Modified Binary Searching for Static Tables

Donatella Merlini, Renzo Sprugnoli, M. Cecilia Verri

Dipartimento di Sistemi e Informatica - Università di Firenze
Via C. Lombroso 6/17 - 50134 Firenze - Italy
{merlini, sprugnoli, verri}@dsi.unifi.it

Abstract. We consider a new method to retrieve keys in a static table. The keys of the table are stored in such a way that a binary search can be performed more efficently. An analysis of the method is performed and empirical evidence is given that it actually works.

Keywords: binary searching, static dictionary problem.

1 Introduction

The present paper was motivated by some obvious observations concerning binary searching when applied to static sets of strings; by following the directions indicated by these observations, we obtained some improvements that reduce execution time for binary searching of $30-50\%$, and also more for large tables.

Let us consider the set of the zodiac names

$$S = \{\mathtt{capricorn}, \mathtt{acquarius}, \mathtt{pisces}, \mathtt{aries}, \mathtt{taurus}, \mathtt{gemini}, \mathtt{cancer}, \mathtt{leo},$$
$$\mathtt{virgo}, \mathtt{libra}, \mathtt{scorpio}, \mathtt{sagittarius}\}$$

and consider the problem of finding out if identifier x belongs or does not belong to S. To do this, we can build a lexicographically ordered table T with the names in S and then use binary searching to establish whether $x \in S$, or not (see Table 1(a)). What is fine in binary searching is that, if the program is properly realized, the number of character comparisons is taken to a minimum. For instance, the case of `aries` is as follows: 1 comparison (`a` against `l`) is used to compare `aries` and the median element `leo`; 1 comparison (`a` against `c`) is used to distinguish `aries` from `cancer`; 2 comparisons (`ar` against `ac`) are used for `aries` against `acquarius`. After that, the searched string is compared to the whole item `aries`.

The problem here is that we do not know in advance how many characters have to be used in each string-to-string comparison; therefore, every string comparison requires a (comparatively) high number of housekeeping instructions, which override the small number of character comparisons and make the matching procedure relatively time consuming.

The aim of this paper is to show that the housekeeping instructions can be almost eliminated by arranging the elements in a suitable way. First of all we have to determine the minimal number of characters able to distinguish the elements

Table 1. Two table's structures for zodiac names

	(a)	**(b): s=1**	
1	acquarius	1	$\boxed{\text{a}}$cquarius
2	aries	1	$\boxed{\text{c}}$ancer
3	cancer	2	l$\boxed{\text{e}}$o
4	capricorn	1	$\boxed{\text{l}}$ibra
5	gemini	1	$\boxed{\text{p}}$isces
6	leo	5	capr$\boxed{\text{i}}$corn
7	libra	1	$\boxed{\text{s}}$agittarius
8	pisces	1	$\boxed{\text{t}}$aurus
9	sagittarius	2	s$\boxed{\text{c}}$orpio
10	scorpio	1	$\boxed{\text{a}}$ries
11	taurus	1	$\boxed{\text{g}}$emini
12	virgo	1	$\boxed{\text{v}}$irgo

of the set: in the previous example the first three characters are sufficient to univocally distinguish strings in S and this is not casual. In fact, if we have a set S containing n strings on the alphabet Σ with $|\Sigma| = \rho$, what we are expecting is that $\lceil \log_\rho n \rceil$ characters actually distinguish the elements in S. The problem is that in general these characters can be in different positions and we have to determine, string by string, where they are. What we wish to show here is that for any given set S, it is relatively easy to find out whether the elements in S can be distinguished using a small amount of characters. In that case, we are then able to organize the set S in a table T such that a modified binary searching can be performed with a minimum quantity of housekeeping instructions, thus improving the traditional searching procedure. Presently, we are also investigating how modified binary search compares to the method proposed by Bentley and Sedgewick [1]. To be more precise, in Section 2 we show that a pre-processing algorithm exists which determines (if any) the optimal organization of the table for the given set S; the time for this pre-processing phase is in the order of $O(un \log n)$, where u is the length of the strings in S. Then, in Section 3, we find the probability that the pre-processing algorithm gives a positive result, i.e., it finds the optimal table organization for set S. Finally, in Section 4, an actual implementation of our method is discussed and empirical results are presented showing the improvement achieved over traditional binary searching.

2 The pre-processing for modified binary search

Let us consider a set S of n *elements* or *keys*, each u characters long (some characters may be blank). We wish to store these elements in a table T in which binary searching can be performed with a minimum quantity of housekeeping instructions. To this purpose we need a vector $V[1 \dots n]$ containing, for every $i = 1, 2, \dots, n$, the position from which the comparison, relative to the element

$T[i]$, has to begin. A possible Pascal-like implementation of this Modified Binary Searching (MBS) method can be the following:

```
function MBS(str:string): integer;
var a, b, k, p: integer; found:boolean;
begin a := 1; b := n; found:=false;
    while a ≤ b do
        k := ⌊(a + b)/2⌋;
        p:=Compare(str, T[k], V[k], s);
        if p = 0
            then found:=true; a := b + 1
            else if p < 0
                then b := k − 1
                else a := k + 1 fi fi od;
    if found and T[k] = str then MBS:=k else MBS:=0 fi
end;
```

Here, the function *Compare(A,B,i,s)* compares the two strings A and B according to s characters beginning from position i; the result is -1, 0 or 1 according to the fact that the s characters in A are lexicographically smaller, equal or greater than the corresponding characters in B.

The procedure, we are going to describe to build table T and vector V, will be called *PerfectDivision(S,n,s)*, where S and n are as specified above and s is the number of consecutive characters used to distinguish the various elements (in most cases $1 \leq s \leq 4$ is sufficient). It returns a table T of dimension n, the *optimal table*, containing S's elements and a vector V of dimension n such that, for all i from 1 to n, $V[i]$ indicates the position from which the comparison has to begin. More precisely, the elements in table T are arranged in such a way that, if we binary search for the element $x = x_1 \ldots x_u$ and compare it with element $T[i] = T[i]_1 \ldots T[i]_u$ we have only to compare characters $x_{V[i]} \ldots x_{V[i]+s-1}$ with characters $T[i]_{V[i]} \ldots T[i]_{V[i]+s-1}$.

The procedure consists of two main steps, the first of which tries to find out the element in S which determines the subdivision of the keys into two subtables. The second step recursively calls *PerfectDivision* to actually construct the two subtables.

We begin by giving the definition of a *selector*:

Definition 1 *Let S be a set containing n u-length strings. The pair $(i, \alpha^{[s]} = a_1a_2\ldots a_s)$, $i \in [1\ldots u-s+1]$, is an $s^{[i]}$-selector for S iff:*

1) $\exists! w \in S : w_iw_{i+1}\ldots w_{i+s-1} = a_1a_2\ldots a_s$;
2) $\exists \lfloor (n-1)/2 \rfloor$ *keys* $y \in S : y_iy_{i+1}\ldots y_{i+s-1} < a_1a_2\ldots a_s$;
3) $\exists \lceil (n-1)/2 \rceil$ *keys* $y \in S : y_iy_{i+1}\ldots y_{i+s-1} > a_1a_2\ldots a_s$.

Theorem 1 *Let S be a set containing n u-length strings. If $n = 1$ or $n = 2$ then S always admits a selector.*

Proof: If $S = \{x\}$ then it admits the $1^{[1]}$-selector $(1, x_1)$. If $S = \{x, y\}$, it admits a $1^{[p]}$-selector (p, z_p) with $p = \min\{i \in [1 \ldots u] : x_i \neq y_i\}$ and $z_p = \min\{x_p, y_p\}$ (such a p exists since x and y are distinct elements). We observe that if S admits a $1^{[p]}$-selector then it also admits an $s^{[q]}$-selector $\forall p + 1 - s \leq q \leq p$. ■

More in general, for $n > 2$ we can give an integer function *FindSelector(***var** *T, a, b, s)* to determine whether or not the elements from position a to position b in a table T containing $b - a + 1$ u-length strings, admit an s-selector; after a call to this function, if the selector has been found and $k = \lfloor (a + b)/2 \rfloor$ is the index of the median element, then the elements in T are arranged in such a way that:

1. the median element $T[k]$ contains $\alpha^{[s]}$ starting at position i;
2. the elements $T[a], \ldots, T[k-1]$ are less than $T[k]$ when we compare the characters from position i to position $i + s - 1$ with $\alpha^{[s]}$;
3. the elements $T[k+1], \ldots, T[b]$ are greater than $T[k]$ when we compare the characters from position i to position $i + s - 1$ with $\alpha^{[s]}$.

The function returns the position $i = sel$ at which the selector begins, if any, 0 otherwise. This value is stored in vector V from procedure *PerfectDivision* after *FindSelector* is called.

FindSelector calls, in turn, a procedure, *Select(***var** *T, a, b, sel, s)*, and a boolean function, *Equal(x, y, sel, s)*, which operate in the following way:

- procedure *Select* arranges the elements in table T, from position a to position b, by comparing, for each element, the s characters beginning at position sel; the arrangement is such that the three conditions 1., 2. and 3. above are satisfied. As is well known, the selection problem to find the median element can be solved in linear time; however, in our algorithm we decided to use a heapsort procedure to completely sort the elements in T. This is slower, but safer[1] than existing selection algorithms, and the ordering could be used at later stages;
- function *Equal* compares the s characters of the strings x, y, beginning at position sel, and returns *true* if the compared characters are equal, *false* otherwise. It corresponds to *Compare(x,y,sel,s)*$= 0$ above.

Function *FindSelector(***var** *T,a,b,s)*: **integer**;
var sel, i: **integer**;
begin
$sel := 1$;
if $b - a \geq 2$ **then**
  FindSelector:=0;
  $i := \lfloor (a + b)/2 \rfloor$;
  while $sel \leq (u - s + 1)$ **do**

[1] The algorithm of Rivest and Tarjan [3, vol. 3, pag. 216], can only be applied to large sets; the algorithm of selection by tail recursion [2, pag. 230] is linear on the average but is quadratic in the worst case.

```
        Select(T, a, b, sel, s);
        if Equal(T[i], T[i+1], sel, s) or Equal(T[i], T[i-1], sel, s) then
              sel := sel + 1
        else
              FindSelector:=sel;
              sel := u + 1 {to force exit from the loop}
        fi
  od
else if b - a = 1 then
  while sel ≤ u do
        if T[a][sel] = T[b][sel] then
              sel := sel + 1
        else
              FindSelector:= min{sel, u - s + 1};
              sel := u + 1 {to force exit from the loop}
        fi
  od
  FindSelector:=sel
else
  FindSelector:=sel
fi
end {FindSelector};

Procedure PerfectDivision(var T,a,b,s);
var k, i: integer;
begin
k :=FindSelector(T, a, b, s);
if k = 0 then
        fail
else
        i := ⌊(a + b)/2⌋;
        V[i] := k;
        if a < i then PerfectDivision(T, a, i-1, s) fi;
        if i < b then PerfectDivision(T, i+1, b, s) fi
fi
end {PerfectDivision};
```

If we want to apply *PerfectDivision* to a set S of n u-length strings, we first store S's elements in a table T, hence call *PerfectDivision(T,1,n,s)* by choosing an appropriate value for s. In practice, we can start with $s = 1$ and then increase its value if the procedure fails. In this case, if $s = s_0 > 1$ the procedure should be executed s_0 times and this increases the complexity of the pre-processing phase.

This procedure applied to the set of zodiac names by using $s = 1$ returns the table T and the vector V as in Table 1(b). Let us take into consideration again the case of `aries`: we start with the median element and since $V[6] = 5$ we have to compare the fifth character of `aries`, `s`, with the fifth character of `capricorn`, i; s>i and we proceed with the new median element `scorpio`. Since $V[9] = 2$ we compare `r` with `c`; in the next step the median element is `gemini` and we compare the first character of `aries` with the first one of `gemini`. After that, the searched string is compared to the whole string `aries` and the procedure ends with success.

For the set of the first twenty numbers' names the procedure fails by using $s = 1$ and returns the situation depicted in Table 2 by choosing $s = 2$.

Table 2. Table T and vector V when $s = 2$ for the first twenty numbers' names

	V	T
1	1	$\boxed{\text{te}}$n
2	3	th$\boxed{\text{re}}$e
3	5	seve$\boxed{\text{n}}$
4	5	seve$\boxed{\text{nt}}$een
5	2	n$\boxed{\text{in}}$e
6	1	$\boxed{\text{fi}}$ve
7	1	$\boxed{\text{on}}$e
8	1	$\boxed{\text{si}}$x
9	1	$\boxed{\text{tw}}$o
10	4	nin$\boxed{\text{et}}$een
11	1	$\boxed{\text{th}}$irteen
12	2	f$\boxed{\text{if}}$teen
13	5	eigh$\boxed{\text{t}}$
14	5	eigh$\boxed{\text{te}}$en
15	2	s$\boxed{\text{ix}}$teen
16	3	tw$\boxed{\text{el}}$ve
17	3	tw$\boxed{\text{en}}$ty
18	3	el$\boxed{\text{ev}}$en
19	4	fou$\boxed{\text{r}}$
20	4	fou$\boxed{\text{rt}}$een

The complexity of *PerfectDivision* is given by the following Theorem:

Theorem 2 *If S is a set of n words having length u and we set $d = u - s + 1$, then the complexity of the procedure PerfectDivision is in the order of $O(dn \log_2^2 n)$ if we use Heapsort as a selection algorithm, and $O(dn \log_2 n)$ on the average if we use a linear selection algorithm.*

Proof: The most expensive part of *PerfectDivision* is the selection phase, while all the rest is performed in constant time. If we use HeapSort as an in-place sorting procedure, the time is $An \log_2 n$, where $A \sim 2 \ln 2$. This procedure is executed at most d times. Let C_n be the total time for finding a complete division for our set of n elements; clearly we have:

$$C_n \sim Adn \log_2 n + 2C_{n/2}.$$

There are general methods, see [4], to find a valuation for C_n; however, let us suppose $n = 2^k$, for some k, and set $c_k = C_{2^k}$; then we have $c_k = Ad2^k k + 2c_{k-1}$. By unfolding this recurrence, we find:

$$c_k = Ad2^k k + 2Ad2^{k-1}(k-1) + 4Ad2^{k-2}(k-2) + \ldots =$$

$$= Ad2^k(k + (k-1) + \cdots + 1) = Ad2^{k-1}k(k+1).$$

Returning to C_n :

$$C_n \approx Ad\frac{n}{2} \log_2 n(\log_2 n + 1) = O(dn(\log_2 n)^2).$$

Obviously, if we use a linear selection algorithm the time is reduced by a factor $O(\log_2 n)$. ■

Since $u \geq d$, this theorem justifies our statement in the Introduction. In many occasions, when n is large, we have $d \approx \log_\rho n = O(\log_2 n)$ and therefore the complexity of the preprocessing phase becomes $O(n(\log_2 n)^2)$

The next section is devoted to the problem of evaluating the probability that a perfect subdivision for the table T exists.

3 The analysis

We now perform an analysis of the pre-processing phase of modified binary searching. Let $\Sigma = \{\sigma_1, \sigma_2, \ldots, \sigma_\rho\}$ be a finite alphabet and let its elements be ordered in some way, for instance $\sigma_1 < \sigma_2 < \ldots < \sigma_\rho$. In real situations, Σ can be the Latin alphabet, $\rho = 26$ or $\rho = 27$ (if we also consider the space) and its ordering is the usual lexicographical order; so, on a computer, Σ is a subset of ASCII codes and if we have $s = 3$ then we should consider triples of letters. We can now abstract and define $A = \Sigma^s$, for the specific value of s, with the order induced by the order in Σ. Then we obscure s and set $A = \{a_1, a_2, \ldots, a_r\}$, where $r = \rho^s$. Finally, we observe that if S is any set of strings, $S \subset \Sigma^u$, for a given starting index the subwords of the words in S beginning at that position and composed by s consecutive characters are a multiset $\bar{S}$ over A. What we are looking for, is a suitable division of this multiset.

Therefore, let us consider the multiset $\bar{S}$, with $|\bar{S}| = n$, over our abstract alphabet A. Our problem is to find the probability that an element $a_m \in A$ exists such that: i) $a_m \in \bar{S}$, but has no duplicate in $\bar{S}$; ii) there are exactly $\lfloor (n+1)/2 \rfloor - 1$ elements $a_i \in \bar{S}$ such that $a_i < a_m$ (i.e., these elements constitute a multiset over $\{a_1, a_2, \ldots, a_{m-1}\}$); iii) there are exactly $\lceil (n+1)/2 \rceil$ elements a_j in $\bar{S}$ such that $a_j > a_m$ (i.e., these elements constitute a multiset over $\{a_{m+1}, \ldots, a_r\}$). In a more general way we can define the following three (m,p)-*separation conditions* for a multiset $\bar{S}$ as above, relative to an element $a_m \in A$ (see also Definition 1):

i) $a_m \in \bar{S}$ and has no duplicate in $\bar{S}$;
ii) there are $(p-1)$ elements in $\bar{S}$ preceding a_m;
iii) there are $(n-p)$ elements in $\bar{S}$ following a_m.

Our first important result is the following:

Theorem 3 *Let $\pi(n,p,r,m)$ be the probability that a multiset $\bar{S}$ on A ($\bar{S}$ and A as above) contains a specific element a_m for which the (m,p)-separation conditions are satisfied; then*

$$\pi(n,p,r,m) = \frac{p}{r^n}\binom{n}{p}(m-1)^{p-1}(r-m)^{n-p}.$$

Proof: Let us count the total number $P(n,p,r,m)$ of the multisets satisfying the three (m,p)-separation conditions for a given element $a_m \in A$ $(1 \leq m \leq r)$. If we imagine that the elements in $\bar{S}$ are sorted according to their order sequence in A, the first $p-1$ elements must belong to the subset $\{a_1, a_2, \ldots, a_{m-1}\}$, and therefore $(m-1)^{p-1}$ such submultisets exist. In the same way, the last $n-p$ elements in $\bar{S}$ must belong to $\{a_{m+1}, \ldots, a_r\}$ and therefore $(r-m)^{n-p}$ such submultisets exist. Since every first part can be combined with each second part, and the pth character must equal a_m by hypothesis, there exists a total of $(m-1)^{p-1}(r-m)^{n-p}$ ordered multisets of the type described. The original multisets, however, are not sorted, and the elements may assume any position in $\bar{S}$. The different combinations are counted by a simple trinomial coefficient, which reduces to a binomial coefficient:

$$\binom{n}{p-1, 1, n-p} = \frac{n!}{(p-1)!\,1!\,(n-p)!} = p\frac{n!}{p!\,(n-p)!} = p\binom{n}{p}$$

We conclude that the total number of multisets we are looking for is:

$$P(n,p,r,m) = p\binom{n}{p}(m-1)^{p-1}(r-m)^{n-p}.$$

Finally, the corresponding probability is found by dividing this expression by r^n, the total number of multisets with n elements. ■

An immediate corollary of this result gives us the probability that, given $\bar{S}$, an element a_m exists that satisfies the (m,p)-separation conditions:

Theorem 4 *Given a multiset $\bar{S}$ as above, the probability $\pi(n,p,r)$ that an element $a_m \in A$ exists for which the (m,p)-separation conditions hold true is:*

$$\pi(n,p,r) = \frac{p}{r^n}\binom{n}{p}\sum_{m=1}^{r}(m-1)^{p-1}(r-m)^{n-p}$$

Proof: Obviously, any $a_m \in A$ could satisfy the separation conditions, each one in an independent way of any other. Therefore, the probability is the sum of the single probabilities. ■

The formula of this theorem is not very appealing and does not give an intuitive idea of how the probability $\pi(n,p,r)$ varies with p; in particular, what it is when $p = \lfloor (n+1)/2 \rfloor$, the case we are interested in. In order to arrive at a more intelligible formula, we are going to approximate it by obtaining its asymptotic value. To do that, we must suppose $n < r$, and this hypothesis will be understood in the sequel. In real situations, where $r = |A| = 27^s$, we simply have to choose a selector of s character with:

- $s = 1$ for tables up to $n = 22$ elements;
- $s = 2$ for tables up to $n = 580$ elements;
- $s = 3$ for tables up to $n = 15,700$ elements;
- $s = 4$ for tables up to $n = 425,000$ elements.

These numbers correspond to $n/r \approx 0.8$ if $|A| = 27^s$, that is $n \approx 27^s 0.8$. Obviously, a selector with $s > 4$ can result in a worsening with respect to traditional binary searching. On the other hand, very large tables should reside in secondary storage.

A curious fact is that the probabilities $\pi(n,p,r)$ are almost independent of p; as we are now going to show, the dependence on p can only be appreciated when p is very near to 1 or very near to n.

Theorem 5 *The probability $\pi(n,p,r)$ of the previous theorem has the following asymptotic approximations ($n < r$):*

$$\pi(n,p,r) = \left(1 - \frac{1}{r}\right)^n \left(1 + O\left(\frac{n^3}{r^3}\right)\right) \quad \text{when} \quad 2 < p \leq n-2; \tag{3.1}$$

$$\pi(n,2,r) = \pi(n,n-1,r) = \left(1 - \frac{1}{r}\right)^n \left(1 - \frac{n(n-1)}{12(r-1)^2} + O\left(\frac{n^3}{r^3}\right)\right); \tag{3.2}$$

$$\pi(n,1,r) = \pi(n,n,r) =$$

$$= \left(1 - \frac{1}{r}\right)^n \left(1 - \frac{n}{2(r-1)} - \frac{n(n-1)}{12(r-1)^2} + O\left(\frac{n^3}{r^3}\right)\right). \tag{3.3}$$

Proof: Let us apply the Euler-McLaurin summation formula to approximate the sum $\sum_{m=1}^{r} (m-1)^{p-1}(r-m)^{n-p}$. By writing m as the continuous variable x, we have:

$$\sum_{m=1}^{r} (m-1)^{p-1}(r-m)^{n-p} = \sum_{m=1}^{r-1} (m-1)^{p-1}(r-m)^{n-p} =$$

$$= \int_1^r (x-1)^{p-1}(r-x)^{n-p} dx + B_1 \left[f(x)\right]_1^r + \frac{B_2}{2!} \left[f'(x)\right]_1^r + \cdots$$

Here, $f(x) = (x-1)^{p-1}(r-x)^{n-p}$, and we immediately observe that for $p > 1$ we have $[f(x)]_1^r = 0$. In the same way, the first $(p-1)$ derivatives are 0 and, in order to find suitable approximations for $\pi(n,p,r)$, we are reduced to the three cases $p = 1, p = 2$ and $p > 2$. By symmetry, we also have $\pi(n,p,r) = \pi(n, n-p+1, r)$.

Since $f(x)$ is a simple polynomial in x, the integral can be evaluated as follows:

$$\int_1^r (x-1)^{p-1}(r-x)^{n-p}dx = \int_0^{r-1} y^{p-1}(r-1-y)^{n-p}dy =$$

$$= \int_0^{r-1} y^{p-1}\left(\sum_{k=0}^{n-p}\binom{n-p}{k}(r-1)^k(-y)^{n-p-k}\right)dy =$$

$$= \sum_{k=0}^{n-p}\binom{n-p}{k}(r-1)^k(-1)^{n-p-k}\int_0^{r-1} y^{n-k-1}dy =$$

$$= \sum_{k=0}^{n-p}\binom{n-p}{k}(r-1)^k(-1)^{n-p-k}\frac{(r-1)^{n-k}}{n-k} = (r-1)^n\sum_{k=0}^{n-p}\binom{n-p}{k}\frac{(-1)^k}{p+k} =$$

$$= (r-1)^n\frac{1}{p\binom{n-p+p}{n-p}} = \frac{(r-1)^n}{p}\binom{n}{p}^{-1}$$

The last sum in this derivation is well-known and represents the inverse of a binomial coefficient (see Knuth [3, vol. 1, pag. 71]). Therefore we obtain formula (3.1) for $2 < p < n-2$. For $p = 2$ we have a contribution from the first derivative:

$$[f'(x)]_1^r = \left[(r-x)^{n-2} - (x-1)(n-2)(r-x)^{n-3}\right]_1^r = -(r-1)^{n-2}$$

and therefore we have formula (3.2). Finally, for $p = 1$ we also have a contribution from $[f(x)]_1^r$, and find formula (3.3). ■

Table 3 illustrates the probabilities $\pi(n,p,r)$ for $n = 12$ and $r = 20$. We used a computer program for simulating the problem, and in the table we show the probabilities found by simulation, the exact probabilities given by Theorem 4, and the approximate probabilities as computed by formulas (3.1), (3.2) and (3.3).

Formula (3.1), in the short version $\pi(n,p,r) = (1-1/r)^n$, allows us to obtain an estimate of the probability of finding a division with $p = \lfloor (n+1)/2 \rfloor$ for a given set S of identifiers or words over some language. In general, several positions d of the identifiers are available for a division (function *FindSelector* finds the first one), and this increases the probability of success. Usually, if s is the length of the selector and u the identifiers' length, we have $d = u - s + 1$, and in general:

Theorem 6 *Let $\bar{S}$ be a multiset of identifiers over A, with $|\bar{S}| = n$ and $|A| = r$. If d positions in the elements of $\bar{S}$ are available for division, then the probability*

Table 3. Simulation, exact and approximate probabilities

SIMULATION FOR MODIFIED BINARY SEARCHING

Number of alphabet elements (r): 20
Number of multiset elements (n): 12
Number of simulation trials: 30000

p	simulat.	exact	approx.
1	0.72176667	0.72739722	0.72746538
2	0.51906667	0.52409881	0.52389482
3	0.54113333	0.54015736	0.54036009
4	0.53856667	0.54042599	0.54036009
5	0.53563333	0.54036133	0.54036009
6	0.54176667	0.54035984	0.54036009
7	0.54300000	0.54035984	0.54036009
8	0.54173333	0.54036133	0.54036009
9	0.53923333	0.54042599	0.54036009
10	0.53963333	0.54015736	0.54036009
11	0.52596667	0.52409881	0.52389482
12	0.72970000	0.72739722	0.72746538

that an alement $a_m \in \bar{S}$ exists satisfying the (m,p)-separation conditions with $p = \lfloor (n+1)/2 \rfloor$ is:

$$\sigma(n,r,d) = 1 - (1 - \pi(n,p,r))^d \approx 1 - \left(1 - \left(1 - \frac{1}{r}\right)^n\right)^d . \tag{3.4}$$

This quantity can be approximated by:

$$\sigma(n,r,d) \approx 1 - \left(\frac{n}{r}\right)^d \exp\left(-\frac{d(n-1)}{2r}\right) . \tag{3.5}$$

Proof: We can consider the d positions as independent of each other, and therefore equation (3.4) follows immediately. To prove (3.5) we need some computations. First of all we have:

$$\left(1 - \frac{1}{r}\right)^n = \exp\left(n \ln\left(1 - \frac{1}{r}\right)\right) = \exp\left(n\left(-\frac{1}{r} - \frac{1}{2r^2} - \frac{1}{3r^3} - \cdots\right)\right) =$$

$$= \exp\left(-\frac{n}{r} - \frac{n}{2r^2} - \frac{n}{3r^3} - \cdots\right) = e^{-n/r} e^{-n/2r^2} e^{-n/3r^3} \cdots$$

By expanding the exponentials, we find:

$$1 - \left(1 - \frac{1}{r}\right)^n \approx 1 - e^{-n/r} e^{-n/2r^2} e^{-n/3r^3} \cdots =$$

$$= 1 - \left(1 - \frac{n}{r} + \frac{n^2}{2r^2} - \frac{n^3}{6r^3} + \cdots\right)\left(1 - \frac{n}{2r^2} + \cdots\right)\left(1 - \frac{n}{3r^3} + \cdots\right) =$$
$$= \frac{n}{r}\left(1 - \frac{n-1}{2r} + \frac{n^2 - 3n + 2}{6r^2} + O\left(\frac{n^3}{r^3}\right)\right).$$

This formula can be written as:

$$1 - \left(1 - \frac{1}{r}\right)^n = \frac{n}{r}\exp\left(-\frac{n-1}{2r}\right)\left(1 + O\left(\frac{n^2}{r^2}\right)\right)$$

and therefore:

$$\left(1 - \left(1 - \frac{1}{r}\right)^n\right)^d \approx \left(\frac{n}{r}\right)^d \exp\left(-\frac{d(n-1)}{2r}\right)$$

which immediately gives equation (3.5). ■

Having found a division for $\bar{S}$, we are not yet finished, because the procedure has to be recursively applied to the subsets obtained from $\bar{S}$ (see procedure *Perfect-Division*). Since $n/2$ is the approximate size of these two subsets, the probability of finding a division for each of them is:

$$\sigma\left(\frac{n}{2}, r, d\right) \approx 1 - \left(\frac{n}{2r}\right)^d \exp\left(-\frac{d(n-2)}{4r}\right);$$

in fact, except that in very particular cases, the number of possible positions, at which division can take place, is not diminished. The joint probability that both subsets can be divided is $\sigma(n/2, r, d)^2$ and the probability of obtaining a complete division of $\bar{S}$, so that modified binary searching is applicable, is:

$$\tau(n, r, d) = \sigma(n, r, d)\sigma(n/2, r, d)^2\sigma(n/4, r, d)^4 \cdots,$$

the product extended up to $\sigma(n/2^k, r, d)^{2^k}$ such that $n/2^k \geq 3$. In fact, as observed in Theorem 1, a division for tables with 1 or 2 elements is always possible (i.e., $\sigma = 1$).

The probability $\tau(n, r, d)$ is the quantity we are mainly interested in. Obviously, $\tau(n, r, d) < \sigma(n, r, d)$ but, fortunately, we can show that these probabilities are almost equal, at least in most important cases.

Theorem 7 *If $\sigma(n, r, d)$ is sufficiently near to 1, then $\tau(n, r, d) \approx \sigma(n, r, d)$.*

Proof: First of all, let us develop $\tau(n, r, d)$:

$$\tau(n, r, d) \approx$$
$$\approx \left(1 - \left(\frac{n}{r}\right)^d \exp\left(-\frac{d(n-1)}{2r}\right)\right)\left(1 - 2\left(\frac{n}{2r}\right)^d \exp\left(-\frac{d(n-2)}{4r}\right)\right)\cdots \approx$$
$$\approx 1 - \left(\frac{n}{r}\right)^d \exp\left(-\frac{d(n-1)}{2r}\right) - 2\left(\frac{n}{2r}\right)^d \exp\left(-\frac{d(n-2)}{4r}\right) -$$

$$-4\left(\frac{n}{4r}\right)^d \exp\left(-\frac{d(n-4)}{8r}\right) - \cdots$$

What we now wish to show is that the first term (after 1) dominates all the others, and therefore $\tau(n,r,d) \approx \sigma(n,r,d)$. Let us consider the ratio between the first and the second term after the 1 :

$$\left(\frac{n}{r}\right)^d \exp\left(-\frac{d(n-1)}{2r}\right) \frac{1}{2}\left(\frac{2r}{n}\right)^d \exp\left(\frac{d(n-2)}{4r}\right) =$$

$$= 2^{d-1} \exp\left(\frac{dn - 2d - 2dn + 2d}{4r}\right) = 2^{d-1} \exp\left(-\frac{dn}{4r}\right).$$

This quantity shows that the first term is much larger than the second, as claimed. ■

4 Experimental results

In order to verify the effectiveness of our modified binary searching method, we devised some experiments comparing traditional and modified binary search programs. We used Pascal as a programming language, well aware that this might not be the best choice. However, our aim was also to show how better is our method when realised without any particular trick in a high level language. When the table is large, we need two or more characters for the selector s; our implementation obviously performs one-character-at-a-time comparisons; since the characters to be compared against s are consecutive, a machine language realization would instead perform a multi-byte loading and comparing, further reducing execution time. Presently, we are developing a C version of our programs to compare them to the most sophisticated realisations of traditional binary searching and to other, more recent approaches to string retrieval, such as the method of Bentley and Sedgewick [1].

We used the following program for traditional binary searching:

```
function BS (str : string) : integer;
var a, b, k : integer; found : boolean;
begin
    a := 1; b := n; found := false;
    while a<=b do begin
        k:=(a+b) div 2;
        if str = T[k]
            then begin a := b+1; found := true end
        else if str < T[k] then b:=k-1 else a:=k+1
    end;
    if found then BS := k else BS := 0
end;
```

For the modified binary searching, the program is:

```
function MBS1 (str : string): integer;
var a, b, j, k : integer; found : boolean;
begin
    a := 1; b := n; found := false;
    while a <= b do begin
        k := (a+b) div 2; j := V[k];
        if str[j] = T[k,j]
            then begin a := b+1; found := true end
            else if str[j] < T[k,j]
                then b := k - 1
                else a := k + 1 end;
    if found and (str = T[k])
        then MBS1 := k else MBS1 := 0
end;
```

This program is to be used when the selector length is 1; for a selector length of 2 we simply perform two cascade **if**'s, and three for a selector length of 3.

We performed 10 blocks of 20,000 searches of all the strings in a table, randomly chosen in a dictionary of 1,524 English words. For small tables (selector length equal to 1) we obtained the average times in the first part of Table 4 (the time unit is inessential; only relative times are of importance). For larger tables with selector length equal to 2 or 3 we obtained the times in parts two and three of Table 4.

5 Conclusions

We have considered a variant of binary searching, which avoids most of the housekeeping instructions related to string comparisons. This requires a suitable pre-processing phase for the table to be searched, and therefore only applies to the static case. What we have shown is:

1. The variant is considerably faster than traditional binary searching; we give empirical evidence of this fact, by comparing actual programs performing both kinds of binary searching.
2. The pre-processing phase is fast, because it runs in time $O(un \log n)$, and produces with a very high probability the optimal arrangement of the table elements. In any case, an almost optimal arrangement can always be found.

It would be interesting to compare our method with searching in more complex structures than tables, as for example tries (see, e.g., [3, vol. 3]). Tries allow us to look for a string in a table by only using some of its characters. This is analogous to what we are doing with modified binary searching, which can be seen as a sort of specified trie searching. Besides, traditional trie searching is performed quickly and without any pre-processing, even for dynamic sets of keys. However,

Table 4. Times for selector of length 1, 2 and 3

n	MBS	BS	gain (%)
10	29.7	56.2	47.2
11	33.9	63.6	46.7
12	36.3	70.9	48.8
13	38.9	80.4	51.6
14	44.5	86.3	48.4
15	47.1	93.8	49.8
16	52.0	103.1	49.6
17	54.4	113.7	52.2
18	58.3	123.1	52.6
20	65.9	139.6	52.8
22	73.3	159.5	54.0
24	81.3	178.8	54.5
26	87.3	197.8	55.9
28	94.4	215.9	56.3
30	101.6	232.8	56.4
32	110.0	254.8	56.8

n	MBS	BS	gain (%)
10	33.0	54.8	39.8
20	69.9	141.1	50.5
30	108.3	234.4	53.8
40	150.5	345.4	56.4
60	235.1	568.5	58.6
80	326.8	825.0	60.4
100	421.8	1084.2	61.1
120	519.6	1340.6	61.2
140	619.5	1625.3	61.9
160	713.4	1919.7	62.8
180	806.5	2214.3	63.6
200	913.3	2510.2	63.6
225	1032.6	2877.0	64.1
250	1146.3	3248.7	64.7
275	1263.1	3648.8	65.4
300	1408.8	4060.6	65.3

n	MBS	BS	gain (%)
50	100.0	228.4	56.2
100	222.8	541.8	58.9
150	346.0	887.6	61.0
200	472.4	1256.6	62.4
250	607.6	1625.4	62.6
300	751.2	2035.4	63.1
350	883.4	2452.8	64.0
400	1032.8	2864.8	63.9
450	1151.4	3283.4	64.9
500	1310.4	3702.0	64.6
550	1447.8	4147.0	65.1
600	1592.8	4610.2	65.5
700	1874.0	5534.2	66.1
800	2202.6	6461.4	65.9
900	2491.4	7377.8	66.2
1000	2760.6	8303.8	66.8

when we deal with static sets, our method reduces the length of the strings to be compared and assures perfectly balanced structure, thus minimizing the average searching time.

Finally, in our opinion, an important aspect of our method is that it can be efficiently realized in a high level language; as is well-known, this is not always possible for other kinds of fast retrieval methods for static tables, such as perfect hashing. This makes our modified binary searching procedure attractive for actual implementation in real systems.

Acknowledgments

We wish to thank the anonymous referee for his useful suggestions to our paper.

References

1. J.L. Bentley, R. Sedgewick: Fast Algorithms for Sorting and Searching Strings. Proceedings of 8th Annual ACM-SIAM Symp. On Discrete Algorithms. (1997) 360–369
2. G. H. Gonnet, R. Baeza-Yates: Algorithms and Data Structures, 2nd edition. Addison Wesley (1991)
3. D. E. Knuth: The Art of Computer Programming. Vol. 1-3. Addison-Wesley (1973)
4. R. Sedgewick, P. Flajolet: An Introduction to the Analysis of Algorithms. Addison-Wesley (1996)