The magic of Algorithms!

Lectures on some algorithmic pearls

Paolo Ferragina, Università di Pisa

These notes should be an advise for programmers and software engineers: no matter how much smart you are, the so called *"5-minutes thinking"* is not enough to get a reasonable solution for your real problem, unless it is a toy one! Real problems have reached such a large size, machines got so complicated, and algorithmic tools became so sophisticated that you cannot improvise to be an *algorithm designer*: you should be trained to be one of them!

These lectures provide a witness for this issue by introducing challenging problems together with elegant and efficient algorithmic techniques to solve them. In selecting their topics I was driven by a twofold goal: from the one hand, provide the reader with an *algorithm engineering toolbox* that will help him/her in attacking programming problems over massive datasets; and, from the other hand, I wished to collect the stuff that I would have liked to see when I was a master/phd student!

The style and content of these lectures is the result of many hours of highlighting and, sometime hard and fatiguing, discussions with many fellow researchers and students. Actually some of these lectures composed the courses in Information Retrieval and/or Advanced Algorithms that I taught at the University of Pisa and in various International PhD Schools, since year 2004. In particular, a preliminary draft of these notes were prepared by the students of the *"Algorithm Engineering"* course in the Master Degree of Computer Science and Networking in Sept-Dec 2009, done in collaboration between the University of Pisa and Scuola Superiore S. Anna. Some other notes were prepared by the Phd students attending the course on *"Advanced Algorithms for Massive DataSets"* that I taught at the BISS International School on Computer Science and Engineering, held in March 2010 (Bertinoro, Italy). I used these drafts as a seed for some of the following chapters.

My ultimate hope is that reading these notes you'll be pervaded by the same pleasure and excitement that filled my mood when I met these algorithmic solutions for the first time. If this will be the case, please read more about Algorithms to find inspiration for your work. It is still the time that *programming is an Art*, but you need the good *tools* to make itself express at the highest beauty!

P.F.

1

Searching Strings by Prefix

This problem is experiencing renewed interest in the algorithmic community because of new applications spurring from Web search-engines. Think to the *auto-completion* feature currently offered by mayor search engines Google, Bing and Yahoo, in their search bars: It is a prefix search executed on-the-fly over millions of strings, using the query pattern typed by the user as the string to search. The dictionary typically consists of the most recent and the most frequent queries issued by other users. This problem is made challenging by the size of the dictionary and by the time constraints imposed by the patience of the users. In this chapter we will describe many different solutions to this problem of increasing sophistication and efficiency both in time, space and I/O complexities.

The prefix-search problem. *Given a dictionary $\mathcal{D}$ consisting of n strings of total length N, drawn from an alphabet of size σ, the problem consists of preprocessing $\mathcal{D}$ in order to retrieve (or just count) the strings of $\mathcal{D}$ that have P as a prefix.*

We mention two other typical string queries which are the *exact* search and the *substring* search within the dictionary strings in $\mathcal{D}$. The former is best addressed via hashing because of its simplicity and practical speed; Chapter **??** will detail several hashing solutions. The latter problem is more sophisticated and finds application in computational genomics and asian search engines, just to cite a few. It consists of finding all *positions* where the query-pattern P occurs as a substring of the dictionary strings. Surprisingly enough, it does exist a simple *algorithmic reduction* from substring search to prefix search over the set of *all suffixes of the dictionary strings*. This reduction will be commented in Chapter **??** where the Suffix Array and the Suffix Tree data structures will be introduced. As a result, we can conclude that the prefix search is the backbone of other important search problems on strings, so the data structures introduced in this chapter offer applications which go far beyond the simple ones discussed below.

1.1 Array of string pointers

We start with a simple, common solution to the prefix-search problem which consists of an array of pointers to strings stored in arbitrary locations on disk. Let us call $A[1, n]$ the array of pointers,

which are *indirectly* sorted according to the strings pointed to by its entries. We assume that each pointer takes w bytes of memory, typically 4 bytes (32 bits) or 8 bytes (64 bits). Several other representations of pointers are possible, as e.g. variable-length representations, but this discussion is deferred to Chapter **??**, where we will deal with the efficient encoding of integers.

Figure 1.1 provides a running example in which the dictionary strings are stored in an array S, according to an arbitrary order.

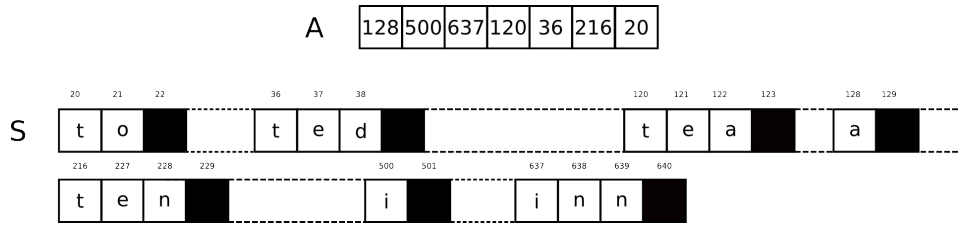

FIGURE 1.1: The array S of strings and the array A of (indirectly) sorted pointers to S's strings.

There are two crucial properties that the (sorted) array A satisfies:

- all dictionary strings prefixed by P occur contiguously if lexicographically sorted. So their pointers occupy a subarray, say $A[l, r]$, which may be possibly empty if P does not prefix any dictionary string.
- the string P is lexicographically located beyween $A[l-1]$ and $A[l]$.

Since the prefix search returns either the number of dictionary strings prefixed by P, hence the value $r - l + 1$, or visualizes these strings, the key problem is to identify the two extremes l and r efficiently. To this aim, we reduce the prefix search problem to the *lexicographic search* of a pattern Q in $\mathcal{D}$: namely, the search of the lexicographic position of Q among $\mathcal{D}$'s strings. The formation of Q is simple: Q is either the pattern P or the pattern $P\#$, where # is larger than any other alphabet character. It is not difficult to convince yourself that $Q = P$ will precede the string $A[l]$ (see above), whereas $Q = P\#$ will follow the string $A[r]$. This means actually that two lexicographic searches for patterns of length no more than $p + 1$ are enough to solve the prefix-search problem.

The lexicographic search can be implemented by means of an (indirect) binary search over the array A. It consists of $O(\log n)$ steps, each one requiring a string comparison between Q and the string pointed by the entry tested in A. The comparison is lexicographic and thus takes $O(p)$ time and $O(p/B)$ I/Os, because it may require in the worst case the scan of all $\Theta(p)$ characters of Q. The poor time and I/O-complexities derive from the *indirection*, which forces no locality in the memory/string accesses of the binary search. The inefficiency is even more evident if we wish to retrieve all n_occ strings prefixed by P, and not just count them. After that the range $A[l, r]$ has been identified, each string visualization elicits at least one I/O because contiguity in A does not imply contiguity of the pointed strings in S.

THEOREM 1.1 *The complexity of a prefix search over the array of string pointers is $O(p \log n)$ time and $O(\frac{p}{B} \log n)$ I/Os, the total space is $N + (1 + w)n$ bytes. Retrieving the n_{occ} strings prefixed by P needs $\Omega(n_{occ})$ I/Os.*

Proof Time and I/O complexities derive from the previous observations. For the space occu-

pancy, A needs n pointers, each taking a memory word w, and all dictionary strings occupy N bytes plus one-byte delimiter for each of them (commonly \0 in C). ■

The bound $\Omega(n_{occ})$ may be a major bottleneck if the number of returned strings is large, as it typically occurs in queries that use the prefix search as a preliminary step to select a *candidate set of answers* that have then to be refined via a proper post-filtering process. An example is the solution to the problem of *searching with wild-cards* which involves the presence in P of many special symbols *. The wild-card * matches any substring. In this case if $P = \alpha * \beta * \cdots$, where $\alpha, \beta, \ldots$ are un-empty strings, then we can implement the wild-card search by first performing a prefix-search for α in $\mathcal{D}$ and then checking brute-forcedly whether P matches the returned strings given the presence of the wild-cards. Of course this approach may be very expensive, especially when α is not a selective prefix and thus many dictionary strings are returned as candidate matches. Nevertheless this puts in evidence how much slow may be in a disk environment a wild-card query if solved with a trivial approach.

1.1.1 Contiguous allocation of strings

A simple trick to circumvent some of the previous limitations is to store the dictionary strings sorted lexicographically and contiguously on disk. This way (pointers) contiguity in A reflects into (string) contiguity in S. This has two main advantages:

- when the binary search is confined to few strings, they will be closely stored both in A and S, so probably they have been buffered by the system in internal memory (*speed*);
- some compression can be applied to contiguous strings in S, because they typically share some prefix (*space*).

Given that S is stored on disk, we can deploy the first observation by blocking strings into groups of B characters each and then *store* a pointer to the first string of each group in A. The sampled strings are denoted by $\mathcal{D}_B \subseteq \mathcal{D}$, and their number n_B is upper bounded by $\frac{N}{B}$ because we pick at most one string per block. Since A has been squeezed to index at most $n_B \leq n$ strings, the search over A must be changed in order to reflect the *two-level structure* given by the array A and the blocks of strings in S. So the idea is to decompose the lexicographic search for Q in a two-stages process: in the first stage, we search for the lexicographic position of Q within the sampled strings of $\mathcal{D}_B$; in the second stage, this position is deployed to identify the block of strings where the lexicographic position of Q lies in, and then the strings of this block are scanned and compared with Q for prefix match. We recall that, in order to implement the prefix search, we have to repeat the above process for the two strings P and $P\#$, so we have proved the following:

THEOREM 1.2 *Prefix search over $\mathcal{D}$ takes $O(\frac{p}{B} \log \frac{N}{B})$ I/Os. Retrieving the strings prefixed by P needs $\frac{N_{occ}}{B}$ I/Os, where N_{occ} is their length. The total space is $N + (1 + w)n_B$ bytes.*

Proof Once the block of strings $A[i, j]$ prefixed by P has been identified, we can report all of them in $O(\frac{N_{occ}}{B})$ I/Os; scanning the contiguous portion of S that contains those strings. The space occupancy comes from the observation that pointers are stored only for the n_B sampled strings. ■

Typically strings are shorter than B, so $\frac{N}{B} \leq n$, hence this solution is faster than the previous one, in addition it can be effectively combined with the technique called *Front-Coding compression* to further lowering the space and I/O-complexities.

1.1.2 Front Coding

Given a sequence of sorted strings is probable that adjacent strings share a common prefix. If ℓ is the number of shared characters, then we can substitute them with a proper variable-length binary encoding of ℓ thus saving some bits with respect to the classic fixed-size encoding based on 4- or 8-bytes. A following chapter will detail some of those encoders, here we introduce a simple one to satisfy the curiosity of the reader. The encoder pads the binary representation of ℓ with 0 until an integer number of bytes is used. The first two bits of the padding (if any, otherwise one more byte is added), are used to encode the number of bytes used for the encoding.[1] This encoding is prefix-free and thus guarantees unique decoding properties; its byte-alignment also ensures speed in current processors.

More efficient encoders are available, anyway this simple proposal ensures to replace the initial $\Theta(\ell \log_2 \sigma)$ bits, representing ℓ characters of the shared prefix, with $O(\log \ell)$ bits of the integer encoding, so resulting advantageous in space. Obviously its final impact depends on the amount of shared characters which, in the case of a dictionary of URLs, can be up to 70%.

Front coding is a *delta*-compression algorithm, which can be easily defined in an incremental way: given a sequence of strings $(s_1, \ldots, s_n)$, it encodes the string s_i using the couple $(\ell_i, \hat{s}_i)$, where ℓ_i is the length of the longest common prefix between s_i and its predecessor s_{i-1} (0 if $i = 1$) and $\hat{s}_i = s_i[\ell_i + 1, |s_i|]$ is the "remaining suffix" of the string s_i. As an example consider the dictionary $\mathcal{D} = \{$ `alcatraz, alcool, alcyone, anacleto, ananas, aster, astral, astronomy` $\}$; its front-coded representation is $(0, \texttt{alcatraz}), (3, \texttt{ool}), (3, \texttt{yone}), (1, \texttt{nacleto}), (3, \texttt{nas}), (1, \texttt{ster}), (3, \texttt{ral}), (4, \texttt{onomy})$.

Decoding a string a pair $(\ell, \hat{s})$ is symmetric, we have to copy ℓ characters from the previous string in the sequence and then append the remaining suffix $\hat{s}$. This takes $O(|s|)$ optimal time and $O(1 + \frac{|s|}{B})$ I/Os, provided that the preceding string is available. In general, the reconstruction of a string s_i may require to scan back the input sequence up to the first string s_1, which is available in its entirety. So we may possibly need to scan $(\hat{s}_1, \ell_1), \ldots, (\hat{s}_{i-1}, \ell_{i-1})$ and reconstruct $s_1, \ldots, s_{i-1}$ in order to decode $(\ell_i, \hat{s}_i)$. Therefore, the time cost to decode s_i might be much higher than the optimal $O(|s_i|)$ cost.[2]

To overcome this drawback, it is typical to apply front-coding to block of strings thus resorting the two-level scheme we introduced in the previous subsection. The idea is to restart the front-coding at the beginning of every block, so the first string of each block is stored *uncompressed*. This has two immediate advantages onto the prefix-search problem: (1) these uncompressed strings are the ones participating in the binary-search process and thus they do not need to be decompressed when compared with Q; (2) each block is compressed individually and thus the scan of its strings for searching Q can be combined with the decompression of these strings without incurring in any slowdown. We call this storage scheme "Front-coding with bucketing", and shortly denote it by FC_B. Figure 1.2 provides a running example in which the strings "alcatraz", "alcyone", "ananas", and "astral" are stored explicitly because they are the first of each block.

As a positive side-effect, this approach reduces the number of sampled strings because it can potentially increase the number of strings stuffed in one disk page: we start from s_1 and we front-compress the strings of $\mathcal{D}$ in order; whenever the compression of a string s_i overflows the current block, it starts a new block where it is stored *uncompressed*. The number of sampled strings lowers from about $\frac{N}{B}$ to about $\frac{FC_B(\mathcal{D})}{B}$ strings, where $FC_B(\mathcal{D})$ is the space required by FC_B to store all the dictionary strings. This impacts positively onto the number of I/Os needed for a prefix search in a obvious manner, given that we execute a binary search over the sampled strings. However space

[1] We are assuming that ℓ can be binary encoded in 30 bits, namely $\ell < 2^{30}$.

[2] A smarter solution would be to reconstruct only the first ℓ characters of the previous strings $s_1, s_2, \ldots, s_{i-1}$ because these are the ones interesting for s_i's reconstruction.

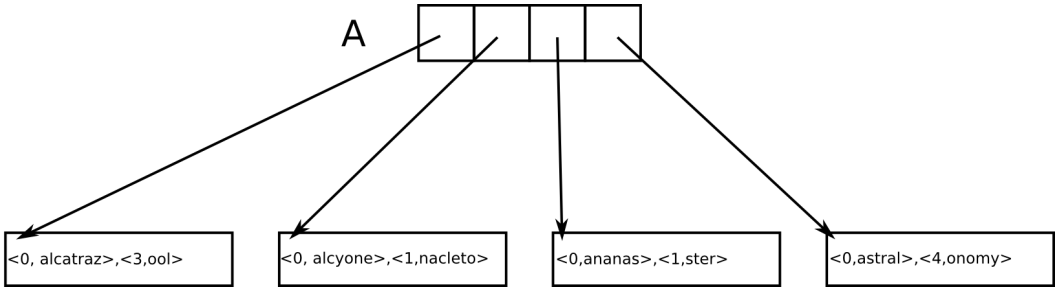

FIGURE 1.2: Two-level indexing of the set of strings $\mathcal{D}$ = { `alcatraz, alcool, alcyone, anacleto, ananas, aster, astral, astronomy`} are compressed with FC_B, where we assumed that each page is able to store two strings.

occupancy increases with respect to $FC(\mathcal{D})$ because $FC_B(\mathcal{D})$ forces the first string of each block to be stored uncompressed; nonetheless, we expect that this increase is negligible because $B \gg 1$.

THEOREM 1.3 *Prefix search over $\mathcal{D}$ takes $O(\frac{p}{B} \log \frac{FC_B(\mathcal{D})}{B}$ I/Os. Retrieving the strings prefixed by P needs $O(\frac{FC_B(\mathcal{D}_{occ})}{B})$ I/Os, where $\mathcal{D}_{occ} \subseteq \mathcal{D}$ is the set of strings in the answer set.*

So, in general, compressing the strings is a good idea because it lowers the space required for storing the strings, and it lowers the number of I/Os. However we must observe that FC-compression might increase the time complexity of the scan of a block from $O(B)$ to $O(B^2)$ because of the decompression of that block. In fact, take the sequence of strings $(a, aa, aaa, \ldots)$ which is front coded as $(0, a), (1, a), (2, a), (3, a), \ldots$. In one disk page we can stuff $\Theta(B)$ such pairs, which represent $\Theta(B)$ strings whose total length is $\sum_{i=0}^{B} \Theta(i) = \Theta(B^2)$ characters. Despite these pathological cases, in practice the space reduction consists of a *constant factor* so the time increase incurred by a block scan is negligible.

Overall this approach introduces a time/space trade-off driven by the block size B. As far as time is concerned we can observe that the longer is B, the better is the compression ratio but the slower is a prefix search because of a longer scan-phase; conversely, the shorter is B, the faster is the scan-phase but the worse is the compression ratio because of a larger number of fully-copied strings. As far as space is concerned, the longer is B, the smaller is the number of copied strings and thus the smaller is the storage space in internal memory needed to index their pointers; conversely, the shorter is B, the larger is the number of pointers thus making probably impossible to fit them in internal memory.

In order to overcome this trade-off we decouple search and compression issues as follows. We notice that the proposed data structure consists of *two* levels: the "upper" level contains references to the *sampled strings* $\mathcal{D}_B$, the "lower" level contains the strings themselves stored in a block-wise fashion. The choice of the algorithms and data structures used in the two levels are "orthogonal" to each other, and thus can be decided independently. It goes without saying that this 2-level scheme for searching-and-storing a dictionary of strings is suitable to be used in a hierarchy of two memory levels, such as the cache and the internal memory. This is typical in Web search, where $\mathcal{D}$ is the dictionary of terms to be searched by users and disk-accesses have to be avoided in order to support each search over $\mathcal{D}$ in few millisecs.

In the next three sections we propose three improvements to the 2-level solution above, two of them regard the first level of the sampled strings, one concerns with the compressed storage of

all dictionary strings. Actually, these proposals have an interest in themselves and thus the reader should not confine their use to the one described in these notes.

1.2 Interpolation search

Until now, we have used binary search over the array A of string pointers. But if $\mathcal{D}_B$ satisfies some statistical properties, there are searching schemes which support faster searches, such as the well known *interpolation search*. In what follows we describe a variant of classic interpolation search which offers some interesting additional properties (details in [3]). For simplicity of presentation we describe the algorithm in terms of a dictionary of integers, knowing that if items are strings, we can still look at them as integers in base σ. So for the prefix-search problem we can pad logically all strings at their end, thus getting to the same length, by using a character that we assume to be smaller than any other alphabet character. Lexicographic order among strings is turned in classic ordering of integers, so that the search for P and $P\#$ can be turned into a search for two proper integers.

So without loss of generality, assume that $\mathcal{D}_B$ is an array of integers $X[1, m] = x_1 \dots x_m$ with $x_i < x_{i+1}$ and $m = n_B$. We evenly subdivide the range $[x_1, x_m]$ into m bins $B_1, \dots, B_m$ (so we consider as many bins as integers in X), each bin representing a contiguous range of integers having length $b = \frac{x_m - x_1 + 1}{m}$. Specifically $B_i = [x_1 + (i-1)b, x_1 + ib)$. In order to guarantee the constant-time access to these bins we need to keep an additional array, say $I[1, m]$, such that $I[i]$ points to the first and last item of B_i in X.

Figure 1.3 reports an example where $m = 12$, $x_1 = 1$ and $x_{12} = 36$ and thus the bin length is $b = 3$.

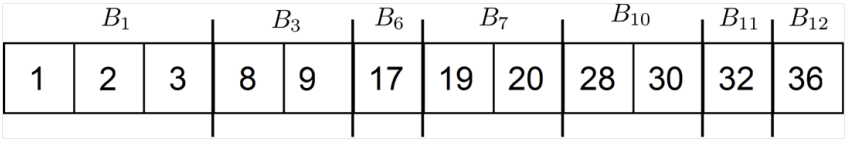

FIGURE 1.3: An example of use of interpolation search over an itemset of size 12. The bins are separated by bars; some bins, such as B_4 and B_8, are empty.

The algorithm searches for an integer y in two steps. In the first step it calculates j, the index of the candidate bin B_j where y could occur: $j = \lfloor \frac{y - x_1}{b} \rfloor + 1$. In the second step, it determines via $I[j]$ the sub-array of X which stores B_j and it does a binary search over it for y, thus taking $O(\log |B_i|) = O(\log b)$ time. The value of b depends on the magnitude of the integers present in the indexed dictionary. Surprisingly enough, we can get a better bound which takes into account the distribution of the integers of X in the range $[x_1, x_m]$.

THEOREM 1.4 *We can search for an integer in a dictionary of size m taking $O(\log \Delta)$ time in the worst case, where Δ is the ratio between the maximum and the minimum gap between two consecutive integers of the input dictionary. The extra space is $O(m)$.*

Proof Correctness is immediate. For the time complexity, we observe that the maximum of a series of integers is at least as large as their mean. Here we take as those integers the gaps $x_i - x_{i-1}$,

and write:

$$\max_{i=2\ldots m}(x_i - x_{i-1}) \geq \frac{\sum_{i=2}^{m} x_i - x_{i-1}}{m-1} \geq \frac{x_m - x_1 + 1}{m} = b \tag{1.1}$$

The last inequality comes from the following arithmetic property: $\frac{a'}{a''} \geq \frac{a'+1}{a''+1}$ whenever $a' \geq a''$, which can be easily proved by solving it.

Another useful observation concerns with the maximum number of integers that can belong to any bin. Since integers of X are spaced apart by $s = \min_{i=2,\ldots,m}(x_i - x_{i-1})$ units, every bin contains no more than b/s integers.

Recalling the definition of Δ, and the two previous observations, we can thus write:

$$|B_i| \leq \frac{b}{s} \leq \frac{\max_{i=2\ldots m}(x_i - x_{i-1})}{\min_{i=2,\ldots,m}(x_i - x_{i-1})} = \Delta$$

.

So the theorem follows due to the binary search performed within B_i. Space occupancy is optimal and equal to $O(m)$ because of the arrays $X[1,m]$ and $I[1,m]$. ∎

We note the following interesting properties of the proposed algorithm:

- The algorithm is oblivious to the value of Δ, although its complexity can be written in terms of this value.
- The worst-case search time is $O(\log m)$, when the whole X ends up in a single bin, and thus the precise bound should be $O(\log \min\{\Delta, m\})$. So it cannot be worst than the binary search.
- The space occupancy is $O(m)$ which is optimal asymptotically; however, it has to be notice that binary search is in-place, whereas interpolation search needs the extra array $I[1,m]$.
- The algorithm reproduces the $O(\log \log m)$ time performance of classic interpolation search on data drawn independently from the uniform distribution, as shown in the following lemma. We observe that, uniform distribution of the X's integers is uncommon in practice, nevertheless we can artificially enforce it by selecting a random permutation $\pi : U \longrightarrow U$ and shuffling X according to π before building the proposed data structure. Care must be taken at query time since we search not y but its permuted image $\pi(y)$ in $\pi(X)$. This way the query performance proved below holds with high probability whichever is the indexed set X. For the choice of π we refer the reader to [6].

LEMMA 1.1 If the m integers are drawn uniformly at random from $[1,U]$, the proposed algorithm takes $O(\lg \lg m)$ time with high probability.

Proof Say integers are uniformly distributed over $[0, U-1]$. As in bucket sort, every bucket B_i contains $O(1)$ integers on average. But we wish to obtain bounds with high probability. So let us assume to partition the integers in $r = \frac{m}{2 \log m}$ ranges. We have the probability $1/r$ that an integer belongs to a given range. The probability that a given range does not contain any integer is $(1 - \frac{1}{r})^m = (1 - \frac{2 \log m}{m})^m = O(e^{-2 \log m}) = O(1/m^2)$. So the probability that at least one range remains empty is smaller than $O(1/m)$; or, equivalently, with high probability every range contains at least one integer.

If this occurs with high probability, the maximum distance between two adjacent integers must be smaller than twice the range's length: namely $\max_i(x_i - x_{i-1}) \leq 2U/r = O(\frac{U \log m}{m})$.

Let us now take $r' = \Theta(m \log m)$ ranges, similarly as above we can prove that every adjacent pair of ranges contains at most one integer with high probability. Therefore if a range contains an integer, its two adjacent ranges (on the left and on the right) are empty with high probability. Thus we can lower bound the minimum gap with the length of one range: $\min_i(x_i - x_{i-1}) \geq U/r' = \Theta(\frac{U}{m \log m})$. Taking the ratio between the minimum and the maximum gap, we get the desired $\Delta = O(\log^2 m)$. ■

If this algorithm is applied to our string context, and strings are uniformly distributed, the number of I/Os required to prefix-search P in the dictionary $\mathcal{D}$ is $O(\frac{p}{B} \log \log \frac{N}{B})$. This is an exponential reduction in the search time performance according to the dictionary length.

1.3 Locality-preserving front coding

This is an elegant variant of front coding which provides a controlled trade-off between space occupancy and time to decode one string [2]. The key idea is simple, and thus easily implementable, but proving its guaranteed bounds is challenging. We can state the underlying algorithmic idea as follows: *a string is front-coded only if its decoding time is proportional to its length, otherwise it is written uncompressed*. The outcome in time complexity is clear: we compress only if decoding is optimal. But what appears surprising is that, even if we concentrated on the time-optimality of decoding, its "constant of proportionality" controls also the space occupancy of the compressed strings. It seems magic, indeed it is!

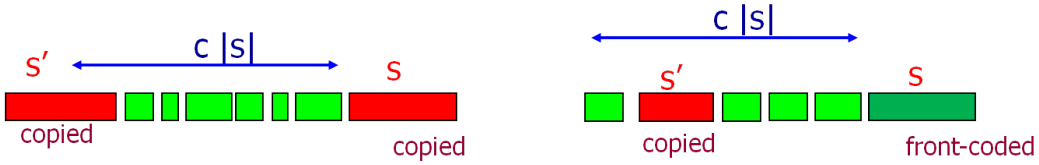

FIGURE 1.4: The two cases occurring in LPFC. Red rectangles are copied strings, green rectangles are front-coded strings.

Formally, suppose that we have front-coded the first $i-1$ strings $(s_1, \ldots, s_{i-1})$ into the compressed sequence $\mathcal{F} = (0, \hat{s}_1), (\ell_2, \hat{s}_2), \ldots, (\ell_{i-1}, \hat{s}_{i-1})$. We want to compress s_i so we scan backward at most $c|s_i|$ characters of $\mathcal{F}$ to check whether these characters are enough to reconstruct s_i. This actually means that an uncompressed string is included in those characters, because we have available the first character for s_i. If so, we front-compress s_i into $(\ell_i, \hat{s}_i)$; otherwise s_i is copied uncompressed in $\mathcal{F}$ outputting the pair $(0, s_i)$. The key difficulty here is to show that the strings which are left uncompressed, and were instead compressed by the classic front-coding scheme, have a length that can be controlled by means of the parameter c as the following theorem shows:

THEOREM 1.5 *Locality-preserving front coding takes at most* $(1+\epsilon)FC(\mathcal{D})$ *space, and supports the decoding of any dictionary string* s_i *in* $O(\frac{|s_i|}{\epsilon B})$ *optimal I/Os.*

Proof We call any uncompressed string s, a *copied* string, and denote the $c|s|$ characters explored during the backward check as the *left extent* of s. Notice that if s is a copied string, there can be

no copied string preceding s and beginning in its left extent (otherwise it would have been front-coded). Moreover, the copied string that precedes S may *end* within s's left extent. For the sake of presentation we call *FC-characters* the ones belonging to the output suffix $\hat{s}$ of a front-coded string s.

Clearly the space occupied by the front-coded strings is upper bounded by $FC(\mathcal{D})$. We wish to show that the space occupied by the copied strings, which were possibly compressed by the classic front-coding but are left uncompressed here, sums up to $\epsilon\ FC(\mathcal{D})$, where ϵ is a parameter depending on c and to be determined below.

We consider two cases for the copied strings depending on the amount of FC-characters that lie between two consecutive occurrences of them. The first case is called *uncrowded* and occurs when that number of FC-characters is at least $\frac{c|s|}{2}$; the second case is called *crowded*, and occurs when that number of FC-characters is at most $\frac{c|s|}{2}$. Figure 1.5 provides an example which clearly shows that if the copied string s is crowded then $|s'| \geq c|s|/2$. In fact, s' starts before the left extent of s but ends within the last $c|s|/2$ characters of that extent. Since the extent is $c|s|$ characters long, the above observation follows. If s is uncrowded, then it is preceded by at least $c|s|/2$ characters of front-coded strings (FC-characters).

FIGURE 1.5: The two cases occurring in LPFC. The green rectangles denote the front-coded strings, and thus their FC-characters, the red rectangles denote the two consecutive copied strings.

We are now ready to bound the total length of copied strings. We partition them into chains composed by one uncrowded copied-string followed by the maximal sequence of crowded copied-strings. In what follows we prove that the total number of characters in each chain is proportional to the length of its first copied-string, namely the uncrowded one. Precisely, consider the chain $w_1 w_2 \cdots w_x$ of consecutive copied strings, where w_1 is uncrowded and the following w_is are crowded. Take any crowded w_i. By the observation above, we have that $|w_{i-1}| \geq c|w_i|/2$ or, equivalently, $|w_i| \leq 2|w_{i-1}|/c = \cdots = (2/c)^{i-1}|w_1|$. So if $c > 2$ the crowded copied strings shrink by a constant factor. We have $\sum_i |w_i| = |w_1| + \sum_{i>1} |w_i| \leq |w_1| + \sum_{i>1}(2/c)^{i-1}|w_1| = |w_1| \sum_{i\geq 0}(2/c)^i < \frac{c|w_1|}{c-2}$.

Finally, since w_1 is uncrowded, it is preceded by at least $c|w_1|/2$ FC-characters (see above). The total number of these FC-characters is bounded by $FC(\mathcal{D})$, so we can upper bound the total length of the uncrowded strings by $(2/c)FC(\mathcal{D})$. By plugging this into the previous bound on the total length of the chains, we get $\frac{c}{c-2} \times \frac{2FC(\mathcal{D})}{c} = \frac{2}{c-2}FC(\mathcal{D})$. The theorem follows by setting $\epsilon = \frac{2}{c-2}$. ■

So locality-preserving front coding (shortly LPFC) is a compressed storage scheme for strings that can substitute their plain storage without introducing any asymptotic slowdown in the accesses to the compressed strings. In this sense it can be considered as a sort of *space booster* for any string indexing technique.

The two-level indexing data-structure described in the previous sections can benefit of LPFC as follows. We can use A to point to the copied strings of LPFC (which are uncompressed). This way the buckets delimited by these strings have variable length, but any string can be decompressed in

optimal time and I/Os (cfr. previous observation about the $\Theta(B^2)$ size of a bucket in classic FC_B). So the bounds are the ones stated in Theorem 1.3 but without the pathological cases commented next to its proof. This way the scanning of a bucket, identified by the binary-search step takes $O(1)$ I/O and time proportional to the returned strings, and hence it is optimal.

The remaining question is therefore how to speed-up the search over the array A. We foresee two main limitations: (i) the binary-search step has time complexity depending on N or n, (ii) if the pointed strings do not fit within the internal-memory space allocated by the programmer, or available in cache, then the binary-search step incurs many I/Os, and this might be expensive. In the next sections we propose a trie-based approach that takes full-advantage of LPFC by overcoming these limitations, resulting efficient in time, I/Os and space.

1.4 Compacted Trie

We already talked about tries in Chapter **??**, here we dig further into their properties as efficient data structures for string searching. In our context, the trie is used for the indexing of the sampled strings $\mathcal{D}_B$ in internal memory. This induces a speed up in the first stage of the prefix search from $O(\log(N/B))$ to $O(p)$ time, thus resulting surprisingly independent of the dictionary size. The reason is the power of the RAM model which allows to manage and address memory-cells of $O(\log N)$ bits in constant time.

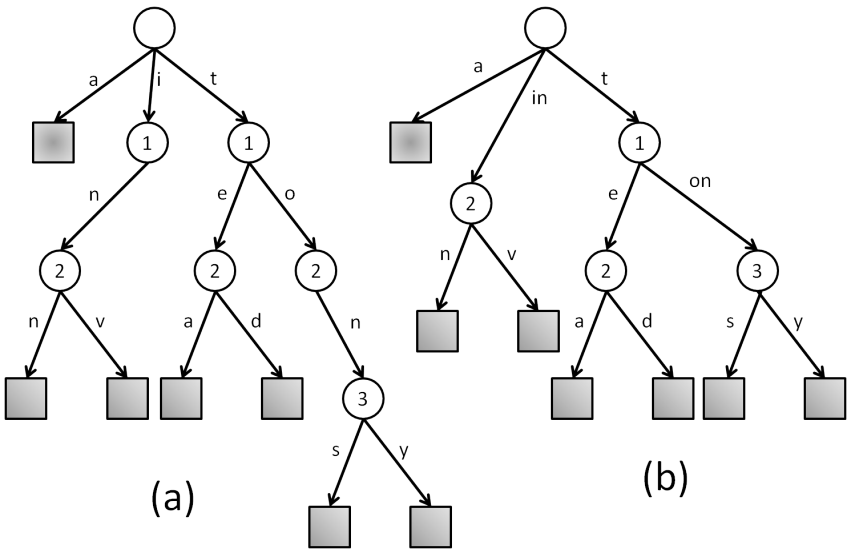

FIGURE 1.6: An example of uncompacted trie (a) and compacted trie (b) for $n = 7$ strings. The integer showed in each internal node u denotes the length of the string spelled out by u. In the case of uncompacted tries they are useless because they correspond to u's depth. Edge labels in compacted tries are substrings of variable length but they can be represented in $O(1)$ space with triples of integers: e.g. `on` could be encoded as $\langle 6, 2, 3\rangle$, since the 6-th string `tons` includes `on` from position 2 to position 3.

A trie is a multi-way tree whose edges are labeled by characters of the indexed strings. An internal

node u is associated with a string $s[u]$ which is indeed a *prefix* of a dictionary string. String $s[u]$ is obtained by concatenating the characters found on the downward path that connects the trie's root with the node u. A leaf is associated with a dictionary string. All leaves which descend from a node u are prefixed by $s[u]$. The trie has n leaves and at most N nodes, one per string character.[3] Figure 1.6 provides an illustrative example of a trie built over 6 strings. This form of trie is commonly called *uncompacted* because it can have *unary paths*, such as the one leading to string `inn`.[4]

If we want to check if a string P prefixes some dictionary string, we have just to check if there is a downward path spelling out P. All leaves descending from the reached node provide the correct answer to our prefix search. So tries do not need the reduction to the lexicographic search operation, introduced for the binary-search approach.

A big issue is how to efficiently find the "edge to follow" during the downward traversal of the trie, because this impacts onto the overall efficiency of the pattern search. The efficiency of this step hinges on a proper storage of the edges (and their labeling characters) outgoing from a node. The simplest data structure that does the job is the *linked list*. Its space requirement is optimal, namely proportional to the number of outgoing edges, but it incurs in a $O(\sigma)$ cost per traversed node. The result would be a prefix search taking $O(p\ \sigma)$ time in the worst case, which is too much for large alphabets. If we store the branching characters (and their edges) into a sorted array, then we could binary search it taking $O(\log \sigma)$ time per node. A faster approach consists of using a full-sized array of σ entries, the un-empty entries (namely the ones for which the pointer in not null) are the entries corresponding to the existing branching characters. In this case the time to branch out of a node is $O(1)$ and thus $O(p)$ time is the cost for searching the pattern Q. But the space occupancy of the trie grows up to $O(N\sigma)$, which may be unacceptably high for large alphabets. The best approach consists of resorting a *perfect hash table*, which stores just the existing branching characters and their associated pointers. This guarantees $O(1)$ branching time in the worst-case and optimal space occupancy, thus combining the best of the two previous solutions. For details about perfect hashes we refer the reader to Chapter **??**.

THEOREM 1.6 *The uncompacted trie solves the prefix-search problem in $O(p+n_{occ})$ time and $O(p + n_{occ}/B)$ I/Os, where n_{occ} is the number of strings prefixed by P. The retrieval of those strings prefixed by P takes $O(N_{occ})$ time, and it takes $O(N_{occ}/B)$ I/Os provided that leaves and strings are stored contiguously and alphabetically sorted on disk. The trie consists of at most N nodes, exactly n leaves, and thus takes $O(N)$ space. The retrieval of the result strings takes $O(N_{occ})$ time and $O(N_{occ}/B)$ I/Os, where N_{occ} is the total length of the retrieved strings.*

Proof Let u be the node such that $s[u] = P$. All strings descending from u are prefixed by P, and they can be visualized by visiting the subtree rooted in u. The I/O-complexity of the traversal is still $O(p)$ because of the jumps among trie nodes. The retrieval of the n_{occ} leaves descending from the node spelling Q takes optimal $O(n_{occ}/B)$ I/Os because we can assume that trie leaves are stored contiguously from left-to-right on disk. Notice that we have identified the strings (leaves) prefixed by Q but, in order to display them, we still need to retrieve them, this takes additional $O(N_{occ}/B)$ I/Os provided that the indexed strings are stored contiguously on disk. This is $O(n_{occ}/B)$ I/Os if we are interested only in the string pointers/IDs, provided that every internal node keeps a pointer to its leftmost descending leaf and all leaves are stored contiguously on disk. (These are the main reasons

[3] We say "at most" because some paths (prefixes) can be shared among several strings.

[4] The trie cannot index strings which are one the prefix of the other. In fact the former string would end up into an internal node. To avoid this case, each string is extended with a special character which is not present in the alphabet and is typically denoted by \$.

for keeping the pointers in the leaves of the uncompacted trie, which anyway stores the strings in its edges, and thus could allow to retrieve them but with more I/Os because of the difficulty to pack arbitrary trees on disk.) ■

A Trie can be wasteful in space if there are long strings with a short common prefix: this would induce a significant number of unary nodes. We can save space by *contracting* the unary paths into one single edge. This way edge labels become (possibly long) sub-strings rather than characters, and the resulting trie is named *compacted*. Figure 1.7 (left) shows an example of compacted trie. It is evident that each edge-label is a substring of a dictionary string, say $s[i, j]$, so it can be represented via a triple $\langle s, i, j\rangle$ (see also Figure 1.6). Given that each node is at least binary, the number of internal nodes and edges is $O(n)$. So the total space required by a compacted trie is $O(n)$ too.

Prefix searching is implemented similarly as done for uncompacted tries. The difference is that it alternates character-branches out of internal nodes, and sub-string matches with edge labels. If the edges spurring from the internal nodes are again implemented with perfect hast tables, we get:

THEOREM 1.7 *The compacted trie solves the prefix-search problem in $O(p + n_{occ})$ time and $O(p + n_{occ}/B)$ I/Os, where n_{occ} is the number of strings prefixed by P. The retrieval of those strings prefixed by P takes $O(N_{occ})$ time, and and it takes $O(N_{occ}/B)$ I/Os provided that leaves and strings are stored contiguously and alphabetically sorted on disk. The compacted trie consists of $O(n)$ nodes, and thus its storage takes $O(n)$ space. It goes without saying that the trie needs also the storage of the dictionary strings to resolve its edge labels, hence additional N space.*

At this point an attentive reader can realize that the compacted trie can be used also to search for the lexicographic position of a string Q among the indexed strings. It is enough to percolate a downward path spelling Q as much as possible until a mismatch character is encountered. This character can then be deployed to determine the lexicographic position of Q, depending on whether the percolation stopped in the middle of an edge or in a trie node. So the compacted trie is an interesting substitute for the array A in our two-level indexing structure and could be used to support the search for the candidate bucket where the string Q occurs in, taking $O(p)$ time in the worst case. Since each traversed edge can induce one I/O, to fetch its labeling substring to be compared with the corresponding one in Q, we point out that this approach is efficient if the trie and its indexed strings can be fit in internal memory. Otherwise it presents two main drawbacks: the linear dependance of the I/Os on the pattern length p, and the space dependance on the block-size B (influencing the the sampling) and the length of the sampled strings.

The *Patricia Trie* solves the former problem, whereas its combination with the LPFC solves both of them.

1.5 Patricia Trie

A Patricia Trie built on a string dictionary is a compacted Trie in which the edge labels consist just of their initial *single characters*, and the internal nodes are labeled with integers denoting the *lengths* of the associated strings. Figure 1.7 illustrates how to convert a Compacted Trie (left) into a Patricia Trie (right).

Even if the Patricia Trie strips out some information from the Compacted Trie, it is still able to support the search for the lexicographic position of a pattern P among a (sorted) sequence of strings, with the significant advantage (discussed below) that this search needs to access only one single string, and hence execute typically one I/O instead of the p I/Os potentially incurred by the edge-resolution step in compacted tries. This algorithm is called *blind search* in the literature [4].

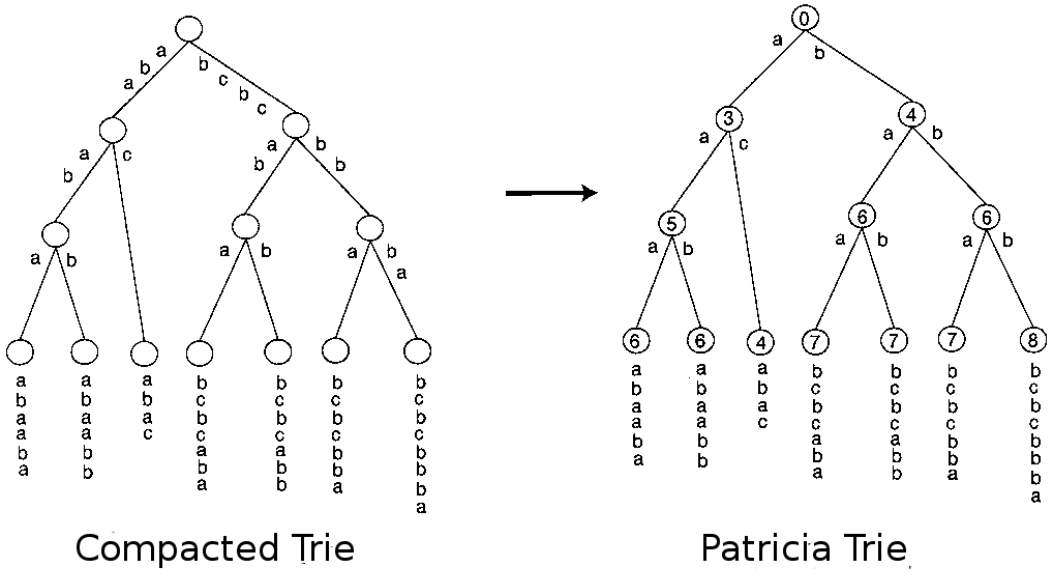

FIGURE 1.7: An example of Compacted Trie and the corresponding Patricia Trie.

It is a little bit more complicated than the prefix-search in classic tries, because of the presence of only one character per edge label, and in fact it consists of three stages:

- Trace a downward path in the Patricia Trie to locate a leaf l which points to an interesting string of the indexed dictionary. This string does not necessarily identify P's lexicographic position in the dictionary (which is our goal), but it provides *enough information* to find that position in the second stage. The retrieval of the interesting leaf l is done by traversing the Patricia Trie from the root and comparing the characters of P with the single characters which label the traversed edges until either a leaf is reached or no further branching is possible. In this last case, we choose l to be any descendant leaf from the last traversed node.
- Compare P against the string pointed by leaf l, in order to determine their longest common prefix. Let ℓ be the length of this shared prefix, then it is possible to prove that (see [4]) the leaf l stores one of the strings indexed by the Patricia Trie that shares the longest common prefix with P. Call s this pointed string. The length ℓ and the two mismatch characters $P[\ell + 1]$ and $s[\ell + 1]$ are then used to find the lexicographic position of P among the strings stored in the Patricia Trie.
- First the Patricia trie is traversed upward from l to determine the edge $e = (u, v)$ where the mismatch character $s[\ell + 1]$ lies; this is easy because each node on the upward path stores an integer that denotes the length of the corresponding prefix of s, so that we have $|s[u]| < \ell \leq |s[v]|$. If $s[\ell + 1]$ is a branching character (i.e. $\ell = |s[u]|$), then we determine the lexicographic position of $P[\ell + 1]$ among the branching characters of node u. Say this is the i-th child of u, the lexicographic position of P is therefore to the immediate left of the subtree descending from this child. Otherwise (i.e. $\ell > |s[u]|$), the character $s[\ell + 1]$ lies within e, so the lexicographic position of P is to the immediate right of the subtree descending from e, if $P[\ell + 1] > s[\ell + 1]$, otherwise it is to the immediate left of that subtree.

A running example is illustrated in Figure 1.8.

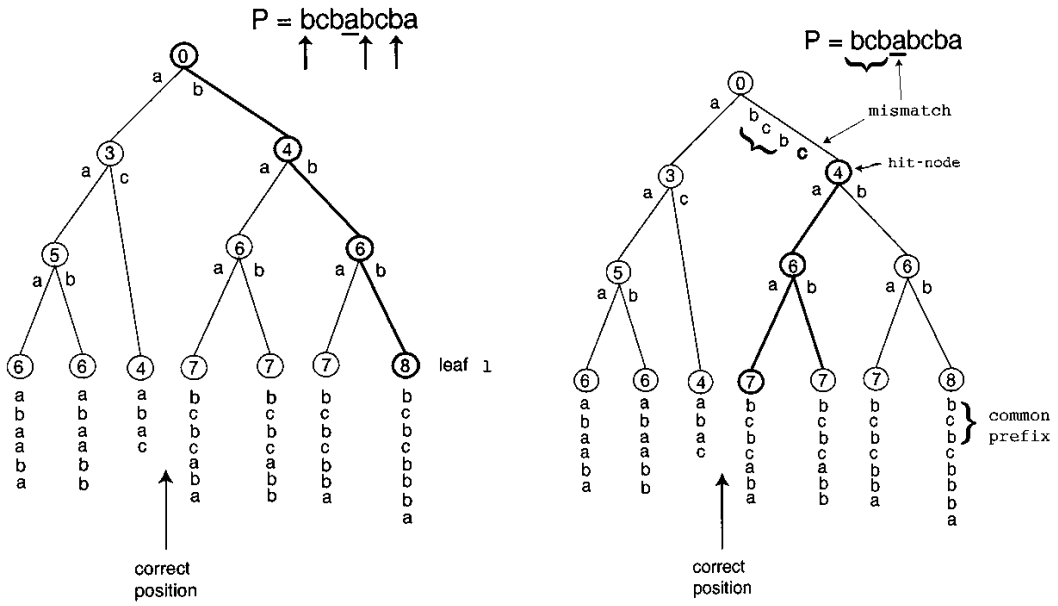

FIGURE 1.8: An example of the first (left) and second (right) stages of the blind search for P in a dictionary of 7 strings.

In order to understand why the algorithm is correct, let us take the path spelling out the string $P[1, \ell]$. We have two cases, either we reached an internal node u such that $|s[u]| = \ell$ or we are in the middle of an edge (u, v), where $|s[u]| < \ell < |s[v]|$. In the former case, all strings descending from u are the ones in the dictionary which share ℓ characters with the pattern, and this is the lcp. The correct lexicographic position therefore falls among them or is adjacent to them, and thus it can be found by looking at the branching characters of the edges outgoing from the node u. This is correctly done also by the blind search that surely stops at u, computes ℓ and finally determines the correct position of P by comparing u's branching characters against $P[\ell + 1]$.

In the latter case the blind search reaches v by skipping the mismatch character on (u, v), and possibly goes further down in the trie because of the possible match between branching characters and further characters of P. Eventually a leaf descending from v is taken, and thus ℓ is computed correctly given that all leaves descending from v share ℓ characters with P. So the backward traversal executed in the second stage of the Blind search reaches correctly the edge (u, v), which is above the selected leaf. There we deploy the mismatch character which allows to choose the correct lexicographic position of P which is either to the left of the leaves descending from v or to their right. Indeed all those leaves share $|s[v]| > \ell$ characters, and thus P falls adjacent to them, either to their left or to their right. The choice depends on the comparison between the two characters $P[\ell + 1]$ and $s[v][\ell + 1]$.

The blind search has excellent performance:

THEOREM 1.8 *A Patricia trie takes $O(n)$ space, hence $O(1)$ space per indexed string (in-*

dependent, therefore, of its length). The blind search for a pattern $P[1, p]$ requires $O(p)$ time to traverse the trie's structure (downward and upward), and $O(p/B)$ I/Os to compare the single string (possibly residing on disk) identified by the blind search. It returns the lexicographic position of P among the indexed strings. By searching for P and P#, as done in suffix arrays, the blind search determines the range of indexed strings prefixed by P, if any.

This theorem states that if $n < M$ then we can index in internal memory the whole dictionary, and thus build the Patricia trie over all dictionary strings and stuff it in the internal memory of our computer. The dictionary strings are stored on disk. The prefix search for a pattern P takes in $O(p)$ time and $O(p/B)$ I/Os. The total required space is the one needed to store the strings, and thus it is $O(N)$.

If we wish to compress the dictionary strings, then we need to resort front-coding. More precisely, we combine the Patricia Trie and LPFC as follows. We fit in the internal memory the Patricia trie of the dictionary $\mathcal{D}$, and store on disk the locality-preserving front coding of the dictionary strings. The two traversals of the Patricia trie take $O(p)$ time and no I/Os (Theorem 1.8), because use information stored in the Patricia trie and thus available in internal memory. Conversely the computation of the lcp takes $O(|s|/B + p/B)$ I/Os, because it needs to decode from its LPFC-representation (Theorem 1.5) the string s selected by the blind search and it also needs to compare s against P to compute their lcp. These information allow to identify the lexicographic position of P among the leaves of the Patricia trie.

THEOREM 1.9 *The data structure composed of the Patricia Trie as the index in internal memory ("upper level") and the LPFC for storing the strings on disk ("lower level") requires $O(n)$ space in memory and $O((1+\epsilon)FC(\mathcal{D}))$ space on disk. Furthermore, a prefix search for P requires $O(\frac{p}{B} + \frac{|s|}{B\epsilon})$ I/Os, where s is the "interesting string" determined in the first stage of the Blind search. The retrieval of the prefixed strings takes $O(\frac{(1+\epsilon)FC(\mathcal{D}_{occ})}{B})$ I/Os, where $\mathcal{D}_{occ} \subseteq \mathcal{D}$ is the set of returned strings.*

In the case that $n = \Omega(M)$, we cannot index in the internal-memory Patricia trie the whole dictionary, so we have to resort the bucketing strategy over the strings stored on disk and index in the Patricia trie only a sample of them. If $N/B = O(M)$ we can index in internal memory the first string of every bucket and thus be able to prefix-search P within the bounds stated in Theorem 1.9, by adding just one I/O due to the scanning of the bucket (i.e. disk page) containing the lexicographic position of P. The previous condition can be rewritten as $N = O(MB)$ which is pretty reasonable in practice, given the current values of $M \approx$ 4Gb and $B \approx$ 32Kb, which make $MB \approx$ 128Tb.

1.6 Managing Huge Dictionaries$^{\infty}$

The final question we address in this lecture is: What if $N = \Omega(MB)$? In this case the Patricia trie is too big to be fit in the internal memory of our computer. We can think to store the trie on disk without taking much care on the layout of its nodes among the disk pages. Unfortunately a pattern search could take $\Omega(p)$ I/Os in the two traversals performed by the Blind search. Alternatively, we could incrementally grow a root page and repeatedly add some node not already packed into that page, where the choice of that node might be driven by various criteria that either depend on some access probability or on the node's depth. When the root page contains B nodes, it is written onto disk and the algorithm recursively lays out the rest of the tree. Surprisingly enough, the obtained packing is far from optimality of a factor $\Omega(\frac{\log B}{\log\log B})$, but it is surely within a factor $O(\log B)$ from the optimal [1].

In what follows we describe two distinct optimal approaches to solve the prefix-search over dictionaries of huge size: the first solution is based on a data structure, called the *String B-Tree* [4], which boils down to a B-tree in which the routing table of each node is a Patricia tree; the second solution consists of applying proper *disk layouts of trees* onto the Patricia trie built over the entire dictionary.

1.6.1 String B-Tree

The key idea consists of dividing the big Patricia trie into a set of smaller Patricia tries, each fitting into one disk page. And then linking together all of them in a B-Tree structure. Below we outline a constructive definition of the String B-Tree, for details on this structure and the supported operations we refer the interested reader to the cited literature.

The dictionary strings are stored on disk contiguously and ordered. The pointers to these strings are partitioned into a set of smaller, equally sized chunks $\mathcal{D}_1, \ldots, \mathcal{D}_m$, each including $\Theta(B)$ strings independently of their length. This way, we can index each chunk $\mathcal{D}_i$ with a Patricia Trie that fits into one disk page and embed it into a leaf of the B-Tree. In order to search for P among those set of nodes, we take from each partition $\mathcal{D}_i$ its *first* and *last* (lexicographically speaking) strings s_{if} and s_{il}, defining the set $\mathcal{D}^1 = \left\{s_{1f}, s_{1l}, \ldots, s_{mf}, s_{ml}\right\}$.

Recall that the prefix search for P boils down to the lexicographic search of a pattern Q, properly defined from P. If we search Q within $\mathcal{D}^1$, we can discover one of the following three cases:

1. Q falls before the first or after the last string of $\mathcal{D}$, if $Q < s_{1f}$ or $Q > s_{ml}$.
2. Q falls among the strings of some $\mathcal{D}_i$, and indeed it is $s_{if} < Q < s_{il}$. So the search is continued in the Patricia trie that indexes $\mathcal{D}_i$;
3. Q falls between two chunks, say $\mathcal{D}_i$ and $\mathcal{D}_{i+1}$, and indeed it is $s_{il} < Q < s_{(i+1)f}$. So we found Q's lexicographic position in the whole $\mathcal{D}$, namely it is between these two adjacent chunks.

In order to establish which of the three cases occurs, we need to search efficiently in $\mathcal{D}^1$ for the lexicographic position of Q. Now, if $\mathcal{D}^1$ is small and can be fit in memory, we can build on it a Patricia trie ad we are done. Otherwise we repeat the partition process on $\mathcal{D}^1$ to build a smaller set $\mathcal{D}^2$, in which we sample, as before, two strings every B, so that $|\mathcal{D}^2| = \frac{2|\mathcal{D}^1|}{B}$. We continue this partitioning process for k steps, until it is $|\mathcal{D}^k| = O(B)$ and thus we can fit the Patricia trie built on $\mathcal{D}^k$ within one disk page[5].

We notice that each disk page gets an even number of strings when partitioning $\mathcal{D}^1, \ldots, \mathcal{D}^k$, and to each pair $\left(s_{if}, s_{il}\right)$ we associate a pointer to the block of strings which they delimit in the lower level of this partitioning process. The final result of the process is then a B-Tree over string pointers. The *arity* of the tree is $\Theta(B)$, because we index $\Theta(B)$ strings in each single node. The nodes of the String B-Tree are then stored on disk. The following Figure 1.9 provides an illustrative example for a String B-tree built over 7 strings.

A (prefix) search for the string P in a String B-Tree is simply the traversal of the B-Tree, which executes at each node a lexicographic search of the proper pattern Q in the Patricia trie of that node. This search discovers one of the three cases mentioned above, in particular:

- case 1 can only happen on the root node;
- case 2 implies that we have to follow the node pointer associated to the identified partition.

[5] Actually, we could stop as soon as $\left|\mathcal{D}^k\right| = O(M)$, but we prefer the former to get a standard B-Tree structure.

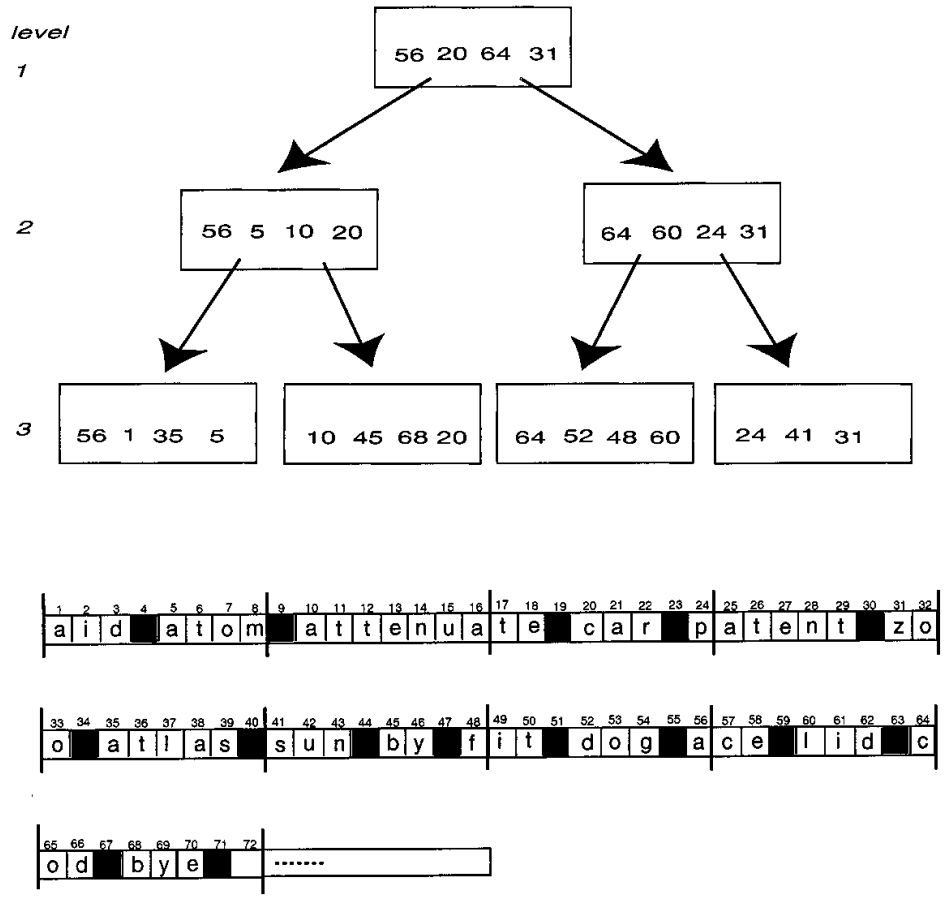

FIGURE 1.9: An example of an String B-tree on built on the suffixes of the strings in $\mathcal{D}$ = {*'ace'*, *'aid'*, *'atlas'*, *'atom'*, *'attenuate'*, *'by'*, *'bye'*, *'car'*, *'cod'*, *'dog'*, *'fit'*, *'lid'*, *'patent'*, *'sun'*, *'zoo'*}. The strings are stored in the B-tree leaves by means of their logical pointers 56, 1, 35, 5, 10, ..., 31. Notice that strings are not sorted on disk, nevertheless sorting improves their I/O-scanning, and indeed our theorems assume an ordered $\mathcal{D}$ on disk.

- case 3 has found the lexicographic position of Q in the dictionary $\mathcal{D}$, so the search in the B-tree stops.

The I/O complexity of the data structure just defined is pretty good: since the arity of the B-Tree is $\Theta(B)$, we have $\Theta(\log_B n)$ levels, so a search traverses $\Theta(\log_B n)$ nodes. Since on each node we need to load the node's page into memory and perform a Blind search over its Patricia trie, we pay $O(1 + \frac{p}{B})$ I/Os, and thus $O(\frac{p}{B} \log_B n)$ I/Os for the overall prefix search of P in the dictionary $\mathcal{D}$.

THEOREM 1.10 *A prefix search in the String B-Tree built over the dictionary $\mathcal{D}$ takes $O(\frac{p}{B} \log_B n + \frac{N_{occ}}{B})$ I/Os, where N_{occ} is the total length of the dictionary strings which are prefixed by P. The data structure occupies $O(\frac{N}{B})$ disk pages and, indeed, strings are stored uncompressed on disk.*

This result is good but not yet *optimal*. The issue that we have to resolve to reach optimality is

pattern rescanning: each time we do a Blind search, we compare Q and one of the strings stored in the currently visited B-Tree node starting from their first character. However, as we go down in the string B-tree we can capitalize on the characters of Q that we have already compared in the upper levels of the B-tree, and thus avoid the rescanning of these characters during the subsequent lcp-computations. So if f characters have been already matched in Q during some previous lcp-computation, the next lcp-computation can compare Q with a dictionary string starting from their $(f+1)$-th character. The pro of this approach is that I/Os turn to be optimal, the cons is that strings have to be stored uncompressed in order to support the efficient access to that $(f+1)$-th character. Working out all the details [4], one can show that:

THEOREM 1.11 *A prefix search in the String B-Tree built over the dictionary $\mathcal{D}$ takes $O(\frac{p+N_{occ}}{B} + \log_B n)$ optimal I/Os, where N_{occ} is the total length of the dictionary strings which are prefixed by P. The data structure occupies $O(\frac{N}{B})$ disk pages and, indeed, strings are stored uncompressed on disk.*

If we want to store the strings compressed on disk, we cannot just plug LPFC in the approach illustrated above, because the decoding of LPFC works only on full strings, and thus it does not support the efficient skip of some characters without wholly decoding the compared string. [2] discusses a sophisticated solution to this problem which gets the I/O-bounds in Theorem 1.11 but in the cache-oblivious model and guaranteeing LPFC-compressed space. We refer the interested reader to that paper for details.

1.6.2 Packing Trees on Disk

We point out that the advantage of finding a good layout for unbalanced trees among disk pages (of size B) may be unexpectedly large, and therefore, must not be underestimated when designing solutions that have to manage large trees on disk. In fact, while balanced trees save a factor $O(\log B)$ when mapped to disk (pack B-node balanced subtrees per page), the mapping of unbalanced trees grows with non uniformity and approaches, in the extreme case of a linear-height tree, a saving factor of $\Theta(B)$ over a naïve memory layout.

This problem is also known in the literature as the *Tree Packing* problem. Its goal is to find an allocation of tree nodes among the disk pages in such a way that the number of I/Os executed for a pattern search is minimized. Minimization may involve either the total number of loaded pages in internal memory (i.e. page faults) , or the number of distinct visited pages (i.e. working-set size). This way we model two extreme situations: the case of a one-page internal memory (i.e. a small buffer), or the case of an unbounded internal memory (i.e. an unbounded buffer). Surprisingly, the optimal solution to the tree packing problem is *independent* of the available buffer size because no disk page is visited twice when page faults are minimized or the working set is minimum. Moreover, the optimal solution shows a nice *decomposability property*: the optimal tree packing forms in turn a tree of disk pages. These two facts allow to restrict our attention to the page-fault minimization problem, and to the design of recursive approaches to the optimal tree decomposition among the disk pages.

In the rest of this section we present two solutions of increasing sophistication and addressing two different scenarios: one in which the goal is to *minimize the maximum number* of page faults executed during a downward root-to-leaf traversal; the other in which the goal is to *minimize the average number* of page faults by assuming an access distribution to the tree leaves, and thus to the possible tree traversals. We briefly mention that both solutions assume that B is known; the literature actually offers cache-oblivious solutions to the tree packing problem, but they are too much sophisticated to be reported in these notes. For details we refer the reader to [1, 5].

Min-Max Algorithm. This solution operates greedily and bottom up over the tree to be packed with

the goal of minimizing the maximum number of page faults executed during a downward traversal which starts from the root of the tree. The tree is assumed to be binary, this is not a restriction for Patricia Tries because it is enough to encode the alphabet characters with binary strings. The algorithm assigns every leaf to its own disk page and the height of this page is set to 1. Working upward, Algorithm 1.1 is applied to each processed node until the root of the tree is reached.

Algorithm 1.1 Min-Max Algorithm over binary trees (general step).

```
Let u be the currently visited node;
if If both children of u have the same page height d then
  if If the total number of nodes in both children's pages is < B then
    Merge the two disk pages and add u;
    Set the height of this new page to d;
  else
    Close off the pages of u's children;
    Create a new page for u and set its height to d + 1;
  end if
else
  Close off the page of u's child with the smaller height;
  If possible, merge the page of the other child with u and leave its height unchanged;
  Otherwise, create a new page for u with height d + 1 and close off the child's page;
end if
```

The final packing may induce a poor page-fill ratio, nonetheless several changes can alleviate this problem in real situations:

1. When a page is closed off, scan its children pages from the smallest to the largest and check whether they can be merged with their parent.
2. Design logical disk pages and pack many of them into one physical disk page; possibly ignore physical page boundaries when placing logical pages onto disk.

THEOREM 1.12 *The Min-Max Algorithm provides a disk-packing of a tree of n nodes and height H such that every root-to-leaf path traverses less than* $1 + \lceil \frac{H}{\sqrt{B}} \rceil + \lceil 2 log_B n \rceil$ *pages.*

Distribution-aware Packing. We assume that it is known an access distribution to the Patricia trie leaves. Since this distribution is often skewed towards some leaves, that are then accessed more frequently than others, the Min-Max algorithm may be significantly inefficient. The following algorithm is based on a Dynamic-Programming scheme, and optimizes the *expected* number of I/Os incurred by any traversal of a root-to-leaf path.

We denote by τ this optimal tree packing (from tree nodes to disk pages), so $\tau(u)$ denotes the disk page to which the tree node u is mapped. Let $w(f)$ be the probability to access a leaf f, we derive a distribution over all other nodes u of the tree by summing up the access probabilities of its descending leaves. We can assume that the tree root r is always mapped to a fixed page $\tau(r) = R$. Consider now the set V of tree nodes that descend from R's nodes but are not themselves in R. We observe that the optimal packing τ induces a tree of disk pages and consequently, if τ is optimal for the current tree T, then τ is optimal for all subtrees T_v rooted in $v \in V$.

This result allows to state a recursive computation for τ that first determines which nodes reside in R, and then continues recursively with all subtrees T_v for which $v \in V$. Dynamic programming

provides an efficient implementation of this idea, based on the following definition: An *i-confined* packing of a tree T is a packing in which the page R contains exactly i nodes (clearly $i \leq B$). Now, in the optimal packing τ, the root page R will contain i^* nodes from the left subtree $T_{left(r)}$ and $(B - i^* - 1)$ nodes from the right subtree $T_{right(r)}$, for some i^*. The consequence is that τ is both an optimal i^*-confined packing for $T_{left(r)}$ and an optimal $(B - i^* - 1)$-confined packing for $T_{right(r)}$. This property is at the basis of the Dynamic-Programming rule which computes $A[v, i]$, for a generic node v and integer $i \leq B$, as the cost of an optimal i-confined packing of the subtree T_v. In the paper [5] the authors showed that $A[v, i]$, for $i > 1$, can be computed as the access probability $w(v)$ plus the minimum among the following three quantities:

1. $A[left(v), i - 1] + w(right(v)) + A[right(v), B]$
2. $w(left(v)) + A[left(v), B] + A[right(v), i - 1]$
3. $\min_{1 \leq j < i-1}\{A[left(v), j] + A[right(v), i - j - 1]\}$

Rule (1) accounts for the (unbalanced) case in which the i-confined packing is obtained by storing $i - 1$ nodes from $T_{left(v)}$ into the v's page; Rule (2) is the symmetric of Rule (1); whereas Rule (3) accounts for the case in which j nodes from $T_{left(v)}$ and $i - j - 1$ nodes from $T_{right(v)}$ are stored into the page of v to form the optimal i-confined packing of T_v. The special case $i = 1$ is given by $A[v, 1] = w(T_v) + A[left(v), B] + A[right(v), B]$.

Algorithm 1.2 deploys these rules to compute the optimal tree packing in $O(nB^2)$ time and $O(nB)$ space.

Algorithm 1.2 Distribution-aware packing of trees on disk.

```
Initialize A[v, i] = w(v), for all leaves v and integers i ≤ B;
while there exist an unmarked node v do
    mark v;
    update A[v, 1] = w(v) + A[left(v), B] + A[right(v), B];
    for i = 2 to B do
        update A[v, i] according to the dyn-prog rule specified in the text.
    end for
end while
```

THEOREM 1.13 *An optimal packing for a f-ary tree of n nodes can be computed in $O(nB^2 \log f)$ time and $O(B \log n)$ space. The packing maps the tree into at most $2\lfloor \frac{n}{B} \rfloor$ disk pages. Optimality is with respect to the* expected *number of I/Os incurred by any root-to-leaf traversal.*

References

[1] Stefan Alstrup, Michael A. Bender, Erik D. Demaine, Martin Farach-Colton, Jan I. Munro, Theis Rauhe, and M. Thorup. *Efficient Tree Layout in a Multilevel Memory Hierarchy*, 2003. Personal Communication, corrected version of a paper appeared in the *European Symposium on Algorithms 2002*.

[2] Michael A. Bender, Martin Farach-Colton, and Bradley C. Kuszmaul. Cache-oblivious string B-trees. In *Procs ACM Symposium on Principles of Database Systems*, pages 223–242, 2006.

[3] Erik D. Demaine, Thouis Jones, and Mihai Pătraşcu. Interpolation search for non-independent data. In *Procs ACM-SIAM Symposium on Discrete algorithms*, pages 529–530, 2004.

[4] Paolo Ferragina and Roberto Grossi. The String B-tree: a new data structure for string search in external memory and its applications. *Journal of the ACM*, 46(2):236–280, 1999.

[5] Joseph Gil and Alon Itai. How to pack trees. *Journal of Algorithms*, 32(2):108–132, 1999.

[6] Michael Luby, Charles Rackoff. How to construct pseudorandom permutations from pseudorandom functions. *SIAM Journal on Computing*, 17, 373–386, 1988.