Searching: An Algorithmic Tour

Ricardo Baeza-Yates
Dept. de Ciencias de la Computación
Universidad de Chile
Blanco Encalada 2120
Santiago, CHILE
E-mail: rbaeza@dcc.uchile.cl

Final draft, November 1995

Abstract

Searching is one of the main computer applications (also in real life!), being databases the best example. In this tutorial we cover many of the concepts involved when designing a search algorithm, and how different assumptions change the classical solutions, exploring several classes of searching algorithms, depending on the domain of the search, the cost model, and other aspects; giving lower and upper bounds for several examples. The steps involved in a solution are the definition of a model, followed by the design and analysis of the algorithm, with possibly an optimality proof.

The best theory is inspired by practice,
the best practice is inspired by theory
Donald Knuth [Knu91].

1 Introduction

When we write an algorithm to solve a problem, we implicitly or explicitly have a model of the problem, a cost function associated to inputs and operations, and so on. To analyze the algorithm we usually study its worst case behavior and/or its average case behavior, analytically and/or experimentally. In some cases the worst case is not very interesting in practice (for example in text searching), so the average case should be used. Average case analysis implies having a probability distribution associated with the input of the problem. In most cases it is difficult to have a reasonable but realistic and simple distribution of the input. In other cases, the realistic model could be very simple, but still the analysis is complex due to the algorithm itself. In both cases, experimental analysis should be used to either corroborate the assumptions of our model or to obtain intuition about analytical solutions.

The above shows the importance of theory and practice. They should go together, helping each other to design better algorithms. On the other hand, finding lower bounds for a given model and problem also helps in improving or developing new algorithms. In this tutorial we stress the connection between design and analysis through some real and some toy problems. At the same time we introduce recent types of algorithms such as on-line or randomized algorithms. For this task we have chosen searching algorithms because of their importance, but also because of their variety. We address uniform and non-uniform access costs, bounded resources, partial information, etc.

To express all the cases involved, we introduce a simple taxonomy to classify our examples in Section 2. Later we use that taxonomy to guide the organization of the following sections. Parts of this paper are unpublished work, for example Section 4 and the analysis of Section 7.2. Other results are properly referenced.

2 Framework

Suppose that we want to search a query q in a set of elements X. To completely define this problem, we need more information. As we assume or know more information, better algorithms can be designed, although they will be less general. We can classify the several variants of this searching problem using the following criteria:

- Set characteristics:
  - atomic vs. complex elements (dimensions, components)
  - finite (size n) or infinite
  - static vs. dynamic
  - with a linear, partial or no ordering.
- Algorithm power:
  - on-line vs. off-line (preprocessing time allowed)
  - sequential vs. parallel (computer model needed)
  - deterministic vs. randomized
  - restrictions on the number of questions (bounded resources)
  - type of answers (probabilistic, bounded number of lies)
  - bounded memory
- Searching model:
  - exact match
  - approximate match (distance function needed)
  - closest match or all matches at a certain distance
- Analytic cost model:
  - time and/or space
  - cost model (cost of each query)
  - worst-case

– average-case (probability distribution of the input needed)

– expected worst-case (as above)

– optimality (lower bound under the model used)

3 Searching Atomic Objects

3.1 Searching a Set of Integers

Consider the simplest case: a set of n integers. This a static set of atomic elements that has a total ordering. We can make it dynamic by using linked lists or search trees, but because that is a well studied case, we refer the reader to any data structures textbook or handbook, for example [GBY91].

The problem is: given a query element, find whether it belongs to the set and possibly where in the set is. If the numbers are given one by one (this is called an on-line problem) the obvious solution is to use sequential search. That is, we compare every element, and in the worst case we need n comparisons (either is the last element or it is not present). Under the traditional RAM model, this algorithm is optimal.

We can improve the worst case of sequential search in a probabilistic sense, if the element belongs to the set and we have all the elements in advance (off-line case). Consider the following *randomized* algorithm. We flip a coin. If it is a head, we search the set from 1 to n. Otherwise, from n to 1. The worst case for each possibility is n comparisons. However, we have two algorithms and not only one. Suppose that the element we are looking for is in position i and that the coin is fair (that is, the probability of a head or a tail is the same). So, the number of comparisons to find the element is i if it is a head, or $n - i + 1$ if it is a tail. So, averaging over both algorithms (note that we are not averaging over all possible inputs), the expected worst case is

$$\frac{1}{2} \times i + \frac{1}{2} \times (n - i + 1) = \frac{n+1}{2}$$

which is independent of where the element is! This is better than n. In other words, our adversary places the element in the middle position because it does not know which algorithm will be used.

However, in the off-line case we can search faster if we allow some time to preprocess the set. Certainly, if we sort the set (using $O(n \log n)$ comparisons in the worst case) we can use the well-known binary search. Using binary search we can solve the problem using at most $\lfloor \log_2 n \rfloor + 1$ comparisons. Therefore, if we do many searches we can amortize the cost of the initial sorting. Binary search is optimal for the RAM comparison model in the worst and the average case. However, by assuming more information about the set or changing the model, we can improve the average or the worst case.

Suppose that we assume that the distribution of the n integers is uniform over a fixed range of $M >> n$ integers. Then, instead of using binary search to divide the set, we can linearly interpolate what should be the position of the searched element with respect to smallest and largest element. In this way, on average, it is possible to prove that $O(\log \log n)$ comparisons are needed if the element is in the set. The proof is quite involved and mathematical [PIA78, GG86], but there are some variations of interpolation search that have a simpler analysis. A first intuition about the complexity could be that if we could do a binary search on the elements on the path of a binary search, the number of comparisons is $O(\log\log n)$. However this is not possible because we don't know the path a priori, and the elements in the path are not necessarily sorted. But we can split this path by levels in the tree. The main fact behind the complexity of $O(\log\log n)$ is that when we do the first interpolation, with very high probability the searched element is at distance $O(\sqrt{n})$. So, the expected number of comparisons is given by the following recurrence

$$C(n) = a + C(b\sqrt{n}), \quad C(1) = 1$$

which gives $C(n) = O(\log\log n)$. A simple variation is the following, which is called binary interpolation search [PR77]. Imagine that we divide the ordered set in $\sqrt{n}$ segments of size approximately $\sqrt{n}$. Then, we use interpolation search on the $\sqrt{n}+1$ keys that are segment boundaries (including the first and the last key) (see Figure 1, left) to find in what segment is the key. After we know the segment, we apply the same algorithm recursively in it. We can think that we have a $\sqrt{n}$-ary search tree where in each node we use interpolation search to find the right pointer. By a simple probabilistic analysis it is possible to show that on average less than 2.5 comparisons are needed to find in what segment is the key. So, we can use the previous recurrence with $a = 2.5$ and $b = 1$, obtaining less than $2.5 \log_2 \log_2 n$ comparisons on average.

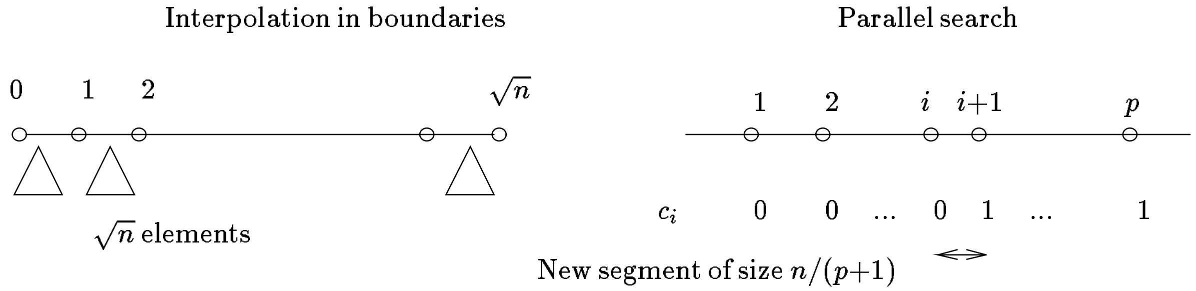

Figure 1: Binary interpolation (left) and parallel binary (right) search.

Suppose now that we change the model by having p processors with a shared memory. That is, we use a parallel RAM or PRAM model. Can we speed up binary search? First, we have to define how the memory is accessed in a concurrent way. The most used model is CREW, that is, concurrent read, but exclusive write (otherwise it is difficult to know the final value of a memory cell after a writing operation). In a CREW PRAM, we can use the following simple parallel binary search. We divide the sorted set in $p+1$ segments (then, there are p keys in the internal segment boundaries). Processor i compares the key to the element stored in the i-th boundary and writes in a variable c_i a 0 if it is greater or a 1 if it is smaller (in case of equality the search ends). All the processors do this in parallel. After this step, there is an index j such that $c_j = 0$ and $c_{j+1} = 1$ (we assume that $c_0 = 0$ and $c_{p+1} = 1$), which indicates in what segment the key should be. Then, processor i compares c_i and c_{i+1} and if they are different writes the new boundaries where the search continues recursively (see Figure 1). This step is also done in parallel (processor 1 and p take care of the extreme cases). When the segment is of size p or less, each processor compares one element and the search ends. Then, the worst case number of key comparisons is given by

$$C(n) = 1 + C\left(\frac{n}{p+1}\right), \quad C(i) = 1 \ (i \leq p)$$

with solution $C(n) = \log_{p+1} n + O(p)$. That is, $C(n) = O(\log n/\log(p+1)$. It is possible to prove that is not possible to do it better. In the PRAM model, the optimal speed up is when the work done for p processors is p times the work of the optimal sequential algorithm. In this case, the total work is $p \log n/\log p$ which is larger than $\log n$. In other words, searching in a sorted set is not a good problem to be solved in parallel. If we restrict the PRAM model to also exclusive reads (EREW), then $C(n) = O(\log n - \log p)$, which is even worse. This is because, at every recursive step, if all the processors cannot read the new segment concurrently, we slow down all the process.

3.2 Searching in an Unbounded Set

In most cases we search in a bounded set. We can also search in an unbounded set. Consider the following game: one person thinks in a positive number and another person has to guess it with questions of the type: *the number x is less than, equal than, or larger than the number that you are thinking?*. This problem was considered in [BY87].

A first obvious solution is to use the sequence 1, 2, ..., n, using n questions. We can do better by using the "gambler" strategy. That is, we use the sequence 1, 2, 4, ..., 2^m, until we have $2^m \geq n$. That is, in the worst case we have, $m = \lfloor \log n \rfloor + 1$. Next, we can use binary search in the interval $2^{m-1} + 1$ to 2^m to search for n, using in the worst case $m - 1$ questions. Hence, the total number of questions is $2m - 1 = 2\lfloor \log n \rfloor + 1$. This algorithm is depicted in Figure 2. That is, only twice a binary search in a finite set of n elements. Can we do better? We can think that what we did is to search the exponent m using sequential search. So, we can use this algorithm, A_1, to search for m using $2\lfloor \log m \rfloor + 1$ questions, and then use binary search, with a total number of questions of $\log n + 2 \log \log n + O(1)$ questions. We could call this algorithm A_2.

In general, we can define algorithm A_k, which uses A_{k-1} to find m and then use binary search. The complexity of such algorithm is

$$\log n + \log \log n + \cdots + \log_{k-1} n + 2 \log_k n + O(1)$$

questions, where $\log_i n$ denotes log applied i times. Of course, if we could knew the value of n in advance, there is an optimal value of k of $O(\log^* n)$[1], because if k is too large, we go too far. However, we do not known n a priori!

3.3 Searching with Bounded Resources

Most of the time we assume that we can perform an unbounded number of questions when searching. However, in many real situations, we search with bounded resources. For example, gasoline when using a car. As an example, we use a variable cost searching problem, initially proposed in [BB80, section 3.2], with some changes, but maintaining the same philosophy. Given a building of n floors and k very hard eggs, we want to answer the following problem: which is the largest floor from where we can throw an egg and the egg does not break?.

Suppose that the answer is floor j. So, we have j floors that will give us a positive answer and $n - j$ floors that give us a negative answer. However, we can only afford up to k negative answers to solve the problem. So, we have a bounded resource: the eggs.

[1] $\log^* n$ is the number of times that we have to apply the log function before we reach a value less than or equal to 0.

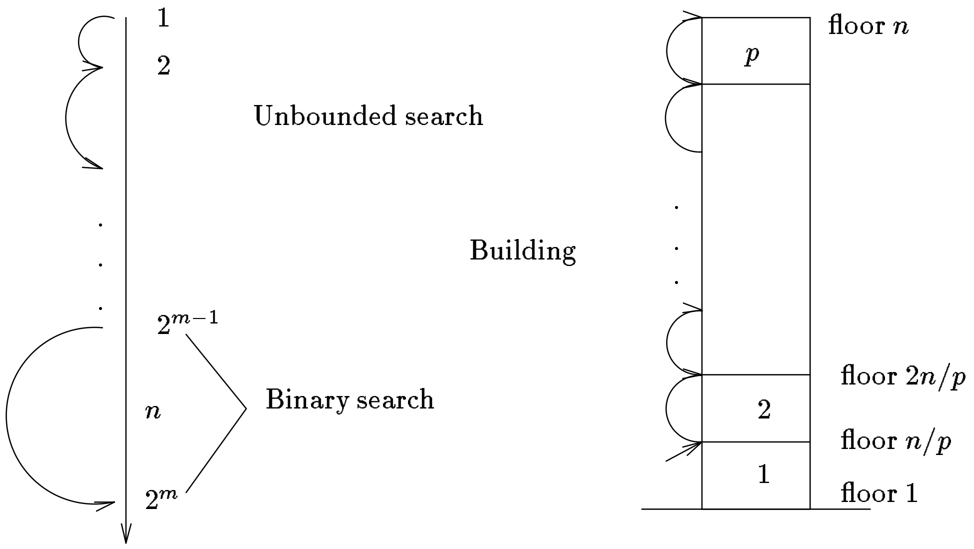

Figure 2: A_1 unbounded search (left) and the egg problem (right).

If we have just one egg, the solution is easy, since we are forced to use sequential search to find j. Any other strategy does not work, because if the egg breaks, we do not solve the problem. If we have many eggs (more precisely $k > \log n$), we can use binary search. In both cases, the solution is optimal in the worst case.

If we have two eggs, the first solution would be to start using binary search with the first egg, and then use the second sequentially in the remaining segment. On the worst case, the first egg breaks in the first throw, giving a $n/2$ throws algorithm. The problem is that both eggs do not perform the same amount of work. We can balance the work by using the following algorithm: the first egg tries sequentially every n/p floors for a chosen p, that is n/p, $2n/p$, etc. When it breaks, the second egg has a segment of approximately p floors to check. See Figure 2. In the worst case the number of floors is p (first egg) plus n/p (second egg). So we have

$$t_2(n) = p + n/p + O(1)$$

Balancing the work, we have $p = n/p$ which implies $p = \sqrt{n}$ giving $t_2 = 2\sqrt{n} + O(1)$. If we use traditional calculus to optimize $t_2(n)$ we achieve the same result. Note also that to succeed, any algorithm has to do sequential search in some segment with the last egg.

We can generalize the above algorithm to k eggs using the partitioning idea recursively. Every egg except the last one partitions the segment in p parts and the last egg uses sequential search. In the worst case, every egg (except the last one) has to perform p throws. The last one does sequential search on a segment of size n/p^{k-1}. So, the total cost is

$$t_k(n) = (k-1)p + \frac{n}{p^{k-1}}$$

Balancing the work for every egg, we must have $p = n/p^{k-1}$, obtaining $p = n^{1/k}$ (same as using calculus!). Then, the final cost is

$$t_k(n) = kn^{1/k}$$

If we consider $k = \log_2 n$, we have

$$t_k(n) = \log_2 n \; 2^{\log_2(n^{1/k})} = \log_2 n \; 2^{\frac{\log_2 n}{k}} = 2\log_2 n$$

which is almost like binary search. In fact, taking care of the partition boundaries, and using an optimal partition, we can save k jumps, which gives the same as binary search. So we have a continuous between sequential and binary search as k grows. In [BB80] it is shown that the optimal partition is related to binomial trees.

4 Searching Non-Uniform Cost Domains

In the traditional RAM model we assume that any memory access has the same cost. However, this is not true if we consider the memory hierarchy of a computer: registers, cache and main memory, secondary storage, etc. As an example of this case, we use the hierarchical memory model introduced in [AACS87]. That is, access to position x is given by a function $f(x)$. The traditional RAM model is when $f(x)$ is a constant function. Based in access times of current devices, possible values are $f(x) = \log x$ or $f(x) = x^\alpha$ with $0 < \alpha \leq 1$. This model has been extended to cover non-atomic accesses (that is, access by blocks) and other variations [ACS87, AV87, AC88].

Given a set of n integers in a hierarchical memory, two problems are discussed. First, given a fixed order (sorted data), what is the optimal worst case search algorithm. Second, what is the optimal ordering (implicit structure) of the data to minimize the worst case search time. This ordering must be described using constant space.

In both cases, we want to have the n elements in n contiguous memory locations starting at some position and only using a constant amount of memory to describe the searching procedure. In our search problem, we consider only successful searches, being the probability of searching each one of the n elements the same.

4.1 Optimal Searching for Sorted Data

Let $C(i,j)$ be the optimal worst case cost to search an element which is between positions i and j of the memory. We can express the optimal worst case cost as

$$C(i,j) = \min_{k=i,\ldots,j} \{f(k) + \max(C(i,k-1), C(k+1,j))\}$$

for $i \geq j$ or 0 otherwise. We are interested in $C(1,n)$. This recurrence can be solved using dynamic programming in $O(n^2)$ time. This problem was considered in [Kni88], where it is shown that for logarithmic or polynomial $f(x)$, the optimal algorithm is $O(f(n)\log n)$. In particular, if $f(x) = x^\alpha$, a lower and upper bound of

$$\frac{n^\alpha \log n}{1+\alpha}$$

for the worst case cost of searching is given in [Kni88].

We can derive immediately the following general bounds:

$$f(n) + f(n-1) \leq C(1,n) \leq f(n/2) + f(3n/4) + \ldots. + f(n) \leq f(n)\log_2 n$$

The upper bound is using binary search, and the lower bound is because we can always be forced to inspect at least the two last elements. That is, there is only a $\log n$ term between both cases. For $f(x)$ a constant, the upper bound is tight (binary search in a RAM). For $f(x) = 2^x$ the lower bound is tight.

The lower bound is tight for any function such that

$$C(n) = f(n-1) + \max(C(n-2), f(n)) = f(n-1) + f(n)$$

That is, first we compare with position $n-1$, and in the worst case, is more expensive (or the same) to inspect position n. Then, we have that the break point is given by

$$f(n) = f(n-2) + f(n-3)$$

with $f(3) > f(2) > f(1) \geq 0$. This recurrence implies that $f(n)$ is at least $O((1.32)^n)$.

The exact complexity for $f(x) = x$ is not known, although we know that is $O(n \log n)$ [Kni88]. The solution does not look well behaved, in the sense that the worst case number of probes does not increase monotonically. Some values are given in Table 1.

n	$C(n)$	$\Delta C(n)$	$\log_2 n$	Optimal Probes	
				Number	Worst Sequence
1	1	1	0	1	1
2	3	2	1	2	2, 1
3	5	2	1	2	2, 3
4	7	2	2	2	3, 4
5	9	2	2	3	4, 2, 3
6	12	3	2	3	5, 3, 4
7	15	3	2	4	6, 4, 2, 3
8	19	4	3	4	7, 5, 3, 4
9	23	4	3	5	8, 6, 4, 2, 3
10	26	3	3	3	7, 9, 10
15	41	3	3	4	12, 8, 10, 11
20	62	5	4	5	17, 13, 9, 11, 12
25	89	6	4	6	22, 18, 14, 10, 12, 13
30	109	4	4	4	25, 29, 27, 28
35	129	4	5	4	28, 32, 34, 35
40	150	5	5	5	33, 25, 29, 31, 32
45	175	5	5	5	38, 30, 34, 36, 37
50	203	6	5	6	43, 35, 27, 31, 33, 34
55	233	6	5	7	48, 40, 32, 24, 28, 30, 31
60	268	7	5	7	53, 45, 37, 29, 33, 35, 36
62	280	5	5	5	53, 59, 57, 55, 56
65	295	5	6	5	56, 62, 60, 58, 59
70	320	5	6	5	59, 67, 63, 65, 66

Table 1: Optimal worst case searching cost for $f(x) = x$.

4.1.1 Optimal Ordering for Worst Case Searching

In this case we can order the elements to minimize the searching cost. A first approach is to store the data as the implicit complete binary search tree induced by a binary search in the sorted data, such that the last level is compacted to the left (*left complete binary tree*). That is, we store the root of the tree in position 1 and in general the children of the element in position i in positions $2i$ and $2i+1$ like in a heap. The worst case search access cost for n of the form $2^k - 1$ is

$$C(n) = f(1) + f(3) + f(7) + \ldots + f(2^k - 1) > f(n) + \log n$$

Then we have the following trivial bounds:

$$f(n) + f(1) \leq C(n) \leq \sum_{i=1}^{\lceil \log_2(n+1) \rceil} f(2^i - 1)$$

where the lower bound is given by the fact that any algorithm can be forced to inspect the last position and another element, and the upper bound is given by the heap already mentioned.

For $f(x) = x$ we have then

$$n + 1 \leq C(n) \leq 2n - O(\log n)$$

and then, the order of both solutions is the same. It is possible to improve this result for a linear cost function, by balancing as much as possible the cost of all the paths in the tree. For this we change the address computation of the children in the left complete binary search tree to: $2j$ and $2j+1$ with $j = 3 \cdot 2^{\lfloor \log i \rfloor} - i - 1$ for position i. In other words, the smallest position of one level gets the children of the biggest position in that level, the second smallest the second biggest, and so on (see Figure 3). For $f(x) = x$ we have

$$C(n) = \frac{5n}{3} + O(1)$$

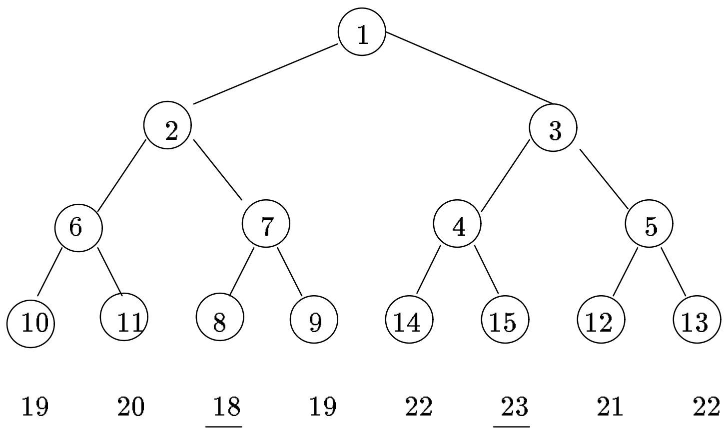

Figure 3: Addressing scheme for balancing all paths.

The optimal search tree does not have to be balanced. One example of this is for $f(x) = 2^x$. For this case, the children of position i are in $i+1$ and $n-i+1$. That is, a left branch of length approximately $n/2$ with one right children per level. The worst case is

$$C(n) = 2 + 2^n$$

and this is optimal. In fact, this is optimal for any function such that

$$f(n) \geq f(2) + \max\{f(n-1), \sum_{i=3}^{\lceil n/2 \rceil} f(i)\}$$

for some $n > n_0$. The first argument of the max implies $f(n) \geq (n-2)f(2)$ and the second argument implies an exponential cost function (a polynomial cost function of $O(n^k)$ gives $O(n^{k+1})$ on the right hand side). Therefore, for any $f(x) = a^x$ $(a > 1)$ we can find n_0 such that the above requirements are satisfied.

Assuming uniform access probabilities, it is not difficult to show that the average case for the above search tree is $O(a^n/n)$ for $f(x) = a^x$. For example, for $f(x) = 2^x$ we have

$$\overline{C}(n) = \frac{2+2^n}{n} + \frac{n-2}{n} 2\overline{C}(n-2)$$

Defining $D_n = n\overline{C}(n)$ we obtain

$$D_n = 2 + 2^n + 2D_{n-2}\ , \quad D(1) = 2, \quad D(2) = 8$$

whose solution is

$$D_n = 2^{n+1} + 2^{(n-4)/2}(7(-1)^n - 5) + 1$$

and then

$$\overline{C}(n) = \frac{2^{n+1}}{n} + O(2^{n/2}/n)$$

In general we may want to put the root in the lower cost position and divide the remaining set of positions such the cost of both subsets is as balanced as possible. We repeat the process in both subsets. This procedure seems to have the same asymptotic cost of the alternating left complete binary tree for a linear cost function.

5 Searching without Information

In this section we also have a non-uniform cost model plus an unbounded domain. In addition the algorithm does not know all the information of the domain and learns about it while searching. All problems considered in this section are of the following form: we are searching for an object in some space under the restriction that for each new "probe" we must pay costs proportional to the distance of the probe position relative to our current probe position and we wish to minimize this cost. This is meant to model the cost in real terms of a robot (or human) searching for an object when the mobile searcher must move about to find the object. Is also the case for many searching problems on secondary memory devices as disk and tapes. These problems also are examples of an on-line algorithms. An algorithm is on-line if receives the information while it is working. An on-line algorithm is called c-competitive if the solution to the problem related to the optimal solution when we have all the information at the beginning (off-line case) is bounded by

$$\frac{Solution(on-line)}{Optimal(off-line)} \leq c$$

To make this concrete, suppose that we are at the origin in the plane and we are searching for a line. Suppose that the line is at distance n steps (we use steps as our metric) away from the origin.

- Given a normal to the line we can find the line in n steps.
- Given the line's distance and slope we can find the line in $3n$ steps.
- Given the line's distance and that the line is horizontal or vertical we can find the line in $3\sqrt{2}n \approx 4.24n$ steps [BYCR88].
- Given the line's distance we can find the line in $(1+\sqrt{3}+7\pi/6)n \approx 6.39n$ steps [Isb57].
- Given the line's slope we can find the line in $9n$ steps [Gal80, BYCR93].
- Given that the line is horizontal or vertical we can find the line in $9\sqrt{2}n$ steps [BYCR88].
- Given nothing at all we can find the line in $13.81n$ steps [BYCR93].

Except for the last two, all of the above results are provably optimal up to lower order terms. These results are representative of the gradation in cost as information about the target decreases. In all cases the competitiveness of the algorithm is given by the coefficient before n.

5.1 Searching for a Point on a Line

Suppose that the robot needs to find some distinguished point on a line. Assume that the point is n steps away along the line and that the robot does not know how far away the point is. What is the minimum number of steps it must make to find the point, as a function of n? For this first and simplest problem, we will spend some time developing the basic ideas and manipulations since they are used several times.

Any algorithm to solve this problem can be described as a function, f, where $f(i)$ is the number of steps it makes to the left (or right) before the i^{th} turn and where the odd terms are the number of steps to the left and the even terms are the number of steps to the right as measured from the origin. That is, starting at the origin, the robot walks $f(1)$ steps to the left, turns and returns to the origin then walks $f(2)$ steps to the right etc. Observe that if the robot is to find the point, then f must be such that

$$f(i) \geq f(i-2)+1 \quad \forall i \geq 1 \text{ where } f(-1)=f(0)=0$$

The optimal function f (up to lower order terms) is given by Linear Spiral Search [BYCR93]: execute cycles of steps where the function determining the number of steps to walk before the i^{th} turn starting from the origin is

$$f(i) = 2^i \quad \forall i \geq 1$$

The total distance walked is $2\sum_{i=1}^{\lfloor \log n \rfloor + 1} 2^i + n$ which is no more than $9n$ steps.

Note that this is a lower bound on the *constant multiple* of the distance walked to the actual distance to the point. In fact there exist algorithms which take no more than $9n - \Theta(\lg n)^i$ steps for any i. In other words, the best possible competitiveness is 9.

5.2 The Average Case for a Point a Bounded Distance Away

Suppose that the robot knows that the point is within n steps and that it is distributed uniformly about the interval of length $2n$ centered on the origin. Then the naive algorithm, go first n steps to one side, and then to the other side, is also the best average case algorithm, with an average distance of $3n/2$. However, if the robot knows that the point is likely to be near the origin, then it might want to turn after a smaller number of steps, since going very far from the origin the probability of finding the point further on is much less than finding it near the origin on the other side. We can show that the optimal average case algorithm for most distributions, including bounded domains, has an *infinite* number of turning points! Intuitively, this happens because there is always a point at which it is better to turn back and look at ranges on the other side of the origin where the probability of finding the point is greater [BYCR93].

5.3 Parallel Searching: a Protocol Problem

Suppose now that we have several robots [BYS94]. Then, we can have several searching models. The simplest one, but not very interesting, is that when a robot finds the goal, it does not care about the other robots. A better model, which we use most of the time, is to assume that the robots have some means of instantaneous contact (say a radio) such that when a robot finds the goal all other robots stop (or go to where the goal is). Yet another model is when the robots do not have a communication device. So, there has to be some kind of protocol such that the robot that finds the goal can find the other robots and direct them to the goal. We include one example for this model.

We consider two measures: the distance travelled by the robots and the time to reach the goal. Without loss of generality, we also assume that all robots are initially placed at the same starting point (origin) and that they travel at the same speed v (if there are different speeds, the worst case will always happen to the slowest robot). We call n/v a unit of time, and we use a normalized speed $v = n$ units of distance/time (remember that n is the possibly unknown distance to the goal).

Consider again the problem of finding some point on a line whose position is unknown to the robot. For two robots the distance travelled is $2n$ and the time necessary to find the goal is 1 unit and more robots do not help. If both robots must reach the goal, the distance increases to $4n$ and the time to 3 units (the robot that finds the goal waits for the other robot).

Let's now have the following variation: both robots cannot communicate. Hence, the robot that finds the goal must find the other robot, tell him, and then both must go back to the goal. To have a solution the robots must agree on a protocol such that they can get in contact in the future. One possibility is to stop and wait at certain intervals before any of them finds the goal (or in other words, they agree to reduce the speed). Suppose that they wait R time/distance units. The robot that finds the goal goes back at full speed until meets the other robot and they return together (see Figure 4). They will meet because one robot is going faster than the other robot.

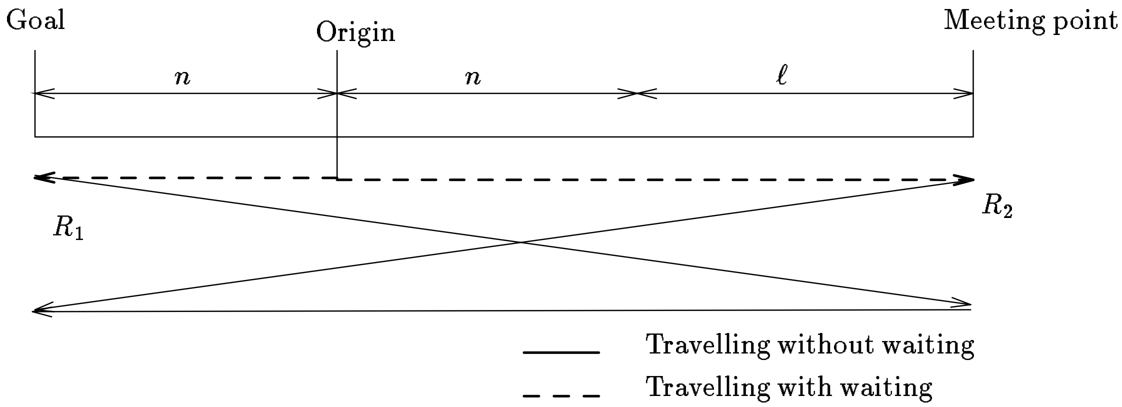

Figure 4: Parallel solution without communication.

After the first robot reaches the goal, let ℓ be the extra distance travelled by the second robot (see Figure 4). We have that ℓ is such that

$$\frac{\ell + 2n}{v} = \ell \left(\frac{1}{v} + \ R \right),$$

where $\frac{\ell+2v}{v}$ is the time of full-speed traversal by the first robot that finds the goal and $\ell(\frac{1}{v} + R)$ is the time spent by the other robot after the goal is found (without it knowing that). Therefore:

$$\ell = \frac{2n}{v\ R}$$

Hence, the total distance is:

$$4n + 2\ell + 2\ell + 4n = 8n \left(1 + \frac{1}{v\ R} \right)$$

Therefore, asymptotically on R, the distance approaches $8n < 9n$.
The total time that both robots take to find the goal is:

$$T(R) = \frac{n}{v} + n\ R + 2\frac{\ell + 2n}{v} = \frac{5n}{v} + n(R + \frac{4}{v^2 R})$$

Minimizing $T(R)$ we obtain: $R = \frac{2}{v}$ (they wait half the time), which gives a total distance of $12n$ and $T = \frac{9n}{v}$ or 9 units of time. That is, they take the same time as one robot! In other words, the advantage of using more than one robot depends upon whether or not they have some instantaneous communication device.

Another possible protocol would be that both robots agree to return to the origin in a certain manner. However, this idea is not better than the one just analyzed on distance or time. However, if we have more robots, we can have two robots searching and coming back to certain points, while the other robots can carry messages between the searchers until the goal is find.

5.4 More Dimensions

The same problems can be stated in two dimensions, as mentioned at the beginning of this section. For example, searching for a line in a plane. There are four cases: the robot either knows or does not know the distance to the line and it either does or does not have any information about the line's slope. We only consider the least information version of the problem, the other cases have been solved elsewhere. The first case, when the slope is known, is trivial. When the distance but not the slope is known the worst case search distance is $(1 + \sqrt{3} + 7\pi/6)n \approx 6.397n$ steps. An algorithm to optimize the average distance walked in this case is discussed in [Glu61a], and a variation where instead of searching a line we are searching for a circle is discussed in [Glu61b]. For further results see [BYCR88, Bel56, Isb57, Ogl74, Jor80, FM86].

Suppose that the robot does not know the line's distance or slope. It seems clear that the optimal search path must be similar with respect to rotations and dilations (that is, the curve must have *spiral similarity*). The only known curve with these properties is the logarithmic spiral.

The robot executes a logarithmic spiral $r = k^\theta$ where $k = 1.250\cdots$ (this value is a numerical approximation for the best logarithmic spiral). If the line is n steps away this algorithm takes approximately $13.81n + O(\ln n)$ steps. (Note that this is only an upper bound since we have assumed that the search path is a logarithmic spiral.) The optimality of this problem is still open.

In general, if we have d dimensions and we are trying to find a goal that is a $(d-1)$-dimensional hyperplane, it is enough to have $d+1$ robots to be able to use a simple symmetric algorithm. For example, in three dimensions, we need 4 robots whose positions at each moment of searching form the vertices of a regular tetrahedron.

Finally, we have been searching for a $(d-1)$-dimensional goal on a d dimensional space with a 0-dimensional object (robot). Another possible generalization is to find the goal with an i-dimensional object, for $0 \leq i \leq d-2$ $(d > 1)$. Another variation is to change the metric space (for example, a Manhattan metric like in [BYCR93]).

6 Searching Text

In this section we search for a non-atomic object: a substring of a longer string or text. For searching text there is a well known space-time trade-off that trades increased index space for faster searching time. Figure 5 shows the main solutions to the problem of searching a substring of a text. The techniques can be divided in three main classes: sequential searching, indexed searching or hybrid methods that combine both indexed and sequential searching. Most of the techniques given are explained later. Random access compression (RAC) is defined as compression schemes that allow random access to the compressed text which support direct searching of a compressed pattern into it by using a standard string searching algorithm. One of this schemes is presented by Manber [Man94].

6.1 Sequential String Searching

It is well known that to search for a pattern of length m in a text of length n (where $n > m$) the search time is $O(n)$ in the worst case (for fixed m). Moreover, in the worst case at least $n - m + 1$ characters must be inspected [Riv77]. However, for different algorithms the constant in the linear

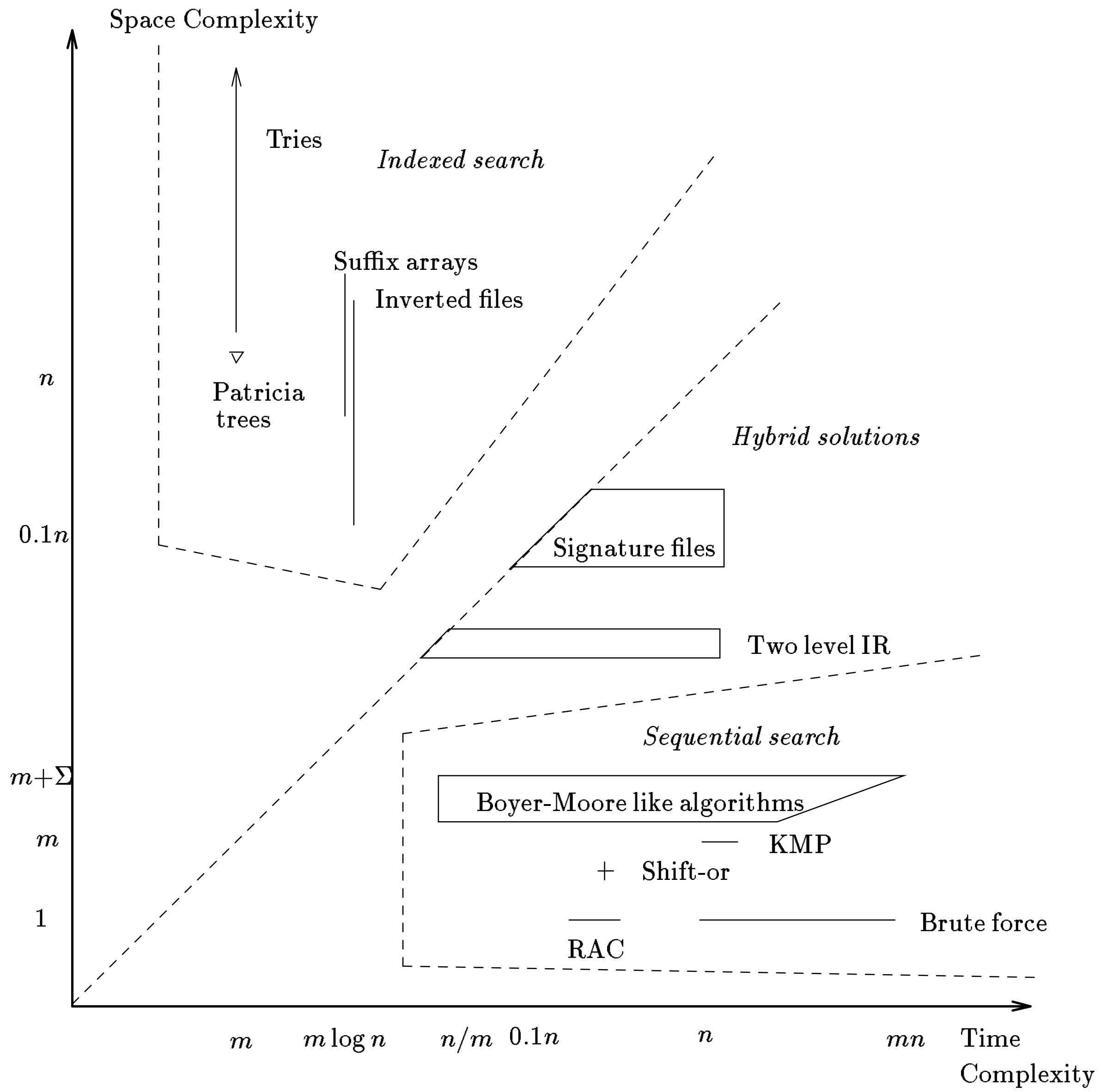

Figure 5: Space-time taxonomy.

term can be very different. For example, in the worst case, the constant multiple in the naive algorithm is m, whereas for the Knuth-Morris-Pratt [KMP77] algorithm it is two. Currently, there is a $4/3n$ upper bound in the worst case number of comparisons with a $(1 + 1/(2m))n$ lower bound for any comparison-based string matching algorithm [CGG90];

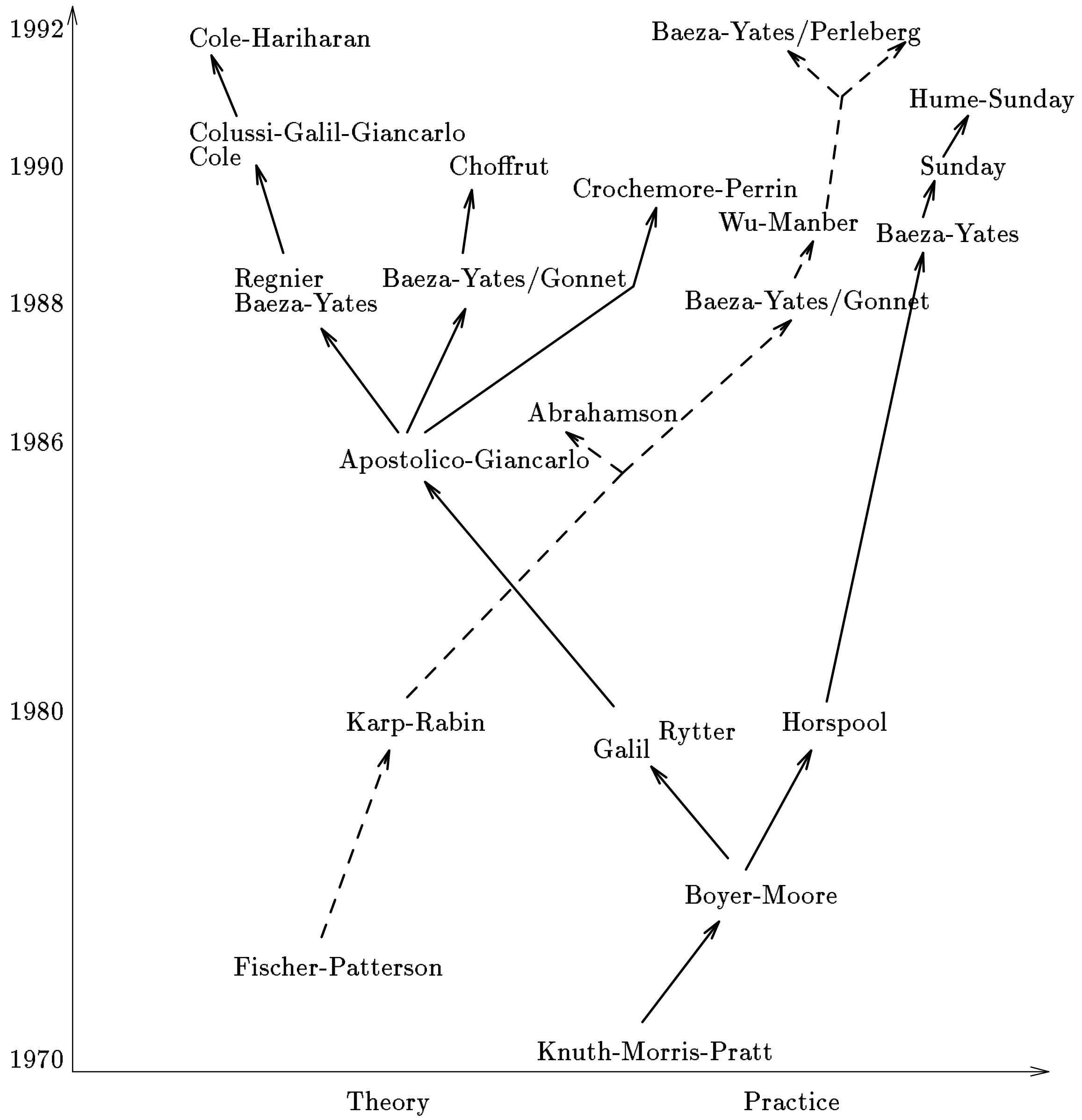

Figure 6: Development of String Matching Algorithms.

We can divide sequential algorithms in two classes: the classical ones that use comparisons and some modern ones that first preprocess all possible comparisons and then search using simpler operations. Here we present the two classical comparison based algorithms: Knuth-Morris-Pratt and a variation of the Boyer-Moore algorithm. For the second case, we present the shift-or and the counting algorithms. Figure 6 shows the development of both classes over time (classical algorithms are shown with solid lines). This figure also shows the theoretical results (left) and the practical ones which lead to fast implementations (right). Currently, it is possible to search in memory faster than 5Mb per second.

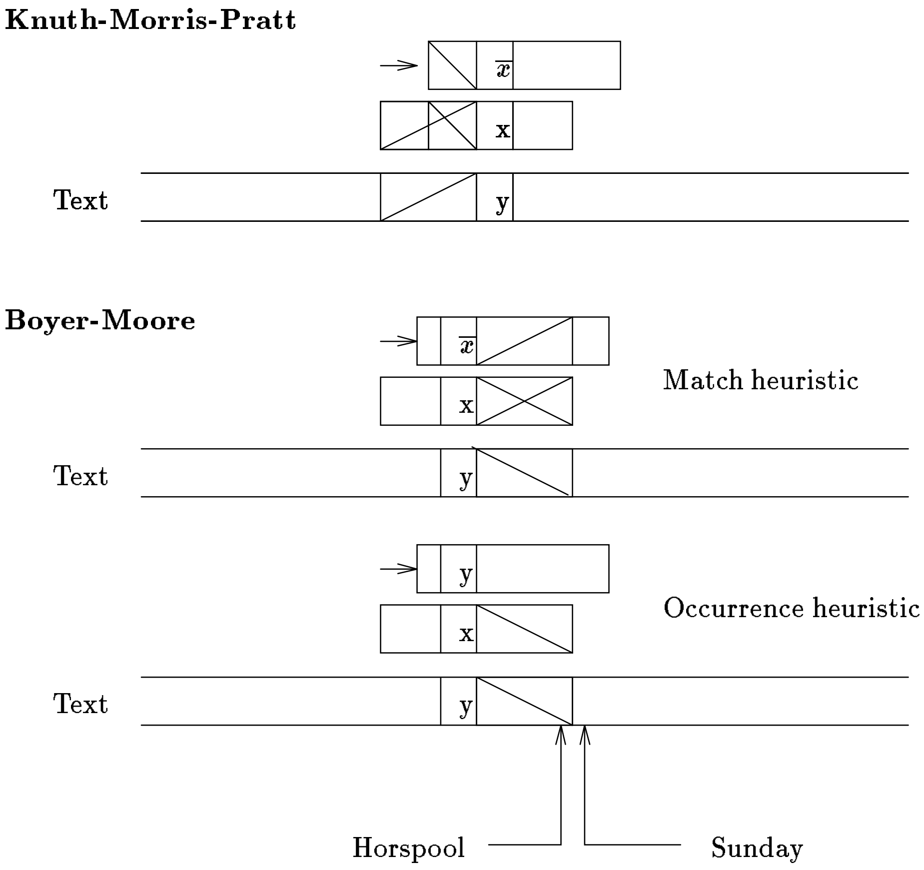

Figure 7: Classical Algorithms.

6.1.1 The Knuth-Morris-Pratt Algorithm

The classic Knuth, Morris and Pratt algorithm, published in 1977 [MP70, KMP77], is the first algorithm for which the constant factor in the linear term, in the worst case, does not depend on the length of the pattern. It is based on preprocessing the pattern in time $O(m)$. In fact, the expected number of comparisons performed by this algorithm (search time only) is bounded by

$$n + O(1) \leq \mathcal{C}_n \leq 2n + O(1) \ .$$

The basic idea behind this algorithm is that each time a mismatch is detected, the "false start" consists of characters that we have already examined. We can take advantage of this information instead of repeating comparisons with the known characters [KMP77]. The idea is shown in the top part of Figure 7, where we see the pieces of string that are equal. Moreover, it is always possible to arrange the algorithm so that the pointer in the text is never decremented. To accomplish this, the pattern is preprocessed to obtain a table that gives the next position in the pattern to be processed after a mismatch. That is, we search prefixes of the pattern in the pattern itself. The exact definition of this table (called *next* in Knuth *et al* [KMP77]) is

$$\begin{aligned} next[j] &= \max\{i|(pattern[k] = pattern[j-i+k] \text{ for } k = 1, \ldots, i-1) \\ &\qquad \text{and } pattern[i] \neq pattern[j]\} \ , \end{aligned}$$

for $j = 1, \ldots, m$. In other words, we consider the maximal matching prefix of the pattern such that the next character in the pattern is *different* from the character of the pattern that caused the mismatch.

Example 1: The *next* table for the pattern *abracadabra* is

```
        a b r a c a d a b r a
next[j] 0 1 1 0 2 0 2 0 1 1 0 5
```

When the value in the *next* table is zero, we have to advance one position in the text and start comparing again from the beginning of the pattern. The last value of the *next* table (five) is used to continue the search after a match has been found. □

The linearity of the algorithm is easy to prove. At any stage, either the pointer in the text or the pattern moves to the right. Each one cannot move more than n times, giving $2n$ comparisons in the worst case. Further explanation of how to preprocess the pattern in time $O(m)$ to obtain this table can be found in the original paper [KMP77] or in most algorithms textbooks.

6.1.2 The Boyer-Moore Algorithm

Also in 1977, the other classic algorithm was published by Boyer and Moore [BM77]. Their main idea is to search from right to left in the pattern. With this scheme, searching is faster on average.

The Boyer-Moore or BM algorithm positions the pattern over the leftmost characters in the text and attempts to match it from right to left. If no mismatch occurs, then the pattern has been found. Otherwise, the algorithm computes a shift; that is, an amount by which the pattern is moved to the right before a new matching attempt is undertaken.

The shift can be computed using two heuristics: the match heuristic and the occurrence heuristic. The *match* heuristic is obtained by noting that when the pattern is moved to the right, it has to

1. match *all* the characters previously matched, and

2. bring a *different* character to the position in the text that caused the mismatch.

These two ideas are shown in the bottom part of Figure 7. The last condition is mentioned in Boyer-Moore's paper [BM77], but was introduced into the algorithm by Knuth *et al* [KMP77].

The *occurrence* heuristic is obtained by noting that we should align the position in the text that caused the mismatch with the first character of the pattern that matches it. However, it is not really necessary to align the mismatched character.

Both shifts can be precomputed based solely on the pattern and the alphabet. Hence, the space needed is $m + c + O(1)$. Given these two shift functions, the algorithm chooses the larger one. The same shift strategy can be applied after a match. In Knuth *et al* [KMP77] the preprocessing of the pattern is shown to be linear in the size of the pattern, as it is for the KMP algorithm; however, their algorithm is incorrect. The corrected version can be found on Rytter's paper [Ryt80]. Later, it was shown that the worst-case number of comparisons is $3n$ [Col91].

A simplified version of the Boyer-Moore algorithm is obtained by using only the *occurrence* heuristic. The main reason behind this simplification is that in practice patterns are not periodic. Also, the extra space needed decreases from $O(m + c)$ to $O(c)$. That is, the space depends only on the size of the alphabet (almost always fixed and known in advance) and not on the length of the pattern (variable). By noticing that after we have found that the pattern is not in the current text position, any text character below the pattern can be used to align the pattern in a new position, Horspool [Hor80] used the last character. However, since we know that the pattern is not there, we can also use the first character of the text after the pattern [Sun90] (see bottom part of Figure 7). That is, the pattern looks like having length $m+1$, and in the best case the number of comparisons is $n/(m + 1)$. Formally, calling the alignment table d, we have

$$d[x] = \min\{s | s = m + 1 \text{ or } (1 \leq s <= m \text{ and } pattern[m + 1 - s] = x)\} \ .$$

Example 4: The d table for the pattern *abracadabra* is

```
d['a'] = 1   d['b'] = 3   d['c'] = 7   d['d'] = 5   d['r'] = 2
```

and the value for any other character is 12. □

Because the matching of the pattern and the corresponding text substring can be done in any order (left to right in the given code), we can preprocess the pattern and find the optimal order, by comparing less frequent characters first. Further improvements can be found in [HS91].

6.1.3 Bit-Parallelism

Consider the following simple parallel algorithm for string searching using m processors. Processor j knows, at any time, whether or not the last j characters of the text that it has seen are equal to the first j characters of the pattern. Consequently, each processor compares the current character that it sees in the text to the $(j+1)$-th character of the pattern, and decides whether the last $j+1$ characters are equal. Processor $m - 1$, then, outputs a signal if the last m characters were equal. The process continues by advancing the reading head, and processor j ($j < m - 1$) sends its state to processor $j + 1$ (after this, processor 0 resets its state), and repeats the process. This scheme is shown in Figure 8, and the search time is n character inspections. The state of every processor can be just a bit if we want to find exact matches or the actual count of matched (or mismatched) characters if we want to solve the problem of string searching with mismatches.

Although this parallel algorithm is not optimal, the crucial point is that the processors can be simulated by a bit sequence of size m for exact matching, updating all the states by or-ing (or and-ing) the sequence with a precomputed value for each symbol in the alphabet. (These values depend only on the pattern.) After that, the final step is simulated by shifting the state by one bit. In pseudo-language we have

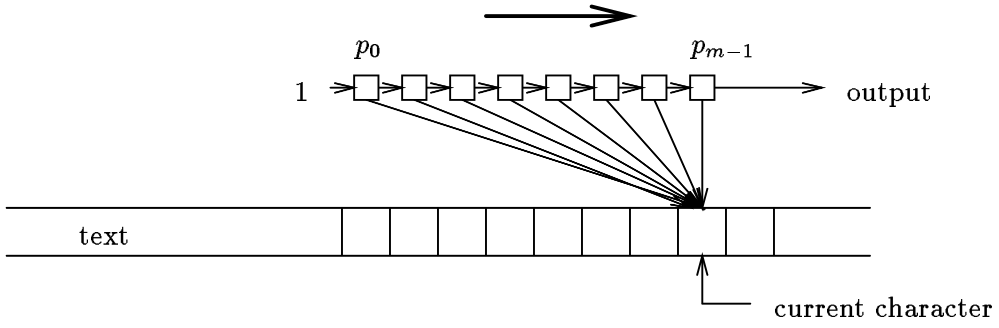

Figure 8: A simple parallel algorithm

```
newstate = (shift left state) or Table[current character]
```

This is extremely fast if m is less or equal than the word size of the machine used, which is almost always the case.

This idea was introduced by Baeza-Yates and Gonnet [BY89, BYG89b] and can be easily extended to string matching with classes (every element of the pattern is a set of symbols, rather than one symbol), string matching with mismatches and multiple string searching [BYG89b]. This idea has been extended to string matching with errors [WM92b, WM92a], and implemented as "agrep", the fastest tool for searching through files, even when we allow errors.

6.1.4 Counting Algorithms

Imagine that there is a counter, c_i, for every position of the text. The counters are initialized to zero, and as each character of the text is read and examined, its position in the pattern (if it is a character in the pattern) is used to find a counter corresponding to the beginning of a potential occurrence in the text, and this counter is incremented. If the character is repeated in the pattern, we increment that many counters. So the basic code is given by

```
for all j such that text[i] = pat[j] do
    increment counter[i-j+1]
```

If there is an instance of the pattern with no mismatches, the counter at the position of this instance of the pattern will be incremented m times. The number of mismatches is equal to m minus the value of the counter, so we can also use this idea for string matching with mismatches. Only m counters are necessary at any one time, so the counters can be implemented with an array of size m that is used in a circular manner. This idea is shown in Figure 9. In practice, because patterns do not have many repeated symbols, the running time is $O(n)$ with $O(m + |\Sigma|)$ extra space, where Σ denotes the underlying alphabet.

This idea is implicit in [MW89], where is used for string matching with insertions and deletions only. It was independently discovered in [BYP92], which presents the algorithm mentioned above for mismatches. This technique is also very fast and does not use character comparisons.

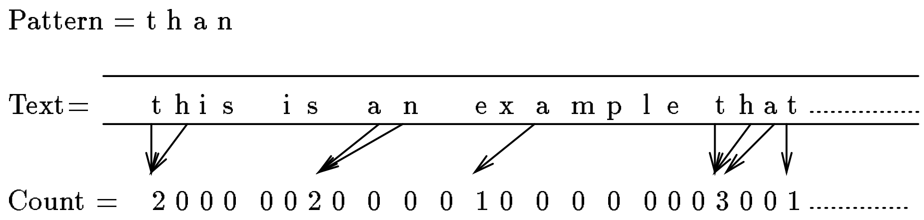

Figure 9: Searching example for the counting approach.

6.2 Retrieval Algorithms for Indexed Text

In this section we structure the algorithms according to the data structure used as index for the text. An index is additional information about the text to achieve faster algorithms than for sequential searching. If the index allows us to reconstruct the text, we say that it is *complete*. In this case we do not have to store the text after the index is built because the index is just an alternate representation for the text.

Inverted files is the traditional technology of standard text retrieval systems. There have been no significant improvements or new breakthroughs in this technology, with the exception of some hybrid algorithms presented here. We refer the reader to [FBY92] for a detailed treatment of standard information retrieval algorithms.

6.2.1 Signature Files

Signature files are an application of hashing to sequential text searching. Because compression techniques are not enough to shrink the text to sizes where a sequential search achieves reasonable answer time, hashing is used to build a compact version of the text, measuring between 10% and 20% of the original size. For this task, the text is divided into words and blocks of words [Fal88]. Every word is hashed to a fixed number of bits and every block is the superimposition of the bit codes of all words in the block. This procedure transforms the text into a sequence of bits. To search in this index, first the query is hashed to a bit string, and that string is sequentially searched in the index (or signature file). Because more than one word can be mapped to the same bit string, we may obtain undesired answers, so that every potential answer must be checked again with the original text.

Reasonable performance can be achieved in medium size files by an adequate selection of parameters, yielding a low probability of undesired answers (or false drops) [FC84, FC87, SDRK87]. However, this method is not suitable for large files or for applications where it is not possible to divide the text into words, such as in protein sequence databases. This technique can be considered a hybrid of indexed and sequential search. A complete survey on the different variations of signature files can be found in [FBY92].

6.2.2 Digital Trees

Another traditional index is a digital tree or trie. Trie search is based on the digital decomposition of the strings stored in it. If the alphabet is ordered, we have a lexicographically ordered tree. The root of the trie uses the first character, the children of the root use the second character, and so on. If the remaining subtrie contains only one string, then it is stored in an external node. Figure 10(a) shows a trie built from the beginning of words of the text "this is a text example ...".

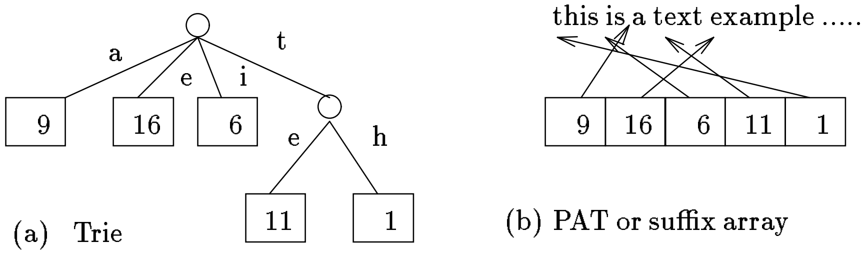

Figure 10: (a) Trie example (external node label indicates position in the text).
(b) Suffix array of the same text example.

If we use suffixes of a same piece of text to build the trie, we call it a suffix tree. This particular case has several properties. A suffix tree may have unary nodes. To avoid unary nodes, we collapse them by indicating in every node which position of the string should be examined. In this way, we make sure that we only need $n-1$ internal nodes and n external nodes, n being the number of index points or suffixes. This variant is called a Patricia tree or a compact suffix tree.

The Patricia tree easily supports prefix and range searching, as well as proximity and regular expression searching [GBY91]. Although a Patricia tree needs $O(n)$ space, the actual constant is important in practical applications. We need at least two pointers for every internal node, and one pointer for every external node. This means $12n$ bytes for n index points.

A simple and efficient solution to this problem is to store only the external nodes. Thus, the whole index degenerates into a single array of external nodes lexicographically ordered by suffixes called PAT array by Gonnet [Gon87]. With some additions this idea was independently discovered by Manber and Myers [MM90] where it is called suffix arrays. The size of the index is now $4n$ bytes, where n is the number of index points.

There is a straightforward argument showing that the PAT or suffix array contains most of the information we had in the Patricia tree at the cost of a factor of $\log_2 n$. The argument simply says that for any interval in the array which contains all the external nodes that would be in a subtree, we can divide the interval according to the next bit which is different in $\log_2 n$ comparisons in the worst case. The sizes of the subtrees are trivially obtained from the limits of any portion of the array. Thus, any operation on a Patricia tree can be simulated in $O(\log n)$ accesses. Figure 10(b) shows this structure for the same example used for the trie.

It turns out that it is not necessary to simulate the Patricia tree for prefix and range searching, so we obtain an algorithm which is $O(\log n)$ instead of $O(\log^2 n)$ for these operations. Actually prefix searching and range searching become more uniform. Both can be implemented by doing an indirect binary search over the array with the results of the comparisons being less than, equal to, or greater than. In this way searching takes at most $2\log_2 n - 1$ string comparisons and $4\log_2 n$ disk accesses (both in the worst case). This represents a significant economy in space at the cost of a modest deterioration in access time [GBYS92]. It is possible to build PAT arrays in $O(n \log n)$ time.

For regular expression searching, the time increases by a factor of $O(\log n)$ [BYG89a]. Details about proximity searching (searching for the occurrence of a string near other string) can be found Manber and Baeza-Yates [MBY91].

6.2.3 A Hybrid Solution

We propose here a generalization for the two-level information retrieval scheme of Manber and Wu [MW93]. In their proposal the text is divided into 256 blocks of the same size, and an inverted file of all the different words of the text is built, indicating only the blocks where each word appears (1 byte per block). For large texts, empirical results show that the inverted file requires less than 5% of the text size, as the number of different words does not grow as fast as the size of the text. To search for a word, we look first in the inverted file, and then we use sequential search (for example, using the best Boyer-Moore type string searching algorithm) in all the corresponding blocks. So, if the word does not exist, the search has logarithmic complexity. Otherwise, in the worst case we may need to search all the blocks. However, in practice, since some words have locality of reference, we have to search just one or two blocks. This idea works reasonable well for texts of size up to 200 megabytes.

This two-level hybrid scheme is based on words. Our generalization allows to search for any sequence of symbols. Instead of an inverted file we use an index storing all different sequences of length k of the text (they could also be all retrievable sequences, for example, just sequences starting at the beginning of words). As before, this index points to all the blocks of the text having each sequence. The block size is also a parameter, denoted by B. Figure 11 shows the basic data structure. The parameter k models the granularity of the index, while B represents the granularity for accessing the text (equivalent to document size in traditional IR systems). For example, if $k = 0$, we do not have an index, and we get sequential search. If $k = O(\log n)$ we have a full index. The size of B should be a multiple of the granularity of the secondary storage used.

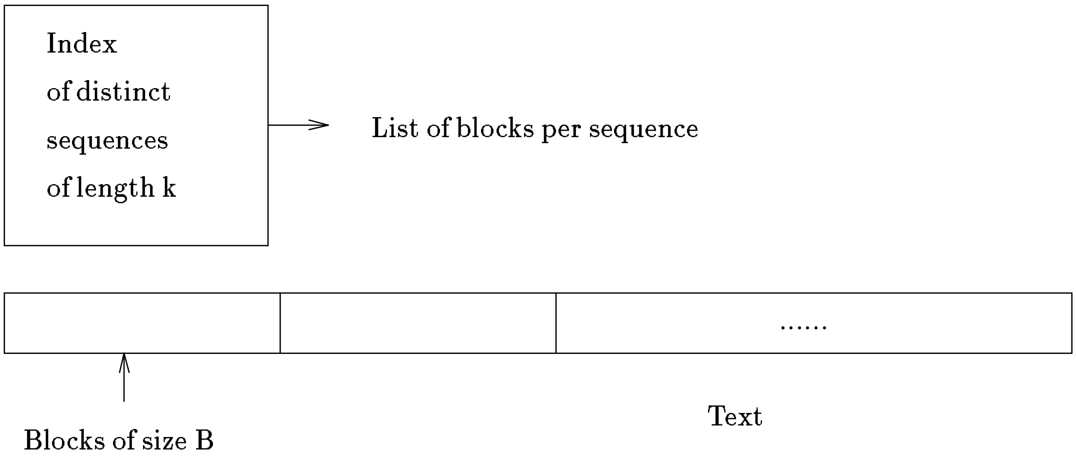

Figure 11: General two-level hybrid scheme.

The searching algorithm is the same as before. If we want to search for a pattern p of length m, we look for p in the index if $m \leq k$, or one sequence of length k extracted from p (we could use more than one to decrease the number of blocks). After that, we use a fast string search algorithm for p in the corresponding block.

Based on the above, the implementation of the index has to provide fast search and also be able to find all blocks having a prefix of a sequence. Other possibilities, using linear space on the number of different sequences are a Patricia tree, a suffix array or an inverted file.

7 Searching for Similar Objects

In this section we show how to search for similar objects under very general assumptions. This section is based on [BYCMW94]. We assume that we have a fixed set X, which we can preprocess and store in a data structure. The problem is to find all elements in the set that are close, under some distance function – we study in detail the Hamming and Levenshtein distance functions – to a query element. We refer the reader to [SW90, Mur83] for techniques that assume linear orderings, Euclidean, and similar distance functions. The problem is much harder, however, for distance functions that are not related to a linear ordering.

The practical approach for finding all elements in a database close to a query element is typically to design quick algorithms that approximate the distance (e.g., BLAST [AGM$^+$90] and FASTA [LP85] for biological sequences, and [BGKN90] for speech recognition). The approximated distance can be used to filter elements that seem too far away, but these algorithms need to be applied to all elements of the database. Their running time is therefore linear in the size of the database.

Another approach, for arbitrary distance functions, was given by Burkhard and Keller [BK73] in the context of database queries. They designed a tree structure, later called BK-trees, making heavy use of the triangle inequality. Different algorithmic approaches that lead to $o(N)$ expected performance both theoretically and in practice were developed recently by several people. They are quite different, although they all use the idea of filtering large regions of the database (or large text) so that more expensive techniques are used only for small regions [Mye94, Ukk92, GCB92, BYG90, Ukk93, CL90].

Here we present yet another approach, similar to BK-trees, which achieves $O(N^\alpha)$, with $\alpha < 1$, expected complexity for finding close matches [BYCMW94]. We make heavy use of the triangle inequality. We present a tree structure that can handle proximity queries for any distance function that satisfies the triangle inequality. The main goal of this structure is to minimize the number of element comparisons, as opposed to other data structure operations (such as tree traversal). This is especially important in applications where computing distances between two elements is a much more expensive operation than, say, following pointers. Our tree structure differs from other tree structures in that the keys on each level of the tree are all the same, so we have just one key per level. In other words, which comparisons we make does not depend on the results of previous comparisons up the tree. Thus the name fixed-queries tree or FQ-tree. This may seem like a poor strategy (e.g., regular binary search trees will yield a linear search with this policy), but it has several advantages for our intended applications. First, it minimizes element comparisons, because traversing different parts of the tree does not require more comparisons. Second, it allows all comparisons to the keys to be done in one batch, enabling easy parallelism. Third, it simplifies the data structure and its analysis.

A major strength of FQ-trees, which also applies to BK-trees, is that the data structure and the search algorithms are independent of the distance function, as long as it satisfies the triangle inequality. If someone comes up with different distance measures or different algorithms to compute them, they will still be able to use our scheme without any other change to filter their data. The amount of filtering may depend on the distance distribution but not the algorithms. Our scheme allows one to use the expensive exact algorithms (e.g., dynamic programming) and still be able (at least for small distances) to search in a large database.

7.1 Fixed-Queries Trees

Let U be the set of all possible elements defined by the problem. For the rest of the section we will assume that $X \subset U$ is a large set of elements, each of which is a large object by itself, and that U is much larger than X. In the analysis section we will concentrate on strings as elements, but they can be arbitrary. An FQ-tree differs from a regular search tree in two major respects. First, the keys are not numbers from an ordered set, they are members of U. Second, all internal nodes at the same level are associated with the same key. We will use keys to denote the elements that are compared while traversing the tree, and elements to denote members of X. Let *dist* be a distance function that satisfies the triangle inequality and assume that the set of all possible distances is a finite set $\{d_0, d_1, d_2, ...d_s\}$ If this set is not finite, or if it is very large, we can discretize it to s different ranges of values.) An FQ-tree for a set of elements X and a distance function *dist* is a tree that satisfies the following four properties.

- All elements of X are associated with the leaves.
- If X contains only b elements, then the tree consists of one (leaf) node containing all elements (b is the *bucket* size).
- All internal nodes of depth r are associated with the same key k_r. The keys can be selected at random, they can be members of X, or they can be selected especially to optimize the tree; we assume that they are random.
- Every internal node v is a root of a valid FQ tree associated with a set $X_v \subset X$. An internal node v with key k_r has one subtree for every non-empty set X_i defined by

$$X_i = \{x \in X_v \text{ such that } dist(k_r, x) = d_i \geq 0\}.$$

Figure 12 shows an example of an FQ-tree.

A search for a query element q proceeds down the tree by computing at each level the distance between q and the key k_i for that level. If the search is an exact search, then at each level the child that corresponds to the distance between q and k_i is selected until a leaf is reached. A neighborhood or proximity search is more complicated. Suppose that we want to find all elements of X that are within distance D to q. We use the triangle inequality to filter nodes. If we are at an internal node v with associated key k_i and if $dist(q, k_i) = d_i$, then all children of v associated with distances d such that $d_i - D \leq d \leq d_i + D$ may contain elements with distance D to q. We search, recursively, all of them. This is where the advantage of having the same key at all nodes of the same level comes to play. Even though we may traverse many children, we make only one key comparison per level. Therefore, the main cost of a proximity search is proportional to the height of the tree

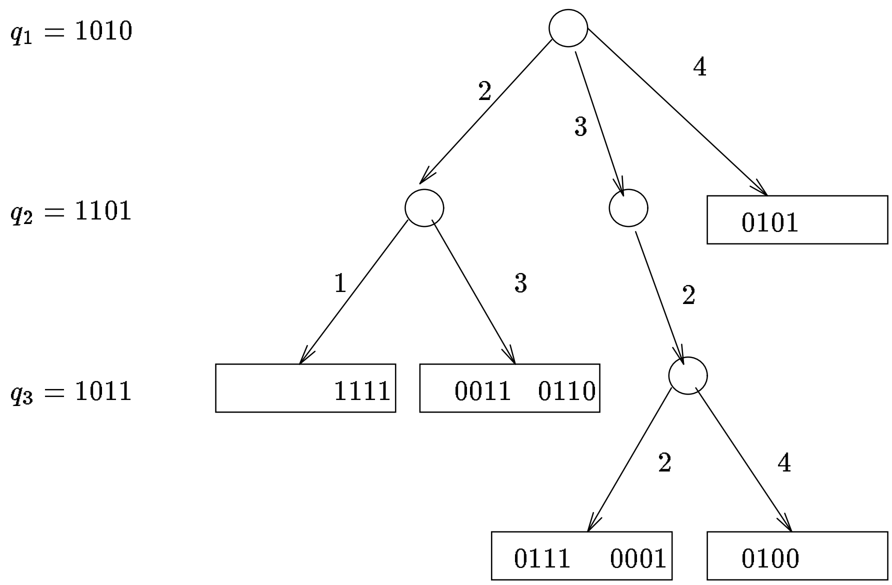

Figure 12: Example of an FQ-tree with $b = 2$ using the Hamming distance.

(which indicates how many key comparisons we need to make), plus the number of elements in all the buckets that are reached in the search (which are all unfiltered elements – those that are within distance $d_i - D$ to $d_i + D$ for all d_i). The overhead of traversing the data structure (e.g., finding the children to follow), is only secondary, because following pointers is much less expensive than comparing two complex objects.

7.2 Improving FQ-trees

There is one important variant of FQ-trees that deserves mention. Given a leaf v, even for buckets of size 1, it may still be useful to increase its depth by adding internal nodes of degree 1 to it. In other words, we may want to replace leaves with paths. The reason that enlarging the tree can actually improve the search is the one-key-per-level rule. If we increase the height of the tree by, say 30, we add only 30 more key comparisons for the entire proximity search. We may add many more node traversals, but as we said, they cost much less. But 30 more key comparisons will filter many more elements, so the final set will be smaller. These additional keys can also be chosen at random. The next paragraph shows that the optimal number of queries is logarithmic, and we obtain logarithmic proximity searching for fixed D.

Because the height is fixed, its analysis is simpler than for FQ-trees [BYCMW94]. Let $F_D^k(n)$ be the number of elements not yet filtered by a proximity search of distance up to D after applying k fixed queries. Then, the expected number of comparisons for a proximity query is

$$P_D^k(n) = k + F_D^k(n)$$

Let β_D be the probability of not filtering an element when doing the proximity search at distance D. If an element is at distance k to a query, it is not filtered with probability $\sum_{i=k-D}^{k+D} p_i$. The element is at distance k with probability p_k, so

$$\beta_D = \sum_{k=0}^{s} p_k \sum_{i=k-D}^{k+D} p_i$$

Note that β_D converges to 1 when D increases. So, the expected number of elements not filtered between two consecutive levels are related by $F_D^k(n) = \beta_D F_D^{k-1}(n)$. Clearly, $F_D^0 = n$, so $F_D^k(n) = \beta_D^k n$. Because $F_D^k(n)$ decreases when k grows, the optimal k is obtained when $P_D^k(n) \leq P_D^{k+1}(n)$. That is, when

$$k + \beta_D^k n \leq k + 1 + \beta_D^{k+1}$$

Solving, we obtain the optimal k for a given D

$$k_D = \frac{\log(n(1-\beta_D))}{\log(1/\beta_D)}$$

Replacing this k in $P_D^k(n)$ we get

$$P_D(n) = \frac{\log(n(1-\beta_D))}{\log(1/\beta_D)} + \frac{1}{1-\beta_D}$$

That is, $P_D(n)$ is logarithmic for the optimal k. Nevertheless, the constant factor in the log term grows exponentially with D, so this is good for small to medium D.

On the other hand, the number of nodes visited is bigger than in FQ-trees. In fact, using a recurrence similar to the one for FQ-trees, it is possible to show that the number of nodes visited is $O(k_D n^\alpha)$ for $\alpha < 1$ which could easily be larger than n even for small D. So, these trees are good when the cost of comparing two elements is very high, like comparing two genetic sequences, polygons or graphs. Another problem is that the variance of the number of elements filtered per level is large (increases with every level), so we may need more queries in practice to achieve the desired filtering.

To decrease the number of nodes visited, we may compress paths of degree 1 by using the same idea of Patricia trees. We can store in every node which fixed query (or how many we have to skip) we have to use in that node. Still, we cannot compress all the nodes if we want to filter that element. Another idea, instead of fixing the height, is fixing the probability of filtering in every path recursively.

Acknowledgements

We thank the helpful comments of Gonzalo Navarro and Nivio Ziviani.

References

[AACS87] A. Aggarwal, B. Alpern, K. Chandra, and M. Snir. A model for hierarchical memory. In *Proc. of the 19th Annual ACM Symp. of the Theory of Computing*, pages 305–314, New York, 1987.

[AC88] A. Aggarwal and K. Chandra. Virtual memory algorithms. In *20th Annual ACM Symp. on Theory of Computing*, pages 173–185, Chicago, 1988.

[ACS87] A. Aggarwal, K. Chandra, and M. Snir. Hierarchical memory with block transfer. In *28th Annual IEEE Symp. on Foundations of Computer Science*, pages 204–216, Los Angeles, 1987.

[AGM+90] S.F. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *J. Molecular Biology*, 15:403–410, 1990.

[AV87] A. Aggarwal and J. Vitter. The i/o complexity of sorting related problems. In *14th ICALP*, pages 467–478, Karlsruhe, West Germany, 1987.

[BB80] J.L. Bentely and D.J. Brown. A general class of resource tradeoffs. In *FOCS*, volume 21, pages 217–228, Syracuse NY, Oct 1980.

[Bel56] R.; Bellman. A minimization problem. *Bulletin of the American Mathematical Society*, 62:270, 1956.

[BGKN90] L. R. Bahl, P.S. Gopalakrishnan, D.S. Kanevsky, and D.S. Nahamoo. A fast admissible method for identifying a short list of candidate words. Technical Report RC 15874, IBM, June 1990.

[BK73] W.A. Burkhard and R.M. Keller. Some approaches to best-match file searching. *Communications of the ACM*, 16(4):230–236, 1973.

[BM77] R. Boyer and S. Moore. A fast string searching algorithm. *C.ACM*, 20:762–772, 1977.

[BY87] R. Baeza-Yates. On the average case of string matching algorithms. Technical Report CS-87-66, Department of Computer Science, University of Waterloo, Ontario, Canada, 1987.

[BY89] R.A. Baeza-Yates. *Efficient Text Searching.* PhD thesis, Dept. of Computer Science, University of Waterloo, May 1989. Also as Research Report CS-89-17.

[BYCMW94] R.A. Baeza-Yates, W. Cunto, U. Manber, and S. Wu. Proximity matching using fixed queries trees. In *5th Combinatorial Pattern Matching*, LNCS 807, pages 198–212, Asilomar, CA, June 1994. Springer-Verlag.

[BYCR88] R.A. Baeza-Yates, J. Culberson, and G. Rawlins. Searching with uncertainty. In R. Karlsson and A. Lingas, editors, *Proceedings SWAT 88, First Scandinavian Workshop on Algorithm Theory, Lecture Notes in Computer Science 318*, pages 176–189, Halmstad, Sweden, July 1988.

[BYCR93] R.A. Baeza-Yates, J. Culberson, and G. Rawlins. Searching in the plane. *Information and Computation*, 106(2):234–252, Oct 1993. Preliminary version titled "Searching with Uncertainty" was presented at SWAT'88, Halmstad, Sweden, LNCS 318, 176-189.

[BYG89a] R. Baeza-Yates and G.H. Gonnet. Efficient text searching of regular expressions. In G. Ausiello, M. Dezani-Ciancaglini, and S. Ronchi Della Rocca, editors, *ICALP'89, Lecture Notes in Computer Science 372*, pages 46–62, Stresa, Italy, July 1989. Springer-Verlag.

[BYG89b] R. Baeza-Yates and G.H. Gonnet. A new approach to text searching. In *Proc. of 12th ACM SIGIR*, pages 168–175, Cambridge, Mass., June 1989. (Addendum in ACM SIGIR Forum, V. 23, Numbers 3, 4, 1989, page 7.).

[BYG90] R.A. Baeza-Yates and G.H. Gonnet. All-against-all sequence matching. Dept. of Computer Science, Universidad de Chile, 1990.

[BYP92] R.A. Baeza-Yates and C.H. Perleberg. Fast and practical approximate pattern matching. In A. Apostolico, M. Crochemore, Z. Galil, and U. Manber, editors, *Combinatorial Pattern Matching*, Lecture Notes in Computer Science 644, pages 185–192, Tucson, AZ, April/May 1992. Springer Verlag.

[BYS94] R. Baeza-Yates and R. Schott. Parallel searching in the plane. In *RFIA'94 (9th Conference on Shape Recognition and Artificial Intelligence*, pages 557–566, Paris, France, Jan 1994. Preliminary version presented in XII Int. Conf. of the Chilean Computer Society, Santiago, Chile, Oct 1992, 269-279.

[CGG90] L. Colussi, Z. Galil, and R. Giancarlo. The exact complexity of string matching. In *31st FOCS*, volume 1, pages 135–143, St. Louis, MO, Oct 1990. IEEE.

[CL90] W. Chang and E. Lawler. Approximated string matching in sublinear expected time. In *Proc. 31st FOCS*, pages 116–124, St. Louis, MO, Oct 1990. IEEE.

[Col91] R. Cole. Tight bounds on the complexity of the Boyer-Moore string matching algorithm. In *SODA'91*, pages 224–233, San Francisco, CA, Jan 1991.

[Fal88] C. Faloutsos. Signature files : an integrated access method for text and attributes suitable for optical disk storage. *BIT*, 28(4):736–754, 1988.

[FBY92] W. Frakes and R. Baeza-Yates, editors. *Information Retrieval: Data Structures and Algorithms*. Prentice-Hall, 1992.

[FC84] C. Faloutsos and S. Christodoulakis. Signature files: An access method for documents and its analytical performance evaluation. *ACM TOOIS*, 2(4):267–288, Oct 1984.

[FC87] C. Faloutsos and S. Christodoulakis. Description and performance analysis of signature file methods. *ACM TOOIS*, 5(3):237–257, 1987.

[FM86] V. Faber and J. Mycielski. The shortest curve that meets all the lines that meet a convex body. *American Mathematical Monthly*, 93:796–801, 1986.

[Gal80] S. Gal. *Search Games*. Academic Press, 1980.

[GBY91] G.H. Gonnet and R. Baeza-Yates. *Handbook of Algorithms and Data Structures - In Pascal and C.* Addison-Wesley, Wokingham, UK, 1991. (second edition).

[GBYS92] G.H. Gonnet, R. Baeza-Yates, and T. Snider. New indices for text: Pat trees and pat arrays. In W. Frakes and R. Baeza-Yates, editors, *Information Retrieval: Algorithms and Data Structures*, chapter 5, pages 66–82. Prentice-Hall, 1992.

[GCB92] G. Gonnet, M. Cohen, and S. Benner. Exhaustive matching of the entire protein sequence database. *Science*, 256:1443–45, 1992.

[GG86] Z. Galil and R. Giancarlo. Improved string matching with k mismatches. *SIGACT News*, 17:52–54, 1986.

[Glu61a] B. Gluss. An alternative solution to the 'lost at sea' problem. *Naval Research Logistics Quarterly*, 8:117–128, 1961.

[Glu61b] B. Gluss. The minimax path in a search for a circle in the plane. *Naval Research Logistics Quarterly*, 8:357–360, 1961.

[Gon87] G.H. Gonnet. PAT 3.1: An efficient text searching system, User's manual. UW Centre for the New OED, University of Waterloo, 1987.

[Hor80] R. N. Horspool. Practical fast searching in strings. *Software - Practice and Experience*, 10:501–506, 1980.

[HS91] A. Hume and D.M. Sunday. Fast string searching. *Software - Practice and Experience*, 21(11):1221–1248, Nov 1991.

[Isb57] J. R. Isbell. An optimal search pattern. *Naval Research Logistics Quarterly*, 4:357–359, 1957.

[Jor80] H. Joris. Le chasseur perdu dans la forêt. *Elemente der Mathematik*, 35:1–14, 1980.

[KMP77] D.E. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J on Computing*, 6:323–350, 1977.

[Kni88] W.J. Knight. Search in an ordered array having variable probe cost. *SIAM J. of Computing*, 17(6):1203–1214, December 1988.

[Knu91] Knuth. Theory and practice. *Theoretical Computer Science*, 90, 1991.

[LP85] D. J. Lipman and W.R. Pearson. Rapid and sensitive protein similarity searches. *Science*, 227:1435–1441, 1985.

[Man94] Udi Manber. A text compression scheme that allows fast searching directly in the compressed file. In M. Crochemore and D. Gusfield, editors, *CPM'94*, LNCS 807, pages 113–124, Asilomar, CA, June 1994. Springer-Verlag.

[MBY91] U. Manber and R. Baeza-Yates. An algorithm for string matching with a sequence of don't cares. *Information Processing Letters*, 37:133–136, February 1991.

[MM90] U. Manber and G. Myers. Suffix arrays: A new method for on-line string searches. In *1st ACM-SIAM Symposium on Discrete Algorithms*, pages 319–327, San Francisco, January 1990.

[MP70] J.H. Morris and V.R. Pratt. A linear pattern matching algorithm. Technical Report 40, Computing Center, University of California, Berkeley, 1970.

[Mur83] F. Murtagh. A survey of recent advances in hierarchical clustering algorithms'. *IEEE Computer*, 26(4):354–359, 1983.

[MW89] U. Manber and S. Wu. An algorithm for approximate string matching with non uniform costs. Technical Report TR-89-19, Department of Computer Science, University of Arizona, Tucson, Arizona, Sept 1989.

[MW93] Udi Manber and S. Wu. A two-level approach to information retrieval. Technical Report TR 93-06, University of Arizona, 1993.

[Mye94] E. Myers. A sublinear algorithm for approximate keyword matching. *Algorithmica*, 1994.

[Ogl74] C. S. Oglivy. *Tomorrow's Math: Unsolved Problems for the Amateur.* Oxford University Press, 1974.

[PIA78] Y. Perl, A. Itai, and H. Avni. Interpolation search - a log log n search. *C.ACM*, 21(7):550–553, July 1978.

[PR77] Y. Perl and Edward M. Reingold. Understanding the complexity of interpolation search. *Inf. Proc. Letters*, 6(6):219–221, Dec 1977.

[Riv77] R. Rivest. On the worst-case behavior of string-searching algorithms. *SIAM J on Computing*, 6:669–674, 1977.

[Ryt80] W. Rytter. A correct preprocessing algorithm for Boyer-Moore string-searching. *SIAM J on Computing*, 9:509–512, 1980.

[SDRK87] R. Sacks-Davis, K. Ramamohanarao, and A. Kent. Multikey access methods based on superimposed coding techniques. *ACM TODS*, 12(4):655–696, 1987.

[Sun90] D.M. Sunday. A very fast substring search algorithm. *Communications of the ACM*, 33(8):132–142, Aug 1990.

[SW90] D. Shasha and Wang. New techniques for best-match retrieval. *ACM Transactions on Information Systems*, 8:140–158, 1990.

[Ukk92] E. Ukkonen. Approximate string matching with q-grams and maximal matches. *Theoretical Computer Science*, 92(1):191–212, January 1992.

[Ukk93] E. Ukkonen. Approximate string matching over suffix trees. In *CPM'94*, LNCS 520, pages 240–248, Padova, Italy, Jun 1993. Springer-Verlag.

[WM92a] S. Wu and U. Manber. Agrep - a fast approximate pattern-matching tool. In *Proceedings of USENIX Winter 1992 Technical Conference*, pages 153–162, San Francisco, CA, Jan 1992.

[WM92b] S. Wu and U. Manber. Fast text searching allowing errors. *Communications of the ACM*, 35:83–91, Oct 1992.