessing time of the automaton considerably depending on the transition function representation used.

Keywords: string pattern matching, Aho-Corasick, implementation, transition function

1 Introduction

String pattern matching has many applications: lexical analysis phase of a compiler, bibliographic search, and more recently, web search engines and computational biology. A specific pattern matching problem is to locate all occurrences of a finite number of keywords in a string of text. An efficient algorithm to solve this problem has been presented by Aho and Corasick [1]. This algorithm is based on building a pattern matching automaton to scan the text.

A central implementation concern of this pattern matcher is the representation of the transition function of the machine. Possible data structures to use are array, linked list, binary search tree and hashing [11]. One special form of hashing that is applicable here is perfect hashing [10]. Another elaborate data structure, the *triple-array*, has been proposed by Aho, Sethi and Ullman [2] as an efficient implementation of a transition function. Aoe [3] has introduced an improvement over the triple-array, called the *double-array* structure. To our knowledge there has been little experimental research addressing the problem of transition function representation for pattern matching automata. Arikawa and Shinohara [4] have studied the run-time efficiency of an Aho-Corasick machine with a few different representations of the transition function.

Up until recently, many applications of string pattern matching have been assuming the use of a relatively small alphabet, for example the ASCII character set that has the size of 128 symbols. With the advent of Java, the use of Unicode as a character set has been growing more and more popular. Unicode has 65536 different character codes, and therefore potentially presents different problems in pattern matching. Also, large alphabets arise naturally in the growing field of computational biology [11]. To our knowledge, there has been little research addressing the use of large alphabets in string pattern matching machines.

The purpose of this paper is to experimentally compare the performance of different representations of the transition function of an Aho-Corasick pattern matching automaton using Unicode character set. We also analyze the problems that are a result of using a large alphabet. The automaton is implemented using Java. We discuss and compare array, linked list, hashing, perfect hashing, red-black tree, and triple-array representations. We measure the speed of these implementations, taking into account the time to build the automaton. The paper by Arikawa and Shinohara [4] only addresses run-time speed; however, we will see that preprocessing speed can be important when using large alphabets, so we include it in our measurements.

We have not found any data on how perfect hashing compares in speed to other simpler data structures when implementing string pattern matching machines. Aoe mentions in his paper [3] that the double-array structure can be used in implementing the transition function instead of perfect hashing, but does not provide any comparison. We explore this by implementing the triple-array structure, which is the basis of the double-array structure, and compare its performance to that of perfect hashing and other representations. This gives indication on the performance of the double-array compared to other structures.

Our experiments show that for the Unicode alphabet, an array implementation of the transition function is surprisingly not always the fastest implementation due to a slow preprocessing time, especially when the number of patterns increases. Linked list, perfect hashing, and triple-array also each suffer from slow preprocessing, with triple-array preprocessing being clearly slowest. This however only becomes evident on very large pattern sets. Overall the fastest implementations are those based on a triple-array and perfect hashing, with ordinary hashing and red-black tree coming not far behind. Both perfect hashing and triple-array offer theoretical constant run-time access speed, but triple-array is faster in our experiments. Linked list is clearly the slowest of these implementations.

The preprocessing stage of the search has two potential bottlenecks: the first is the building of the trie where we have to check if a transition is already present, and the second is the phase where we build the failure function using a breadth-first search. The performance of a data structure during these two phases largely determines the preprocessing speed of the implementation. The speed of the preprocessing stage of the search gains importance if the length of the target text is small.

The rest of this paper is organized as follows. In Section 2, we define a pattern matching automaton and present algorithms that build the automaton and perform the search. In Section 3 we present the different transition function representations. In Section 4 we discuss the implementation details of the pattern matching automaton, and in Section 5 we present and analyze the experimental data. Section 6 is a summary of the results.

2 Pattern matching automaton

An Aho-Corasick pattern matching automaton [1] is defined for a finite set of strings K as $M = (S, \Sigma, g, f, h)$, where

- S is a finite set of states. Each state is represented by a number. 0 is designated as an initial state.

- Σ is an input alphabet.
- $g : S \times \Sigma \to S \cup \{\mathit{fail}\}$ is a transition function.
- $f : S - \{0\} \to S$ is a failure function.
- $h : S \to \mathcal{P}(K)$ is an output function, where $\mathcal{P}(K)$ is the power set of K.

A pattern matching automaton is built from the pattern set K. The automaton scans the target text one character at a time, and outputs any locations in the text that contain a pattern in K. The patterns may overlap with each other. Initially the automaton starts in state 0. Now assume that the automaton has reached state s. After reading an input symbol a, we check if $g(s, a) \neq \mathit{fail}$. If true, the automaton changes to state $g(s, a)$. If $g(s, a) = \mathit{fail}$, the automaton changes to state $f(s)$ using the failure function, until $g(s, a) \neq \mathit{fail}$. Finally, the automaton outputs $h(s_n)$ using the output function, where s_n is the state reached after reading the input symbol a. The value of the output function may be empty. The automaton repeats this sequence of actions until all the characters of the target text are processed. This search is described in Algorithm 1 [1].

Algorithm 1 The search phase of the Aho-Corasick automaton for target text string $a_1 a_2 \ldots a_n$.

```
state = 0;
for $i = 1$ to $n$ do
    while $g(\text{state}, a_i) = \mathit{fail}$ do
        state = $f(\text{state})$;
    end while
    state = $g(\text{state}, a_i)$;
    print $h(\text{state})$;
end for
```

The automaton is built using Algorithms 2 and 3 [1]. In Algorithm 2, the transition function g and a part of the output function h are computed. Notable here is that when adding a new pattern p to the automaton in the while loop of the *enter*-method, we have to check for each character a of the pattern if $g(\mathit{state}, a) \neq \mathit{fail}$, where *state* is the current state of the automaton. In Algorithm 3, the failure function f is computed and the output function h is completed. Here note that we execute a breadth-first search of the states in the while loop. This means that for each state s, we have to follow each transition out from s. We will see that these two requirements are important in the efficient implementation of the transition function.

Algorithm 2 Phase one of the construction of the Aho-Corasick automaton for patterns $K = \{k_1, \ldots, k_n\}$, where the transition function g and part of the output function h are calculated.

```
newState = 0;
for $i = 1$ to $n$ do ENTER($k_i$);
end for
Set $g(0, a)$=0 for each $a$ such that $g(0, a) = \textit{fail}$;

procedure ENTER($a_1 a_2 \ldots a_m$)
    state = 0;
    $j = 1$;
    while $g(\text{state}, a_j) \neq \textit{fail}$ do
        state = $g(\text{state}, a_j)$;
        $j = j + 1$;
    end while
    for $p = j$ to $m$ do
        newState = newState + 1;
        $g(\text{state}, a_p)$ = newState;
        state = newState;
    end for
    $h(\text{state}) = \{a_1 a_2 \ldots a_m\}$;
end procedure
```

Algorithm 3 Phase 2 of the construction of the Aho-Corasick automaton, where the failure function f is calculated and the output function h is completed.

```
empty queue;
for all $a$ such that $g(0, a) = s \neq 0$ do
    add $s$ to queue;
    $f(s) = 0$;
end for
while queue is not empty do
    remove $r$ from queue;
    for all $a$ such that $g(r, a) = s \neq \textit{fail}$ do
        add $s$ to queue;
        state = $f(r)$;
        while $g(\text{state}, a) = \textit{fail}$ do
            state = $f(\text{state})$;
        end while
        $f(s) = g(\text{state}, a)$;
        $h(s) = h(s) \cup h(f(s))$;
    end for
end while
```

3 Transition function representations

The most straightforward implementation of the transition function is to code it as a two-dimensional array A indexed by a character a and a state s. Because in a typical pattern matching automaton each state contains only a few transitions, the array A is essentially a sparse matrix. In this representation we have to use some character code such as -1 as meaning "not defined" when there is no transition out of state s on character a, i.e. when $g(s, a) = \textit{fail}.$.

The array representation has a serious drawback: we need to have an Σ-sized array for each state of the automaton. When Σ is large, say 65536 as in Unicode, the amount of space required for the transition function grows high as the number of states of the automaton increases. Because in a state s the transition function is not defined for most characters, it is possible to store transitions for those characters a only for which $g(s, a) \neq \textit{fail}$. This is the basis for all the other data structure representations that we consider. We can store the transitions from each state in a linked list, a balanced tree or a hash table. Let n be the number of transitions leaving from a state of the automaton. In the worst case a linked list has $O(1)$ insertion time and

$O(n)$ access time, a balanced tree has $O(\log n)$ insertion time and $O(\log n)$ access time and a hash table has $O(1)$ insertion time and $O(n)$ access time. However, the average access time of a hash table is $O(1)$.

Two more elaborate implementations are possible. We discuss these next.

3.1 Perfect hashing

In perfect hashing we select the hash function so that the worst case access time is $O(1)$. This can be done by taking advantage of the static set of keys for an Aho-Corasick automaton: once we have computed the transition function, no updates or deletions will occur. There are a number of techniques that can be used for perfect hashing [13, 7, 10]. We use here a method introduced by Fredman, Komlós and Szemerédi [10]. This method was chosen because it is relatively simple to implement and it has been described and analyzed thoroughly in the textbook by Cormen, Leiserson, Rivest and Stein [8].

The idea is to use two-level hashing and to choose the hash function from a *universal* class of hash functions [6] at each level. The hash tables of these two levels are called the primary and secondary hash tables. Fredman et al. have shown that by using a universal hash function with the two-level hashing scheme and by choosing the size of the primary and secondary hash tables carefully, the expected worst case access time for this form of hashing is $O(1)$. We use the following class of universal hash functions [6, 8]:

$$h_{a,b}(k) = ((ak + b) \bmod p) \bmod m,$$

where m is the size of the hash table, $p \geq m$ is a prime, $a \in \{1, 2, \ldots, p-1\}$, $b \in \{0, 1, \ldots, p-1\}$, and k is the key to hash.

The textbook by Cormen et al. [8] describes this form of perfect hashing, but does not provide an algorithm to build the hash tables. In Algorithm 4 we present a straightforward implementation of this perfect hashing scheme. It is a Las Vegas algorithm that builds the perfect hash tables in linear expected time and linear space.

The algorithm proceeds in two phases. In phase 1 we hash the keys to linked lists in the primary hash table. First we choose the four parameters a, b, m and p for the hash function. A prime p is chosen that is greater than any key value, and the size m of the hash table is set to the number of keys n. The hashing is done in the repeat-until loop by selecting parameters a and b randomly from the designated intervals, and hashing the keys k to lists K_i in the primary hash table, where $i \in \{0, 1, \ldots, m-1\}$. We continue the loop until we have discovered such values of a and b that the expression $\sum_{i=0}^{m-1} |K_i|^2$, which corresponds to the combined size of the secondary hash

Algorithm 4 Algorithm for construction of a perfect hash table for patterns $K = \{k_1, \ldots, k_n\}$

.

```
// Phase 1: build the primary hash table
Choose a prime $p$, where $p \geq \max\{k_1, \ldots, k_n\}$;
$m = n$;
**repeat**
    Randomly choose $a \in \{1, \ldots, p-1\}$ and $b \in \{0, \ldots, p-1\}$;
    **for** $i = 0$ to $m - 1$ **do** $K_i$ = empty list;
    **end for**
    **for all** $k \in \{k_1, \ldots, k_n\}$ **do**
        $j = ((ak + b) \bmod p) \bmod m$;
        Add key $k$ to list $K_j$;
    **end for**
    size = $\sum_{i=0}^{m-1} |K_i|^2$;
**until** size $< 4n$

// Phase 2: build the secondary hash tables
**for** $i = 0$ to $m - 1$ **do**
    // Now $K_i = \{k \in K \mid h(k) = i\}$;
    $m_i = |K_i|^2$ ;
    **repeat**
        Randomly choose $a_i \in \{1, \ldots, p-1\}$ and $b_i \in \{0, \ldots, p-1\}$;
        Hash keys $k \in K_i$ to a secondary hash table $i$
        with the hash function $h_i(k) = ((a_i k + b_i) \bmod p) \bmod m_i$;
    **until** NOT collisions
**end for**
```

tables, is less than $4n$.

In phase 2, the secondary hash tables are built. In the for-loop we go through all the lists of the primary hash table. The keys in each list are hashed to a secondary hash table. First we set the size m_i of the secondary hash table to be the square of the number of keys in the list, and let p be the same prime as in phase 1. Then in the repeat-until loop the selection of the parameters a_i and b_i of the hash function is executed in a similar manner as in phase 1. This time we abort if any collisions are detected. The loop is executed until we find parameters a_i and b_i that do not produce any collisions. After this has been done for all buckets $i = 1, 2, \ldots, m - 1$ of the primary hash table, the hashing is complete.

Next we will show that the expected time complexity and space complexity of Algorithm 4 is linear to the number of keys n.

Theorem 1. *The expected time complexity of Algorithm 4 is $O(n)$.*

Proof. In phase 1 the prime p can be chosen in constant time. The first repeat-until loop is executed until the size needed for the secondary hash tables is less than $4n$. Let q be the probability of this event. It has been shown [8] that $q > 1/2$ if we use a universal hash function at each level. Let X be a random variable that counts the number of repetitions required for the condition of the loop to become true. Because X is geometrically distributed, it follows that the expected value $E(X) = 1/q < 2$, so the loop is executed on average less than 2 times. Inside the repeat-until loop the time complexity of the two for loops and the evaluation of the sum is $O(n)$, so in all phase 1 takes $O(n)$ time.

In phase 2 the second repeat-until loop is executed until no collisions occur. In a similar way as in phase 1, we see that this loop is also executed less than 2 times on average. We can assume that we can get the size of the number of keys hashed to a primary hash table list in constant time. In the two nested loops we hash every key k on the average less than 2 times. Because there are n keys in total, it follows that the time required by phase 2 is $O(n)$. Therefore the expected time complexity of Algorithm 4 is $O(n)$. $\square$

Theorem 2. *The space required by algorithm 4 is $O(n)$.*

Proof. The size of the primary hash table is n, and in phase 1 we need additionally $O(n)$ space for the keys in the lists. Phase 1 has ensured that the space required for the secondary hash tables is less than $4n$, so the space required by Algorithm 4 is $O(n)$. $\square$

The use of perfect hashing for implementing the transition function using Algorithm 4 requires that we know beforehand the number of keys n that we are going to hash. Because this is not known for each state of the Aho-Corasick automaton until after executing Algorithm 2, we chose to build the transition function first using another structure and then use this auxiliary structure to build the perfect hash tables. We used the linked list implementation for this purpose for its simplicity.

3.2 Triple-array

The triple-array data structure [2] consists of three arrays *base*, *next*, and *check* (see Figure 1). The idea is that when searching for $g(s, a)$ for state s and character a, we first retrieve the value base[s], and then inspect the value check[base[s]+a], where character a is treated as an integer. If the value check[base[s]+a]=s, then $g(s, a)$ = next[base[s]+a], otherwise $g(s, a) = \textit{fail}$. The triple-array allows for fast indexing and saves space when compared to an ordinary array implementation. The structure makes use of the fact that the matrix of a transition function is sparse, so it is possible to overlap the rows of the matrix to a single array. The amount of space saved depends on how well we can fill the next and check arrays. We used a first-fit-decreasing method suggested by Aoe [3] to build the arrays in our implementation. The pseudocode is given in Algorithm 5.

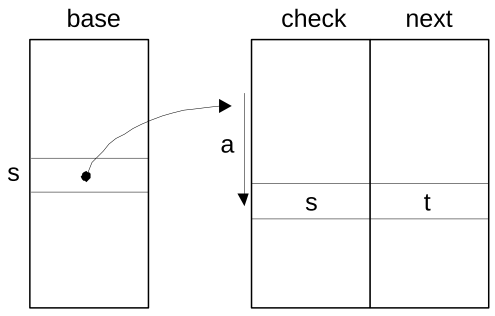

Figure 1: The triple-array structure with the three arrays *base*, *next*, and *check* for $g(s, a) = t$.

The input for the algorithm is an array *list* that contains a list of outgoing transitions for each state, and an array *count* that contains the number of transitions going out from each state. The size of the count array is the number of states n. The algorithm starts with sorting the states in decreasing order by the number of outgoing transitions. First we construct an array *bucket* that contains a list for each possible number of outgoing transitions: bucket[4] would be a list of the states that have four outgoing transitions. The bucket array is initialized in the first loop. The size of the bucket array

Algorithm 5 Algorithm for construction of the triple array by a first-fit-decreasing method.

```
for $s = 1$ to $n$ do
    add state $s$ to list bucket[count[$s$]];
end for
for $p = m$ downto 1 do
    for all $s$ in bucket[$p$] do
        overlap:
        for all $a$ in list[$s$] do
            if check[base[$s$]+a] $\neq -1$ then
                base[$s$] = base[$s$] + 1;
                goto overlap;
            end if
        end for
        for all $a$ in list[$s$] do
            check[base[$s$]+$a$] = s;
            next[base[$s$]+$a$] = $g(s, a)$;
        end for
    end for
end for
```

is the size of the alphabet m, because at most we can have a transition for each character in the alphabet going out from a state. The entries in the arrays *base* and *next* are assumed to be initialized to 0. The entries in the check array are initialized to -1, because we use 0 for the initial state of the automaton. The size of the *next* and *check* arrays is not known in advance, because we cannot be sure on how well the transitions will overlap. In the worst case the size could be $n \times m$ if no overlapping can be done. In practice a smaller size can be chosen by empirical observations.

Next we start to search for values for the *base* array that determine the starting indices in the *next* and *check* arrays. We iterate through all the states in decreasing order of their out-degree. For each state we try to fit the transitions to the next array starting from index 0 so that no collisions occur, meaning that no two transitions map to the same *next* array index. If a collision occurs, we abort and start over by increasing the corresponding *base* array value by one, and iterating through all the transitions for the state again. When an index is found that produces no collisions, the *next* array is updated with the next state given by the transition, and the *check* array is updated with the number of the state that we are processing. This whole procedure is repeated until all the transitions for each state have been successfully stored in the *next* array without collisions.

In the worst case the triple-array access time is $O(1)$ and building time is $O(n^2m + m^2n)$ [3]. The average building time for triple-array is difficult to analyze, because the average time for the first-fit-decreasing method depends largely on how sparse the matrix is and how uniformly the values in the matrix are spread. The building time of the arrays directly contributes to the preprocessing time of the pattern matching automaton. Thus, the actual performance of the triple array implementation is not obvious from the theoretical worst-case analysis.

When constructing the triple-array, we need to have the transition function already built. The list implementation was used for this purpose as in perfect hashing.

4 Implementation

The implementation was done using Java 2 SDK version 1.4.2. We decided to use the Java Collections Framework libraries for most of the underlying data structure implementations. The library offers fast and tested implementations of linked list, hash table, and balanced tree data structures. For list implementation the class `LinkedList` was used. For implementing the hash table, we used the `HashMap` class, which uses a bitwise shift-add-xor hash function that is documented in a Sun Developer Network bug report [14]. For the tree implementation, the class `SortedMap` which implements a red-black tree was used.

In all the implementations, the idea is to save the transition function representation for each state of the automaton in a `HashMap` object. A lookup of a value $g(s, a)$ is done by getting the data structure (e.g. list) for state s from the `HashMap` and executing the get operation on the underlying structure. Using an array could have been possible here, but we decided to use `HashMap` because the number of states of the automaton is not known in advance. Also this way we only have to store those states which have transitions. In all other structures besides perfect hashing and triple-array we build the transition function representation dynamically by adding each transition to the automaton one at a time. Perfect hashing and triple-array transition functions are built by first storing the transitions in a list and then using it to build the final structure.

5 Experimental results

A freely available plain text version of the Bible [5] was used as a target text. Each test was run 20 times, and outliers were removed from the samples by using the Generalized ESD Test [12]. The number of outliers to search for was set to 3 in the test. The final time was calculated as a mean of the remaining values.

First, we measured the matching time with different sizes of the pattern set. Pattern sets consisted of words randomly chosen from the target text. The sizes of the sets were $10, 20, \ldots, 100$. These were assumed to be typical pattern sizes for natural language text searching applications. In the matching process 500,000 characters of the target text were read. This was found to be enough for the differences between the implementations to be noticeable.

5.1 Run-time speed

Results measuring only the run-time speed are shown in Figure 2. Here we see that list is clearly the slowest of the implementations. The next three, red-black tree, hashing, and perfect hashing, are quite even. Triple-array and array implementations are the fastest, with both performing similarly. The relative order of the structures in this test is largely as expected based on their time complexity. The speed of the implementations is largely independent of the number of patterns. Interesting to note is that the triple-array is clearly faster than perfect hashing, although both have $O(1)$ worst-case access time. The performance of the triple-array and array is about the same, in spite of the more complex lookup operation of the triple-array.

5.2 Combined run-time and preprocessing speed

When we include the preprocessing time in the measurements (see Figure 3) we see interesting behavior: the speed of the array implementation deteriorates as the number of patterns increases. This was found to be due to a surprisingly slow preprocessing time for the array. This is examined further later when we measure only the preprocessing time. The relative order of the other structures remains the same.

In another test we measured the combined preprocessing and run-time speed of the implementations while varying the target text length. The number of patterns to search for was 30. We varied the length of the text from 100,000 to 1,200,000 characters. Results are shown in Figure 4. Here the slow preprocessing time of the array is also evident: on smaller sizes of

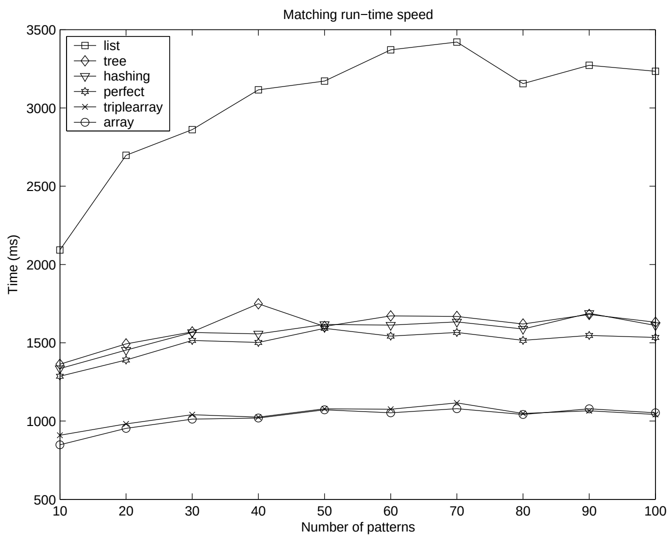

Figure 2: Matching speed excluding the preprocessing time with different number of patterns.

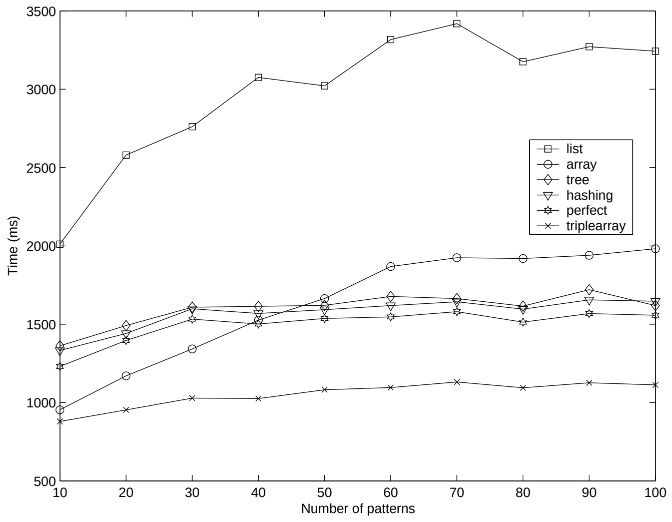

Figure 3: Combined run-time and preprocessing speed with different number of patterns.

the target text the array is among the slowest structures, but as the target text size grows the array becomes faster. This is because the relative share of the preprocessing time of the total search time grows smaller when the text size increases. On this experiment the array becomes the second fastest implementation only after the target text size of about 700,000 characters. The triple-array is faster than array on all tested lengths of the target text.

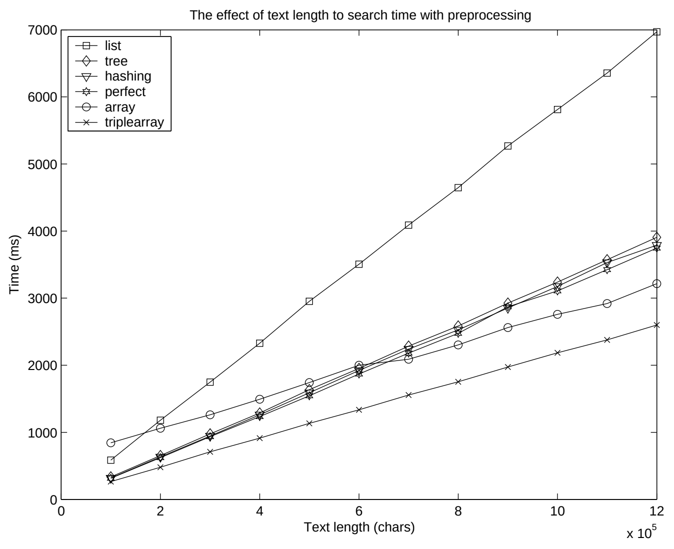

Figure 4: Matching speed including the preprocessing time with different lengths of the target text.

5.3 Preprocessing speed

Next we measured the preprocessing time only. Because the performance in this test differs quite a lot between different structures, we present the results as a table instead of a figure (see Table 1). Here we see that the preprocessing time for list, red-black tree, hashing, and perfect hashing is negligible, only a couple of milliseconds. The triple-array preprocessing time seems to grow much faster than these four. However, it is evident that the preprocessing time for array representation is significantly slower than any other structure. It takes almost a second to preprocess the array for 100 patterns, when for other structures it takes only a couple of milliseconds, or just under a hundred for triple-array. The slowness of the array preprocessing was found to be due to the breadth-first search performed in the stage 2 of the preprocessing. In this phase we need to go through all the transitions that go out from a state. When the transition function is represented by an

Table 1: The preprocessing time (ms) for different representations.

#patterns	list	tree	hash	perf.h.	triple-a.	array
10	2	2	1	4	8	148
20	1	1	1	2	12	285
30	1	1	1	2	16	321
40	2	2	2	3	31	602
50	3	2	2	4	39	662
60	3	2	3	5	56	753
70	3	3	3	5	66	808
80	5	4	3	7	93	958
90	4	3	3	6	82	964
100	4	3	4	6	83	915

array, we have to examine each index of the array to check if there is indeed a transition out for each character. When the alphabet size is 65536 and we have to do this for each state of the automaton, this phase of preprocessing is slow for array representation especially when the number of patterns is large, which leads to many states for the automaton.

The triple-array preprocessing time is affected by the first-fit-decreasing method that is used to build the structure, and seems to be approximately quadratic to the number of patterns. The performance according to the previous test was good enough for the pattern set sizes up to 100, but by increasing the size of the pattern sets to 400, we see that the performance of the triple-array preprocessing quickly deteriorates when compared with hashing, for example (see Figure 5). We were able to make the triple-array implementation significantly faster by using an array of int as the base-table data type instead of a `HashMap`. This way we could use the basic data type of `int` instead of `Integer` objects and thus avoid the overhead of creating many small objects. Using this implementation, the time for preprocessing the 400 pattern set was cut to just over 400 ms compared to 1800 ms. This was a big improvement, although the growth rate seemed to stay quadratic. However, this change in implementation would have made the comparisons between the other structures unfair, as they use a `HashMap` for storing the transitions for each state. We chose to use the `HashMap`-based implementation in the other tests, and in spite of this the triple-array performed consistently best of the representations for the smaller pattern set sizes.

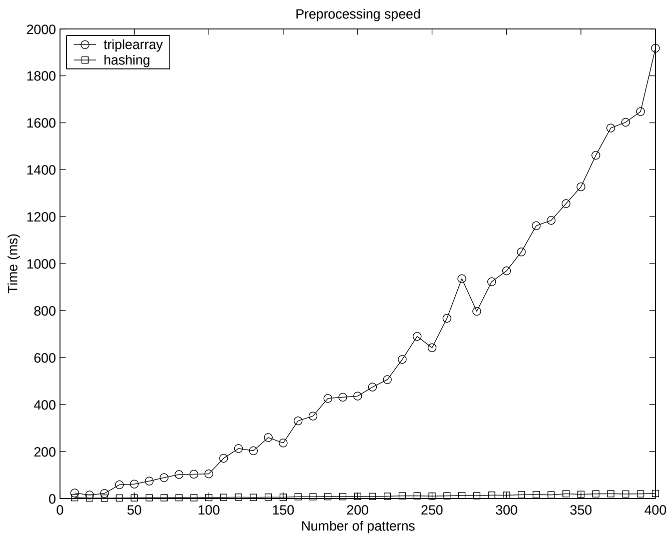

Figure 5: Preprocessing speed of triple-array and hashing for larger pattern sizes.

5.4 Preprocessing speed with large pattern sets

In the final test, we examined further the differences between the preprocessing time for those structures that were fastest in the first preprocessing time test, namely list, hashing, perfect hashing, and red-black tree. In this test we used special "pathological" pattern sets. The patterns were constructed so that the first character was different for each pattern in a set, so for example the 5-pattern set was

{``(AAAA'', ``)AAAA'', ``*AAAA'', ``+AAAA'', ``,AAAA''}.

The generation of the first letter of the patterns was started from Unicode character 40. The intent of this set of patterns was to increase the number of transitions out from the initial state in order to create a worst-case scenario. We also used larger pattern sets that consisted of up to 4000 patterns. The results are shown in Figure 6.

In this test we see that the preprocessing of perfect hashing and list is slowed down considerably for large pattern sets. The similar behavior of list and perfect hashing is due to the use of a list as an auxiliary structure used in the construction of the perfect hash tables. Intuitively the list would appear to have a preprocessing time of $O(l)$ due to constant insertion time. However, the preprocessing time for list is surprisingly $O(l^2)$, where l is the total number of non-fail transitions. This is because, as discussed in Section 2, in phase 1 of the preprocessing it is necessary to check if transition for a character a is present before adding it to the transition function. Because this lookup has to be done for each transition, the time to insert all l transitions is $O(l^2)$. The performance of perfect hashing for large pattern sets could be improved by using another auxiliary structure instead of a linked list that has a faster preprocessing time. Hashing or a balanced tree could be used for this purpose.

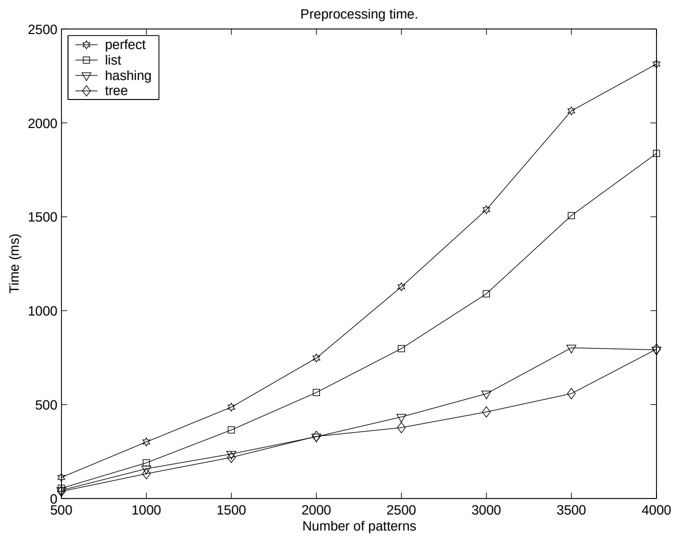

Figure 6: Preprocessing speed for pathological pattern sets with a large number of patterns.

Red-black tree and hashing performed best in this test, with red-black tree slightly faster than hashing. This is interesting because the insertion time of red-black tree is $O(\log n)$ and of hashing $O(1)$. One thing that could affect this is that the iteration of all hash table values takes time in proportion to the size of the hash table, which is always larger than the number of keys in the table. The iteration of a red-black tree can be done with a straightforward $O(n)$ tree traversal algorithm. In the preprocessing we do more iterating than simple insert operations. The performance of both structures is consistently good for patterns set sizes of several thousands.

6 Conclusions

We have examined several different transition function representations for implementing an Aho-Corasick pattern matching automaton in Java using Unicode alphabet. Besides the more traditional structures of array, linked list, balanced tree, and hashing, we studied two more sophisticated ones:

perfect hashing and triple-array. We also proposed an algorithm that builds the perfect hash tables in linear time and space.

In our experiments the triple-array performed best of the representations. Also good alternatives were perfect hashing, ordinary hashing, and red-black tree. We noticed that array implementation is not suitable for transition function representation for Unicode alphabet. In addition to requiring a lot of memory, the Aho-Corasick preprocessing using the array was found to be very slow for a large number of patterns. Thus, for pattern set sizes of up to 100, triple-array structure can be recommended as it was fastest in our tests.

For larger pattern sets of a few hundred patterns, the preprocessing stage for our implementation of the triple-array slowed down considerably. This could be offset by using the basic data type of `int` instead of `Integer` objects in Java. However, it seems that the quadratic time performance of the triple-array preprocessing is due to the first-fit-decreasing method used in building the structure. For other structures besides array and triple-array, the preprocessing time was only a couple milliseconds even for 400 patterns. Therefore for pattern set sizes of a few hundred, red-black tree, hashing or perfect hashing can be recommended if slow preprocessing time of the triple-array affects the overall speed of the search.

We also used very large "pathological" pattern sets in order to study if there were notable differences between the preprocessing times between linked list, hashing, red-black tree, and perfect hashing structures. In this test with the sizes of the pattern sets of several thousands, the difference between preprocessing times became evident: list and perfect hashing have quadratic time performance, while hashing and red-black tree are much faster. The quadratic preprocessing time of perfect hashing was due to using linked list when building the structure, and could probably be improved by using another faster auxiliary structure such as hashing in the implementation.

Two important requirements for the structures were discovered: lookup speed and iteration speed of a data structure are important for fast preprocessing, in addition to fast insertion time. Lookup speed is needed when building the trie, and iteration speed while doing the breadth-first search. Of course, lookup speed is also important in the search phase. These requirements seem to be important when using a large alphabet such as Unicode.

From our results it can also be deduced that the double-array structure should be the fastest representation for up to 100 patterns, as Aoe has concluded that double-array is slightly faster than triple-array. It would be interesting to see if the preprocessing time for double-array slows down the same way as triple-array with pattern sets of several hundreds. Testing which data structure would be the best auxiliary structure for perfect hashing would also be interesting. Also, perhaps other forms of perfect hashing would be better suited to pattern matching applications, such as dynamic perfect hashing techniques [9].

References

[1] A. Aho and M. Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] A. Aho, R. Sethi, and Ullman J. *Compilers: Principles, Techniques and Tools.* Addison Wesley, 1986.

[3] J. Aoe. An efficient implementation of static string pattern matching machines. *IEEE Transactions on Software Engineering*, 15(8):1010–1016, 1989.

[4] S. Arikawa and T. Shinohara. A run-time efficient realization of Aho-Corasick pattern matching machines. *New Generation Computing*, 2(2):171–186, 1984.

[5] Bill McGinnis Ministries. King James Bible, 2002. `http://www.patriot.net/users/bmcgin/ministries.html`.

[6] Carter, J. and Wegman, M. Universal classes of hash functions. *Journal of Computer and System Sciences*, 18(2):143–154, 1979.

[7] R. Cichelli. Minimal perfect hash functions made simple. *Communications of the ACM*, 23(1):17–19, 1980.

[8] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition.* The MIT Press, Cambridge (MA), USA, 2001.

[9] M. Dietzfelbinger, A. Karlin, K. Mehlhorn, F. Meyer auf der Heide, H. Rohnert, and R. Tarjan. Dynamic perfect hashing: Upper and lower bounds. In *IEEE Symposium on Foundations of Computer Science*, pages 524–531, 1988.

[10] M. Fredman, J. Komlós, and E. Szemerédi. Storing a sparse table with O(1) worst case access time. *Journal of the Association for Computing Machinery*, 31(3):538–544, 1984.

[11] D. Gusfield. *Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology.* Cambridge University Press, New York, USA, 1997.

[12] B. Iglewicz and D.C. Hoaglin. *How To Detect and Handle Outliers.* ASQC Quality Press, Milwaukee (WI), USA, 1993.

[13] R. Sprungnoli. Perfect hashing functions: a single probe retrieving method for static sets. *Communications of the ACM*, 20(11):841–850, 1977.

[14] Sun Developer Network. *Bug 4669519*, 2002. `http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=4669519`.

[15] R. Tarjan and Yao A. Storing a sparse table. *Communications of the ACM*, 22(11):606–611, 1979.