OPTIMAL AND NEARLY OPTIMAL STATIC WEIGHTED SKIP LISTS

CONRADO MARTÍNEZ AND SALVADOR ROURA

ABSTRACT. We consider the problem of building a static (i.e. no updates are performed) skip list of n elements, given these n elements and the corresponding access probabilities or weights. We develop a dynamic programming algorithm that builds an optimal skip list in the sense that the average access cost is minimized. We also consider nearly optimal skip lists, whose average access cost is not optimal but good enough, and can be built more efficiently than optimal skip lists. Several related issues are also discussed, for instance, other approaches to the construction of nearly optimal skip lists or the construction of optimal skip lists that minimize different kinds of search costs.

1. INTRODUCTION

There are many instances where we have to deal with a *static* data set, i.e. no insertions, deletions or modifications are needed, and therefore it is convenient to organize the information to make the accesses to that information as efficient as possible. Just to mention a few such instances, consider dictionaries, KWIC (key-word-in-context) indexes or applications to data compression and coding theory.

It is customary to assume that the data set consists of n items, each one having a distinct key drawn from some totally ordered set, and that the probabilities of access to the items are given beforehand. This last assumption can be reasonablely satisfied by conducting an empirical study of the relative frequencies of access to the items.

The items can be organized as a binary search tree, as an m-ary search tree, etc. Since the data set is static and each item has a probability of access, the data structures that represent the data set are called *static weighted data structures*. If we denote $C_k(T)$ the cost of an access to the k-th item in the data structure T and p_k denotes the probability of access to that item, the target is to build a data structure T_{opt} such that the average cost per access is minimum. That is,

$$\sum_{1\le k\le n} p_k C_k(T_{\text{opt}}) = \min_T \sum_{1\le k\le n} p_k C_k(T).$$

This research was supported by the ESPRIT BRA Program of the EC under contract no. 7141, project ALCOM II.
The research of the second author has also been supported by a grant from CIRIT (Comissió Interdepartamental de Recerca i Innovació Tecnològica).

Such a data structure is called *optimal*; we can speak thus about optimal binary search trees, optimal m-ary search trees, etc.

The problem can be further generalized if we consider the construction of optimal data structures when we are given $2n+1$ probabilities $p_1, \ldots, p_n$ and $q_0, \ldots, q_n$ where

$$p_k = \text{probability of accessing the item } x_k, \qquad k = 1, \ldots, n;$$
$$q_k = \text{probability of accessing a key in the interval } (\text{key}(x_k), \text{key}(x_{k+1})), \quad k = 0, \ldots, n.$$

We assume in the definition above that $\text{key}(x_1) \leq \text{key}(x_2) \leq \cdots \leq \text{key}(x_n)$, that q_0 is the probability of accessing a key smaller than any of the n keys, and that q_n is the probability of accessing a key larger than any other in the data structure.

In 1971, Knuth [7] showed that an optimal binary search tree can be constructed in time $\mathcal{O}(n^2)$, for the case of general p and q weights, thus beating the more or less obvious $\mathcal{O}(n^3)$ algorithm to construct optimal binary search trees using dynamic programming. The same year, Hu and Tucker [3] showed a way to construct an optimal tree in time $\mathcal{O}(n \log n)$, for the case that all p's are null.

Similar algorithms have been considered for multi-way search trees [2], alphabetic trees [5], split trees [4], etc.

Besides the construction of optimal data structures, the construction of *nearly optimal data structures* has also attracted the interest of researchers, see for instance [9, 10]. As the name indicates, a nearly optimal data structure is not optimal, but its performance is not far from that of its optimal counterpart. There are several reasons why nearly optimal data structures are interesting: building a nearly optimal data structure is usually cheaper both in time and space resources than building an optimal one, their performance does not change a lot if there are slight changes of the access probabilities, and the analysis of their performance is much easier. In fact, it is rather difficult to analyze the performance of the optimal data structures in a direct manner, and hence it is common to get upper bounds on that performance by considering the performance of the nearly optimal data structures [1].

As an indication of the interest of the subject, over one hundred references about optimal and nearly optimal data structures have appeared in the specialized literature during the last 25 years.

This paper is organized as follows. In Section 2 we briefly review *skip lists* and the search algorithm for those data structures; we also introduce the basic definitions and notation for the rest of the paper. Before going into the main bulk of the paper, we present an information-theoretic lower bound for the average cost in skip lists in Section 3. Section 4 is devoted to the description of the algorithm to construct optimal skip lists. It follows the dynamic programming approach used in previous works. The first solution is then significantly improved—one order of magnitude— thanks to the monotonicity properties of the subdivision of the problem into smaller subproblems; analogous properties were first noticed by Knuth in his work about optimal binary search trees [7]. Yao [16] describes a nice generalization, showing sufficient conditions

that allow an $\mathcal{O}(n)$ speedup of the dynamic programming approach. Another example where these monotonicity properties can be applied is in the construction of optimal multi-way search trees [2].

We then study in Section 5 the construction of nearly optimal skip lists, where we present one of the possible approaches to this problem. In turn, the analysis of the performance of such nearly optimal skip lists provides upper bounds for the performance of optimal skip lists.

Several different approaches to the construction of nearly optimal skip lists as well as the construction of optimal skip lists that minimize other kinds of search costs are the topic of Section 6. Finally, in Section 7 we discuss future work in this area.

The full implementation in C of the algorithms may be found in the Appendixes at the end.

2. Skip Lists

Skip lists were introduced by W. Pugh in 1990 [15]. They can be used to represent abstract data types such as dictionaries and ordered lists. Skip list algorithms are very simple to implement, providing a significant constant factor improvement over balanced and self-adjusting trees. Several versions of Pugh's *probabilistic* skip lists, such as the *deterministic skip lists* of Munro, Papadakis and Sedgewick [12], have been introduced hereafter.

Probabilistic skip lists are space efficient, requiring an average of 2 (or less) pointers per item and no balance, priority or weight information. For the analysis of the performance of probabilistic skip lists, see for instance [6, 13].

We shall assume that the items to be stored in a skip list have different keys drawn from some totally ordered set. We sketch here an informal description of the probabilistic skip lists; more details can be found in Pugh's article [15].

A skip list representing the set of items $X = \{x_1, x_2, \ldots, x_n\}$ consists of a collection of sorted linked lists, called *level* 1, *level* 2, ... We assume w.l.o.g. that $\text{key}(x_1) < \text{key}(x_2) < \cdots < \text{key}(x_n)$. Each item x_i is stored in one or more of these linked lists. A positive integer denoted $\text{level}(x)$ is associated to each item x. If the level of an item x is l, that means that x belongs to the linked lists level $1, 2, \ldots$ up to l.

In the case of probabilistic skip lists, the level of each item x is independently given by a geometrically distributed random variable with parameter p. More precisely,

$$\Pr\{\text{level}(x) = l\} = pq^{l-1}, \quad l \geq 1, \quad 0 < q = 1 - p < 1.$$

To implement a skip list S, we allocate a node for each item in the skip list. Each node contains an item and as many pointers as the level of the item indicates. These pointers, called *forward* pointers, point to the successors of the item in each list. If $1 \leq l \leq \text{level}(x)$, $\text{forward}(x, l)$ denotes the succesor at level l of the item x. We also allocate a header node, called $\text{header}(S)$, with pointers to the first item of each linked list. The header is assumed to be the zero-th item in the skip list and stores

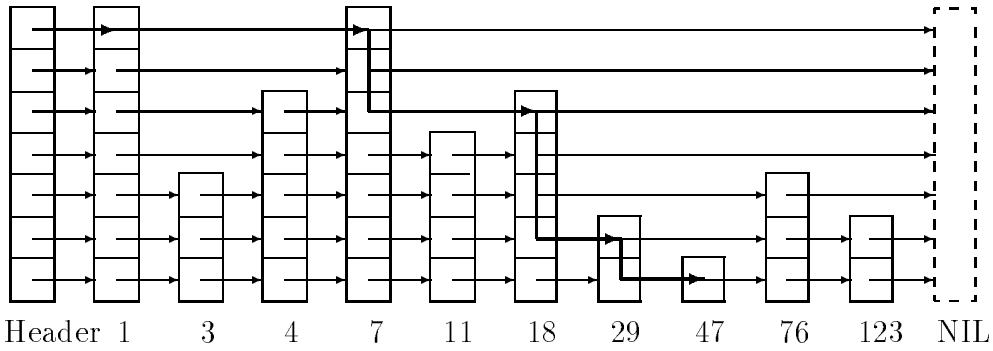

FIGURE 1. A skip list of 10 items and the search path to the interval (47,76].

a dummy key $-\infty$. Finally, we follow the convention that there exists an $(n+1)$-th item, called **NIL**, pointed to by the last item of each linked list and storing a key greater than any legal key: $\text{key}(\textbf{NIL}) = \infty$.

The number of linked lists in the skip list S or, equivalently, the maximum level among the levels of its items is called the *height* of S and denoted $\text{H}(S)$. If S were empty then we assume that $\text{H}(S) = 0$. By convention, $\text{level}(\textbf{NIL}) = \text{H}(S) + 1 = \text{level}(\text{header}(S)) + 1$.

Given S and a key a, the search procedure (see Figure 2) returns the unique item x in S such that $\text{key}(x) < a \leq \text{key}(\text{forward}(x, 1))$. The algorithm keeps a current item/level (x, l), which is initially set to $(\text{header}(S), \text{level}(S))$. The main loop mantains the invariant "$\text{key}(x) < a \leq \text{key}(y)$," where y is the first item to the right of x such that its level is greater than l. The loop finishes when the current level l is 0, since at that point it holds $\text{key}(x) < a \leq \text{key}(\text{forward}(x, 1))$. The current item/level is changed in each iteration according to the following rules: if the key of the successor of the current item/level is smaller than a, x is set to its successor at level l; otherwise, the current level is decremented.

We will use the number of key comparisons to measure the cost of a search. If we are looking for the key of x_{k+1} or a key in the range $(\text{key}(x_k), \text{key}(x_{k+1}))$, the main loop iterates until x points to x_k; an additional comparison suffices to know whether the search key was present or not. The cost of a successful search for x_{k+1} or *path length* of x_k, will be denoted $C_k(S)$; the additional comparison needed at the end is not counted. As we discussed above, $C_k(S)$ is also the cost of an unsuccessful search for any key belonging to the interval $(\text{key}(x_k), \text{key}(x_{k+1}))$. The path length $C_k(S)$ is well defined for $k = 0, 1, \ldots, n$; recall that we consider the header as the 0-th item and **NIL** as the $(n+1)$-th item.

Clearly, the path length $C_k(S)$ can be decomposed into two contributions: the horizontal search cost, $F_k(S)$, plus the height $\text{H}(S)$ of the skip list. The horizontal search cost $F_k(S)$ is the number of forward pointers that should be traversed to reach x_k.

```
x := header(S); l := height(S);
while l > 0 do
    b := key(forward(x, l))
    if
        a ≤ b  ⟶  l := l − 1
      ▯ a > b  ⟶  x := forward(x, l)
    fi
end
if
    a = key(forward(x, 1))  ⟶  success
  ▯ a ≠ key(forward(x, 1))  ⟶  failure
fi
```

FIGURE 2. Search algorithm

Since $C_k(S)$ is both the cost of a successful search for x_{k+1}, $0 \leq k < n$, and the cost of an unsuccessful search for the interval (x_k, x_{k+1}), $0 \leq k \leq n$, it makes little sense to consider probabilities of access to items (p's) and probabilities of access to intervals (q's), unless we use a search procedure different from the one we have just described (see Section 7). Henceforth, we will assume that we are just given the probabilities of access

$$p_k = \Pr\{\text{access to the interval } (x_k, x_{k+1}]\}, \qquad 0 \leq k \leq n.$$

To end up this section, we define the *average path length* and *average horizontal path length* of a skip list S with respect to an access probability distribution $p = (p_0, p_1, \ldots, p_n)$.

Definition 2.1. *The* average path length *of S,* a.k.a. weigthed path length, *with respect to the access probability distribution $p = (p_0, \ldots, p_n)$ is*

$$\mathcal{C}(S, p) = \sum_{0 \leq k \leq n} p_k \, C_k(S).$$

Definition 2.2. *The* average horizontal path length *of S with respect to the access probability distribution $p = (p_0, \ldots, p_n)$ is*

$$\mathcal{F}(S, p) = \sum_{0 \leq k \leq n} p_k \, F_k(S).$$

By definition,

$$\mathcal{C}(S, p) = H(S) + \mathcal{F}(S, p).$$

3. A Lower Bound

The following lemma shows a non trivial lower bound on the average path length of any skip list, $\mathcal{C}(S,p)$, although it is not difficult to prove nor surprising.

Lemma 3.1. *For any skip list S and any probability distribution p,*

$$\mathcal{C}(S,p) \geq \mathcal{H}(p),$$

where $\mathcal{H}(p)$ denotes the entropy of the access probability distribution p, i.e.

$$\mathcal{H}(p) = -\sum_{0\leq k\leq n} p_k \log_2 p_k.$$

Proof. The lemma is easily proved by induction on n. For $n=0$, $\mathcal{C}(S,p)=\mathcal{H}(p)=0$, where S is the empty skip list. Assume now that Lemma 3.1 holds for skip lists of size $m<n$ and let S be an arbitrary skip list of size n. Then,

$$\begin{aligned}\mathcal{C}(S,p) &= \sum_{0\leq k\leq n} p_k\, C_k(S) = \sum_{0\leq k\leq u-1} p_k\ (C_k(\sigma) + \mathrm{H}(S) - \mathrm{H}(\sigma)) \\ &\quad + \sum_{u\leq k\leq n} p_k\ (C_{k-u}(\tau) + 1 + \mathrm{H}(S) - \mathrm{H}(\tau)),\end{aligned}$$

where x_u is the first item whose level is $\mathrm{H}(S)$, σ is the subskiplist that precedes x_u and τ the subskiplist that follows it (see Figure 3).

We can express the formula above in terms of average path lengths and take into account that $\mathrm{H}(S)-\mathrm{H}(\sigma)\geq 1$ and $\mathrm{H}(S)-\mathrm{H}(\tau)\geq 0$:

$$\begin{aligned}\mathcal{C}(S,p) &= w_{0,u-1}\mathcal{C}(\sigma,p') + w_{0,u-1}\,(\mathrm{H}(S)-\mathrm{H}(\sigma)) + w_{un}\mathcal{C}(\tau,p'') \\ &\quad + w_{un} + w_{un}\,(\mathrm{H}(S)-\mathrm{H}(\tau)) \geq w_{0,u-1}\mathcal{C}(\sigma,p') + w_{0,u-1} + w_{un}\mathcal{C}(\tau,p'') + w_{un} \\ &= w_{0,u-1}\mathcal{C}(\sigma,p') + w_{un}\mathcal{C}(\tau,p'') + 1,\end{aligned}$$

where $w_{ij} = p_i + \cdots + p_j$, $p' = (p_0/w_{0,u-1}, \ldots, p_{u-1}/w_{0,u-1})$ and $p'' = (p_u/w_{un}, \ldots, p_n/w_{un})$. We assume that for $i\leq k\leq j$, $p_k/w_{ij}=0$ if $w_{ij}=0$. Applying the inductive hypothesis, since the size of both σ and τ is smaller than n:

$$\begin{aligned}\mathcal{C}(S,p) &\geq w_{0,u-1}\mathcal{C}(\sigma,p') + w_{un}\mathcal{C}(\tau,p'') + 1 \geq w_{0,u-1}\mathcal{H}(p') + w_{un}\mathcal{H}(p'') + 1 \\ &\geq w_{0,u-1}\mathcal{H}(p') + w_{un}\mathcal{H}(p'') + \mathcal{H}(w_{0,u-1}, w_{un}) = \mathcal{H}(p).\end{aligned}$$

The last two steps of this derivation are based upon elementary properties of the entropy function. □

4. Optimal Skip Lists

This section considers the problem of constructing an optimal skip list, if we are given beforehand all the items of the skip list and an access probability distribution $p = (p_0, \ldots, p_n)$. The algorithm and its analysis are still valid with slight modifications if we want to work with access weights instead of probabilities. From a practical point of view, it is advantageous to work with integer weights and avoid floating point arithmetic, but for the theoretical development, we shall stick to access probabilities.

A skip list is *optimal* with respect to the access distribution p, if its average path length is minimum. Since both the height and the horizontal path length of the items depend uniquely on the levels of the items, the problem of constructing an optimal skip list reduces to the computation of the level that should be given to each item, given p. Once we know the level of each item, the actual optimal skip list can be easily constructed in time $\mathcal{O}(n \log n)$, since its height never exceeds $\mathcal{O}(\log n)$.

A solution to this problem uses dynamic programming as in the case of the construction of optimal search trees [7, 8]. The key observation here is that the principle of optimality that works for search trees fails for the case of skip lists, unless we restrict the notion of optimality to horizontal costs of skip lists of fixed height and then look for the optimal skip list among the optimal skip lists of each fixed height. Therefore, we define *h-optimal* skip lists: a skip list is h-optimal with respect to an access distribution p if its height is less or equal to h and its average horizontal path length w.r.t. p is minimum over the set of skip lists of height $\leq h$. It is easy to see that for any h, there must be at least an h-optimal skip list of height exactly h.

Let S_{ij} be an h-optimal skip list for the items $x_{i+1}, \ldots, x_j$, w.r.t. a properly normalized access distribution $\overline{p} = (\overline{p}_i, \ldots, \overline{p}_j)$. Hence,

$$f_{ij}^h = \mathcal{F}(S_{ij}, \overline{p}) = \sum_{i \leq k \leq j} \overline{p}_k F_k(S_{ij})$$

is minimum. The normalized access probability distribution is given by $\overline{p}_k = p_k / w_{ij}$, where $w_{ij} = \sum_{i \leq k \leq j} p_k$. Moreover, we can assume w.l.o.g. that S_{ij} has height h, and therefore its average path length is $c_{ij}^h = h + f_{ij}^h$. Note that the definition above makes sense for $0 \leq i \leq j \leq n$ and $h > 0$.

Let $\delta_{ij}^h = w_{ij} f_{ij}^h$. If we are able to compute δ_{0n}^h —and consequently, able to compute $c_{0n}^h = h + \delta_{0n}^h$— for $h = 1, 2, \ldots$, then the average path length of an optimal skip list with respect to p is c_{0n}^ℓ, where ℓ is such that $c_{0n}^\ell \leq c_{0n}^k$ for any k.

Let us consider first the trivial cases in the computation of δ_{ij}^h. If $i = j$ then the h-optimal skip list is always an empty skip list, so $\delta_{ii}^h = 0$. Similarly, if $i + 1 = j$, any h-optimal skip list contains one item, namely x_{i+1}, and $\delta_{i,i+1}^h = p_{i+1}$. When $h = 1$, all h-optimal skip lists are just simple linear linked lists and it follows that

$$\delta_{ij}^1 = \sum_{i \leq k \leq j} p_k (k - i).$$

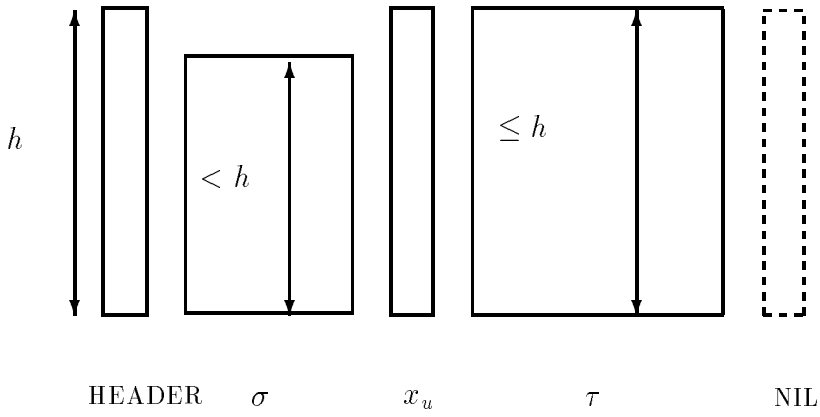

FIGURE 3. Recursive decomposition of a skip list.

Assume now that $h > 1$ and $i + 1 < j$. Take S_{ij} as before, an h-optimal skip list and decompose it as follows: let x_u be the least item in S_{ij} such that its level is h, with $i < u \leq j$. Let σ be the skip list containing the items $x_{i+1}, \ldots, x_{u-1}$ and τ the skip list for $x_{u+1}, \ldots, x_j$ (see Figure 3). Since S_{ij} is h-optimal, it follows that σ must be an $(h-1)$-optimal skip list and τ an h-optimal skip list with respect to the corresponding normalized access distributions.

The relation between horizontal path lengths in S_{ij} and the skip lists σ and τ is

$$F_k(S_{ij}) = \begin{cases} F_k(\sigma) & \text{if } i \leq k \leq u-1, \\ F_k(\tau) + 1 & \text{if } u \leq k \leq j. \end{cases}$$

From the equation above

$$\begin{aligned} \delta_{ij}^h &= \sum_{i \leq k \leq j} p_k F_k(S_{ij}) \\ &= \sum_{i \leq k \leq u-1} p_k F_k(\sigma) + \sum_{u \leq k \leq j} p_k (F_k(\tau) + 1) \\ &= \delta_{i,u-1}^{h-1} + \delta_{uj}^h + w_{uj}. \end{aligned}$$

The hypothesis of h-optimality of S_{ij} requires that

$$\delta_{ij}^h = \min\left(i < k \leq j : \delta_{i,k-1}^{h-1} + \delta_{kj}^h + w_{kj}\right). \tag{1}$$

We define b_{ij}^h to be the index of the pivot item in the h-optimal skip list S_{ij}, i.e. $b_{ij}^h = u$. If there were several indexes u, $i < u \leq j$, such that

$$\delta_{i,u-1}^{h-1} + \delta_{uj}^h + w_{uj}$$

is minimum, we take the largest one.

The set of matrixes $\{b^h\}_{h>0}$ implicitly give us the level that should be assigned to each item: if ℓ is the height of the optimal skip list and $u = b^\ell_{0n}$ then x_u is the first item with level ℓ; if $u > 1$ then $b^{\ell-1}_{0,u-1}$ is the index of the first item with level $\ell - 1$; if $u < n$ then b^ℓ_{un} is the index of the second item of level ℓ, etc.

The dynamic programming solution based upon Equation 1 needs space $\mathcal{O}(n^2)$ and time $\mathcal{O}(n^3)$ to compute the optimal skip list for a given fixed height. In the next section, we investigate nearly optimal skip lists, providing thus upper bounds for the average path length and the height of optimal skip lists. We show there that the height of an optimal skip list never exceeds $\log_2(n+1) + \mathcal{H}(p) + 3 = \mathcal{O}(\log n)$. Therefore, the computation of the optimal skip list would require space $\mathcal{O}(n^2 \log n)$ and time $\mathcal{O}(n^3 \log n)$.

Nevertheless, we can improve the performance of the algorithm if we observe that each matrix b^h is monotone for each row and column:

$$b^h_{i,j-1} \leq b^h_{ij} \leq b^h_{i+1,j}.$$

The proof of the monotonicity of b^h can be found in Appendix A and is similar to the one used in the case of optimal binary search trees [8, 11, 16]. In the case of binary search trees, b_{ij} is defined as the index of the root of the optimal tree for x_{i+1} up to x_j; the matrix $b = (b_{ij})$ implicitly gives the shape of the optimal tree and it is monotone in each row and column.

Because of the monotonicity of b^h, the recurrence in Equation 1 reads now

$$\delta^h_{ij} = \min\left(b^h_{i,j-1} \leq u \leq b^h_{i+1,j} : \delta^{h-1}_{i,u-1} + \delta^h_{uj} + w_{uj}\right).$$

It is not difficult to prove that this change improves the time complexity of the computation of δ^h_{0n} by a factor of n. The computation of the value δ^h_{ij} needs $b^h_{i+1,j} - b^h_{i,j-1} + 1$ elementary operations. Therefore, the time complexity of the computation of δ^h_{0n} is now

$$\begin{aligned}\mathcal{O}\left(\sum_{0\leq i<n-1}\sum_{i+1<j\leq n}(b^h_{i+1,j} - b^h_{i,j-1} + 1)\right) &= \mathcal{O}\left(\sum_{1\leq i\leq n-1} i + \sum_{1\leq i\leq n-1}(b^h_{i,n} - b^h_{0,i})\right)\\ &= \mathcal{O}\left(\frac{(n-1)n}{2}\right) + \sum_{1\leq i\leq n-1}\mathcal{O}(n) = \mathcal{O}(n^2).\end{aligned}$$

Hence, the time complexity of the algorithm to compute the optimal skip list also improves by a factor of n and is now $\mathcal{O}(n^2 \log n)$. The space needed by the new solution is still $\mathcal{O}(n^2 \log n)$.

A full implementation of the algorithm, including all the details, is given in Appendix B. The reader should notice that the algorithm given in Appendix B stops as soon as it finds a value ℓ such that $c^\ell_{0n} \leq c^{\ell+1}_{0n}$; we conjecture that if $c^\ell_{0n} \leq c^{\ell+1}_{0n}$

then c_{0n}^{ℓ} is actually the cost of an optimal skip list although we have not been able to prove it.

5. Nearly Optimal Skip Lists

There are two important reasons for considering *nearly optimal* skip lists, that is, skip lists whose average path length is not guaranteed to be minimum, but which is very close to it.

The first reason is a practical one: nearly optimal skip lists can be constructed in time $\mathcal{O}(n)$ and space $\mathcal{O}(n)$, while their performance is guaranteed to be logarithmic in the number of items and improves as the access probability distribution gets more and more skewed.

The second reason is that the average path length of a nearly optimal skip list can be easily analyzed, thus providing upper bounds to the performance of optimal skip lists. On the contrary, tight bounds for the average path length of optimal skip lists cannot be easily derived in a direct manner.

The construction of nearly optimal skip lists that we present here is based in the method of strict bisection: in each step, we choose the pivot item by considering a *reference point* γ and a *resolution* ℓ. The pivot item is such that the cumulated weight of the probabilities prior to it is below $\gamma + 2^{-\ell}$. The result is that whenever we follow a forward pointer or decrement the current level during a search, we discard a subskiplist whose cumulated weight is roughly one half of the weight of the skip list we were at that moment.

Let $X = \{x_1, \ldots, x_n\}$ be the set of items to be stored by a nearly optimal skip list S and $p = (p_0, \ldots, p_n)$ the given access probability distribution, as usual. Assume we have to construct a nearly optimal skip list S_{ij} for the items $x_{i+1}, \ldots, x_j$ with $i < j$ and let

$$w_i \equiv w_{0i} = p_0 + p_1 + \cdots + p_i, \quad 0 \leq i \leq n.$$

Hence, $w_i \leq w_j$ and there exists $\ell \geq 1$ and $0 \leq \gamma \leq 1$ such that

$$\gamma \leq w_i \leq w_j \leq \gamma + 2^{-\ell+1}.$$

Note that this assumption is trivially true for $i = 0$, $j = n$, $\gamma = 0$ and $\ell = 1$.

If $i + 1 = j$ the skip list S_{ij} contains a single item, x_{i+1}, so it can be trivially constructed. On the other hand, if $i + 1 < j$, we look for the greatest index k such that $w_{k-1} \leq \gamma + 2^{-\ell} \leq w_k$. If $i + 1 < k < j$ and since the middle point of the interval $(\gamma, \gamma + 2^{-\ell+1})$ is $\gamma + 2^{-\ell}$, we recursively construct the nearly optimal skip lists $S_{i,k-1}$ and S_{kj}, taking x_k as the pivot item of S_{ij}, that is, as the **NIL** item with respect to $S_{i,k-1}$ and as a header for S_{kj}. To construct $S_{i,k-1}$, we use the same reference point γ and the resolution $\ell := \ell + 1$, whereas we take the new reference point $\gamma := \gamma + 2^{-\ell}$ and the resolution $\ell := \ell + 1$ to construct S_{kj}. In both recursive calls the assumption on their respective γ and ℓ holds, if it were true when we were constructing S_{ij}.

```
procedure construct_near_optimal (in i, j, h : integer; in ℓ : integer; in γ : real)
    {w and level are global arrays initialized somewhere else }
    var k : integer end
    if
        h = 1  ⟶  for k := i + 1 to j do
                       level[k] := 1
                   end
      [] h > 1  ⟶  k := search_pivot(i, j, γ + 2^{-ℓ})

                   {i + 1 ≤ k ≤ j, w_{k-1} ≤ γ + 2^{-ℓ} < w_k, or
                    k = i + 1 if w_i > γ + 2^{-ℓ} or k = j if w_j ≤ γ + 2^{-ℓ} }

                   level[k] := h
                   if  i + 1 < k  ⟶  construct_near_optimal(i, k − 1, h − 1, ℓ + 1, γ)
                   fi
                   if  k < j  ⟶  construct_near_optimal(k, j, h − 1, ℓ + 1, γ + 2^{-ℓ})
                   fi
    fi
end
```

FIGURE 4. Main recursive procedure to construct nearly optimal skip lists.

The skip list $S_{i,k-1}$ is accessed by going one level down from the top node/level of x_i, whilst S_{kj} is accessed by following the forward pointer that goes from the top node/level of x_i to the top node/level of x_k.

If $k = i + 1$ or $k = i$ we take $k := i + 1$, and if $k = j$ or $k = j + 1$ we take $k := j$. In either situation one of the subskiplists is empty and hence, trivially constructed. The recursive algorithm is given in Figure 4.

If h is the height of S_{ij} then we set the level of x_k to be h; and we also require that both $S_{i,k-1}$ and S_{kj} have height $h - 1$ (see the discussion below and Section 6). If $h = 1$, the recursion ends, and the level of all items from x_{i+1} to x_j is set to 1.

It is absolutely necessary to fix the height of the nearly optimal skip list *a priori*; otherwise, the recursive construction would only stop if no new pivot could be found —that is, the trivial case when $i + 1 = j$—. Then, it could happen that the skip list had height $\mathcal{O}(\min\{n, \log_2(1/p_{\min})\})$, where $p_{\min}$ is the smallest access probability. If this happened, then the average path length could be worse than $\mathcal{O}(\log n)$.

Let us now consider the average path length of such nearly optimal skip lists. Let h be the height of the skip list S_{near} built by the algorithm. The algorithm induces a partition of the real interval $[0, 1]$ into 2^{h-1} subintervals, each of length 2^{-h+1}. Therefore, we can define I_j to be the set of indexes of the cumulated weigths that fall in the j-th interval, $0 \le j \le 2^{h-1} - 1$, that is, $I_j = \{i \mid j2^{-h+1} < w_i \le (j + 1)2^{-h+1}\}$. If $w_i = 0$ then we use the convention that $i \in I_0$.

Figure 5 shows a nearly optimal skip list with $h = 3$ and $n = 11$, the sets I_j and

the partition of the interval $[0,1]$ induced by the algorithm; only some of the p_i's and w_i's have been labelled.

It is not difficult to see that the skip list S_{near} that satisfies the following property: the number of forward steps $f_k = F_k(S_{\text{near}})$ needed to access a key in the range $(\text{key}(x_k), \text{key}(x_{k+1})]$ is 1 if k was selected as the pivot at the first recursive stage, at most 2 if k was selected as a pivot in the second recursive stage, etc. Let i and j be the first and last indexes in a call to construct_near_optimal during the ℓ-th stage —since the value of the resolution is incremented in all recursive calls, the levels of recursion or stages can be unambiguosly labelled by the value of the parameter ℓ—. During this call, we select the k-th item as the pivot for the nearly optimal skip list containing the items $\{x_{i+1}, \ldots, x_j\}$. By construction,

$$2^{-\ell+1} \geq w_j - w_i = p_{i+1} + p_{i+2} + \cdots + p_j \geq p_k.$$

Since $f_k \leq \ell$ we have

$$\begin{aligned} p_k &\leq 2^{-f_k+1}, \\ f_k &\leq 1 - \log_2 p_k. \end{aligned}$$

Let x_k be an item that is never selected as a pivot by the algorithm. Therefore, k cannot be the first element of its index set I_j and then the range $(\text{key}(x_k), \text{key}(x_{k+1})]$ would be accessed, in the worst case, as in a simple linked list starting at the first item of I_j.

Define a_k to be the rank of k minus one within its index set I_j, that is, $a_k = r - 1$ if k is the r-th element in I_j. The number of horizontal steps to reach the pivot preceding x_k is at most $h - 1$. There could be more than one pivot in the index set I_j; but since we are seeking an upper bound, we assume that the access from the first element in the index set of k —this element must be a pivot— to the element x_k is done in a purely sequential manner. Then it follows that $f_k \leq h - 1 + a_k$. On the other hand, $p_k \leq 2^{-h+1}$ because it is assumed that k is not the first element within its interval. Therefore,

$$\begin{aligned} p_k &\leq 2^{-h+1} \leq 2^{-f_k+a_k}, \\ f_k &\leq a_k - \log_2 p_k. \end{aligned}$$

To find an upper bound for the average horizontal path length of S_{near}, we consider separately the items that are selected as pivots and those that are not (notice that

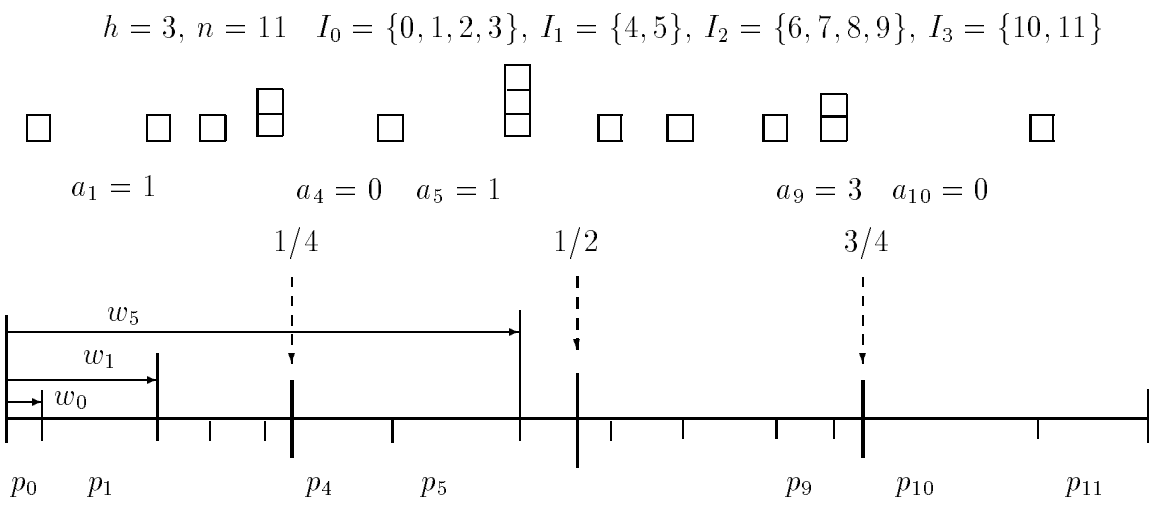

FIGURE 5. Partition of the probability distribution.

$a_k \neq 0$ for all these items). Hence,

$$\begin{aligned}
\mathcal{F}(S_{\text{near}}, p) &= \sum_{0 \le k \le n} p_k f_k = \sum_{\substack{0 \le k \le n \\ k \text{ is a pivot}}} p_k f_k + \sum_{\substack{0 \le k \le n \\ k \text{ is not a pivot}}} p_k f_k \\
&\le \sum_{\substack{0 \le k \le n \\ k \text{ is a pivot}}} p_k (1 - \log_2 p_k) + \sum_{\substack{0 \le k \le n \\ k \text{ is not a pivot}}} p_k (a_k - \log_2 p_k) \\
&= \sum_{\substack{0 \le k \le n \\ k \text{ is a pivot}}} p_k - \sum_{0 \le k \le n} p_k \log_2 p_k + \sum_{\substack{0 \le k \le n \\ k \text{ is not a pivot}}} p_k a_k \\
&\le 1 + \mathcal{H}(p) + \sum_{\substack{0 \le k \le n \\ a_k \neq 0}} p_k a_k .
\end{aligned}$$

Since a_k is always less than the rank of k within I_j then it is also true that $a_k \le \text{Card}(I_j)$. Moreover, by definition the sum of the probabilities for all $k \in I_j$, except the first one, must be less or equal to 2^{-h+1}. Hence,

$$\begin{aligned}
\mathcal{F}(S_{\text{near}}, p) &\le 1 + \mathcal{H}(p) + \sum_{0 \le j \le 2^{h-1}-1} \Bigg(\sum_{\substack{k \in I_j \\ a_k \neq 0}} p_k \text{Card}(I_j) \Bigg) \\
&= 1 + \mathcal{H}(p) + \sum_{0 \le j \le 2^{h-1}-1} \Bigg(\text{Card}(I_j) \sum_{\substack{k \in I_j \\ a_k \neq 0}} p_k \Bigg) \\
&\le 1 + \mathcal{H}(p) + \frac{1}{2^{h-1}} \sum_{0 \le j \le 2^{h-1}-1} \text{Card}(I_j) = 1 + \mathcal{H}(p) + \frac{n+1}{2^{h-1}}.
\end{aligned}$$

We can conclude that

$$\mathcal{C}(S_{\text{near}}, p) = H(S_{\text{near}}) + \mathcal{F}(S_{\text{near}}, p) \leq h + 1 + \mathcal{H}(p) + \frac{n+1}{2^{h-1}}.$$

Finding the minimum of the function $c(h) = h + 1 + \mathcal{H}(p) + (n+1)/2^{h-1}$ gives $h_{\min} = \log_2(n+1) + \log_2(\ln 2) + 1 \approx \log_2(n+1) + 0.47$. Using values of h between $\log_2(n+1)$ and $\log_2(n+1)+1$ to construct the nearly optimal skip list we get $\mathcal{C}(S_{\text{near}}, p) \leq \log_2(n+1) + \mathcal{H}(p) + 3$. This is also an upper bound to the average path length of the optimal skip list, since it must be less or equal to that of the nearly optimal skip list. Moreover, the height of the optimal skip list is less than its average path length and hence we conclude that it is $\mathcal{O}(\log n)$ as claimed in the previous section.

Notice that building a skip list using simple dichotomy, i.e. the $n/2$-th item is given level $\log_2 n$, the $n/4$-th and $3n/4$-th items are given level $\log_2 n - 1$ and so forth, guarantees logarithmic average path length. But for these balanced skip lists an upper bound on the average path length is $2\log_2 n + \mathcal{O}(1)$; if the access probability is skewed then the upper bound $\log_2 n + \mathcal{H}(p) + \mathcal{O}(1)$ for nearly optimal skip lists proves that taking into account the access probabilities is worthy, and nearly optimal skip lists are as easy to build as the balanced ones.

To prove that the time complexity of the algorithm that builds the nearly optimal skip list is $\mathcal{O}(n)$, we give an upper bound of the number of steps needed to do the search of the pivots at every level of recursion, now labelled by the parameter h. At the first level, h, the algorithm searches for the pivot item x_k between n possibilities, so it takes $\lceil \log_2 n \rceil$ steps, at most, to find it using binary search. At a second level, $h-1$, we have to find two pivots: one between 1 and $k-1$, and another between $k+1$ and n. Hence, in the worst case, we need $2\lceil \log_2(n/2) \rceil$ steps. In general, at a level $h-i$ the maximum number of steps needed is $2^i \lceil \log_2(n/2^i) \rceil$.

Therefore, an upper bound, $U(n,h)$, of the number of steps needed by the algorithm is

$$U(n,h) = \sum_{0 \leq i \leq h-2} 2^i (\log_2 \left(n/2^i\right) + 1) = (1 + \log_2 n)(2^{h-1} - 1) - ((h-3)2^{h-1} + 2).$$

Using $h = \log_2(n+1)$, yields $U(n,h) = 2n - \log_2 n + \mathcal{O}(1) = \mathcal{O}(n)$. Besides the cost of searching the pivots, the rest of the algorithm contributes $\mathcal{O}(n)$ steps to the overall time complexity.

Once the level of each item is computed by the algorithm, the actual skip list can be constructed in time $\mathcal{O}(n)$ since we need to allocate n nodes plus the header and there are $\mathcal{O}(n)$ pointers to be linked. This claim trivially follows from the fact that any nearly optimal skip list has one element of level h, at most two of level $h-1$, etc. and $h = \Theta(\log n)$.

6. Related Issues

In this section we do a quick overview of several interesting issues, related to the construction of optimal and nearly optimal skip lists. We start considering a number of different strategies to construct nearly optimal skip lists.

First, the partitioning criterium can be changed and based upon the quantities $s_i = w_{i-1} + p_i/2$ instead of w_i's. The idea is that the algorithm would select the elements closer to the reference points as pivots. There are instances where this partitioning criterium performs better than the one using w_i's; but there are other instances, where the contrary is true. The upper bound for the average path length of the nearly optimal skip list built using the s_i's is $h + \mathcal{H}(p) + (n+1)/2^{h-2} + 2$, which is not very different from the one we obtain when using $w_i's$ as the partitioning criterium.

As we have already said in Section 5 the algorithm uses strict bisection: it cuts the interval $[0,1]$ into 2^{h-1} intervals of equal length. However, it is also possible to construct a nearly optimal skip list using *relative* bisection. In a given stage, if we are building a skip list S_{ij} for the items $x_{i+1}, \ldots, x_j$, the selected pivot is the element that leaves at its right half or more of the cumulated probability of access to S_{ij}. Then the procedure is recursively applied to construct the skip lists to the left and to the right of the pivot. It is not difficult to show that an upper bound of the type $\mathcal{O}(\mathcal{H}(p) + \log n)$ for the average path length also holds in this case.

Last but not least, the algorithm that we have presented selects a pivot of level h and constructs nearly optimal skip lists to the left and to the right of the pivot, each one of height $h-1$. We say in this case that the nearly optimal skip list is built using *symmetric recursion*. However, the nearly optimal skip list to the right of the pivot could have height h. In this case, we say that the skip list is built by *asymmetric recursion*. If we follow this procedure then the average path length of the skip list is always at least as good as if the skip list were built using symmetric recursion. But it is important to mention that for some probability distributions, the skip lists built by symmetric and asymmetric recursion have exactly the same average path length.

The tradeoff for constructing a possibly better skip list using asymmetric recursion is that the time complexity of the algorithm and the space needed by the resulting skip list rise from $\mathcal{O}(n)$ to $\mathcal{O}(n \log n)$.

Other meaningful questions arise if we consider the construction of optimal skip lists for alternative definitions of the search cost.

To begin with, we should notice that the procedures described in this paper can be used without changes if we want to construct efficient skip lists for successful searching: just let p_i be the probability of access to the $(i+1)$-th item, where $i = 0, \ldots, n-1$ and $p_n = 0$.

However, if successful queries to the static skip lists are frequent it may be worth using a variant of the search algorithm that stops as soon as the key we are searching

for is seen. If the key that we search for is present, then we will need less vertical steps and eventually less horizontal steps. In this variant of the search, we have to perform two comparisons in each iteration and decide whether decrementing the current level, following the forward pointer or reporting success upon the outcome of these comparisons.

Easy generalizations of our algorithms for optimal and nearly optimal skip lists can be defined when the cost described above is to be minimized. The average path length is defined in terms of the path length of each of the items and their probabilities of access, instead of the path length of intervals and their probabilities.

It can be shown using similar techniques that

$$\mathcal{H}(p) \leq \mathcal{C}'(S_{\text{opt}}, p) \leq \mathcal{C}'(S_{\text{near}}, p) \leq 2\mathcal{H}(p) + O(1),$$

where $\mathcal{C}'(\cdot)$ denotes the modified average cost, S_{opt} is an optimal skip list with respect to $\mathcal{C}'$ and p, and S_{near} is a nearly optimal skip list built using the algorithm given in Section 5 with $h = \log_2 n + O(1)$.

Another possibility, now introducing two types of access probabilities, is to consider skip lists minimizing

$$\mathcal{C}(S, p) = \sum_{1 \leq k \leq n} p_k C'_k(S) + \sum_{0 \leq k \leq n} q_k C_k(S),$$

where p_k is the probability of accessing the k-th item, $C'_k(S)$ is the cost of a successful search for that item, q_k is the probability of accessing the k-th interval and $C_k(S)$ is the cost of an unsuccessful search for that interval. Now, $C_k(S) \neq C'_{k+1}(S)$, so it is convenient to dealt with both p's and q's. By contrast, in the paper, we have discussed the construction of optimal skip lists under the assumption that the cost of accessing the $(k+1)$-th item and the k-th interval was the same, and p_k was taken as the probability of accessing a key in the *closed* interval $(\text{key}(x_k), \text{key}(x_{k+1})]$.

Under some circumstances, the cost of searches is mainly due to key-to-key comparisons. The search algorithm can be then improved by avoiding redundant comparisons between the search key and the key of some item that has been inspected already during the search. These redundant comparisons may be substituted by "cheaper" pointer-to-pointer comparisons [14]. If we use this improvement and successful searches stop as soon as the item is located (i.e. the search does not continue in lower level lists of the skip list) and we want to minimize the average cost measured in number of key comparisons, then the well-known algorithms for optimal and nearly optimal binary search trees can be used. The reason is that for this model of costs, there is bijection between skip lists and binary search trees (down=left, forward=right) such that the costs of accessing each item and interval are the same in both data structures.

7. Conclusions

Further research in this topic includes improving the upper bounds on the performance of both optimal and nearly optimal skip lists, other approaches to the construction of nearly optimal skip lists (as those already discussed in the previous section), more efficient algorithms for special cases of the weights, and proving the following two conjectures:

i) If $c_{0n}^{\ell} \leq c_{0n}^{\ell+1}$, i.e. the average path length of ℓ-optimal skip lists is smaller or equal than the average path length of $(\ell+1)$-optimal skip lists, then $c_{0n}^{\ell} \leq c_{0n}^{\ell+1} \leq c_{0n}^{\ell+2} \leq c_{0n}^{\ell+3} \leq \ldots$. In other words, if ℓ is the minimum value such that $c_{0n}^{\ell} < c_{0n}^{\ell+1}$, then there exists an optimal skip list such that its height is ℓ.

ii) Furthermore, for any access probability distribution p and for any n, there exists an optimal skip list of height $\mathcal{O}(\mathcal{H}(p))$ (the entropy of p). This is consistent with our empirical observation that optimal skip lists tend to "shallow", i.e. their height tends to be small, particularly for very skewed distributions.

References

1. B. Allen, *On the costs of optimal and near-optimal binary search trees*, Acta Informatica **18** (1982), 255–263.
2. L.R. Gotlieb, *Optimal multi-way search trees*, SIAM J. on Computing **10** (1981), 422–433.
3. T.C. Hu and A.C. Tucker, *Optimal computer search trees and variable-length alphabetical*, SIAM J. Appl. Math. **21** (1971), 514–532.
4. S.-H.S. Huang and C.K. Wong, *Optimal binary split trees*, Journal of Algorithms **5** (1984), 69–79.
5. A. Itai, *Optimal alphabetic trees*, SIAM J. on Computing **5** (1976), 9–18.
6. P. Kirschenhofer and H. Prodinger, *The path length of random skip lists*, Acta Informatica **31** (1994), 775–792.
7. D.E. Knuth, *Optimum binary search trees*, Acta Informatica **1** (1971), 14–25.
8. ———, *The art of computer programming: Sorting and searching*, vol. 3, Addison-Wesley, 1973.
9. J. F. Korsh, *Growing nearly optimal binary search trees*, Inform. Proc. Lett. **14** (1982), 139–143.
10. K. Mehlhorn, *Nearly optimal binary search trees*, Acta Informatica **5** (1975), 287–295.
11. ———, *Data structures and algorithms: Sorting and searching*, vol. 1, Springer-Verlag, 1984.
12. J.I. Munro, T. Papadakis, and R. Sedgewick, *Deterministic skip lists*, ACM-SIAM Symposium on Discrete Algorithms 1992 (SODA), pp. 367–375.
13. T. Papadakis, J.I. Munro, and P.V. Poblete, *Average search and update costs in skip lists*, BIT **32** (1992), 316–332.
14. W. Pugh, *A skip list cookbook*, Tech. Report CS-TR-2286.1, UMIACS-TR-89-72.1, Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, MD, 1990.
15. ———, *Skip lists: A probabilistic alternative to balanced trees*, Comm. ACM **33** (1990), 668–676.
16. F.F. Yao, *Efficient dynamic programming using quadrangle inequalities*, ACM Symposium on Theory of Computing 1980 (STOC), pp. 426–435.

Appendix A. Proof of the Monotonicity of b^h

We prove in this Appendix that, for all $h \geq 1$, and for all $0 \leq i < j+1 \leq n+1$

$$b^h_{i,j-1} \leq b^h_{ij} \leq b^h_{i+1,j},$$

where b^h_{ij} is the largest u, $i < u \leq j$, such that

$$\delta^{h-1}_{i,u-1} + \delta^h_{uj} + w_{uj}$$

is minimum.

Our proof follows that given by Mehlhorn in [11] for the case of optimal binary search trees.

First of all, we have that w_{ij} satisfies

$$w_{ij} + w_{i'j'} = w_{i'j} + w_{ij'}, \qquad i \leq i' \leq j \leq j'. \tag{2}$$

Next we show that δ^h_{ij} satisfies the quadrangle inequality, i.e.

$$\delta^h_{ij} + \delta^h_{i'j'} \leq \delta^h_{i'j} + \delta^h_{ij'}, \quad i \leq i' \leq j \leq j', \quad h > 0. \tag{3}$$

We prove that δ_{ij} satisfies the inequality above by induction on $l = j' - i$ and h and repeatedly using the following property

$$\delta^h_{ij} \leq \delta^{h-1}_{i,k-1} + \delta^h_{kj} + w_{kj},$$

for all $i < k \leq j$, as well as the fact that $\delta^h_{ii} = 0$ and $\delta^{h-1}_{ij} \geq \delta^h_{ij}$. We also recall that, by definition, for $u = b^h_{ij}$

$$\delta^h_{ij} = \delta^{h-1}_{i,u-1} + \delta^h_{uj} + w_{uj}.$$

The inequality for $h = 1$ is in fact an equality: for any $i \leq i' \leq j \leq j'$

$$\delta^1_{ij} + \delta^1_{i'j'} = \delta^1_{i'j} + \delta^1_{ij'}.$$

If $l \leq 1$ the quadrangle inequality trivially holds no matter what h is, since either $i = i'$ or $j = j'$.

Assume now that $h > 1$, $i < i' = j < j'$ and $b^h_{ij'} = u \leq j$. The left hand side of the inequality is

$$\begin{aligned}
\delta^h_{ij} + \delta^h_{jj'} &\leq \delta^{h-1}_{i,u-1} + \delta^h_{uj} + w_{uj} + \delta^h_{jj'} \\
&\leq \delta^{h-1}_{i,u-1} + \delta^h_{uj} + w_{uj'} + \delta^h_{jj'} \\
&\leq^{h.i.} \delta^{h-1}_{i,u-1} + w_{uj'} + \delta^h_{uj'} + \delta^h_{jj} = \delta^h_{ij'} + \delta^h_{jj} = \delta^h_{ij'}.
\end{aligned}$$

If $i < i' = j < j'$, but $j < u \leq j'$, then we have

$$\begin{aligned}
\delta^h_{ij} + \delta^h_{jj'} &\leq \delta^h_{ij} + \delta^{h-1}_{j,u-1} + \delta^h_{uj'} + w_{uj'} \\
&\leq \delta^{h-1}_{ij} + \delta^{h-1}_{j,u-1} + \delta^h_{uj'} + w_{uj'} \\
&\leq^{h.i.} \delta^{h-1}_{i,u-1} + \delta^{h-1}_{jj} + \delta^h_{uj'} + w_{uj'} = \delta^h_{ij'},
\end{aligned}$$

and hence the quadrangle inequality also holds.

To prove that Equation 3 is satisfied when $i < i' < j < j'$ we should consider the case where $u = b^h_{i'j}$ is smaller than $u' = b^h_{ij'}$ and its symmetric case, where $u > u'$. We only develop the proof for the case $u \leq u'$. Note that in this case $i < i' < u \leq u' \leq j'$ and $u \leq j < j'$.

$$\begin{aligned}
\delta^h_{ij} + \delta^h_{i'j'} &\leq \delta^{h-1}_{i,u-1} + \delta^h_{uj} + w_{uj} + \delta^{h-1}_{i',u'-1} + \delta^h_{u'j'} + w_{u'j'} \\
&\leq^{h.i.} \delta^{h-1}_{i,u'-1} + \delta^h_{uj} + w_{uj} + \delta^{h-1}_{i',u-1} + \delta^h_{u'j'} + w_{u'j'} \\
&= \delta^{h-1}_{i,u'-1} + \delta^h_{u'j'} + w_{u'j'} + \delta^{h-1}_{i',u-1} + \delta^h_{uj} + w_{uj} \\
&= \delta^h_{ij'} + \delta_{i'j}.
\end{aligned}$$

The last step of our proof requires the introduction of the function $\delta_k(i,j)$

$$\delta_k(i,j) = \delta^{h-1}_{i,k-1} + \delta^h_{kj} + w_{kj}.$$

Let $i < k \leq k' \leq j$. Applying Equations 2 and 3 yields

$$\begin{aligned}
\delta_k(i,j) + \delta_{k'}(i,j+1) &= \delta^{h-1}_{i,k-1} + \delta^h_{kj} + w_{kj} + \delta^{h-1}_{i,k'-1} + \delta^h_{k',j+1} + w_{k',j+1} \\
&\leq \delta^{h-1}_{i,k-1} + \delta^h_{k,j+1} + w_{kj} + \delta^{h-1}_{i,k'-1} + \delta^h_{k'j} + w_{k',j+1} \\
&= \delta^{h-1}_{i,k-1} + \delta^h_{k,j+1} + w_{k,j+1} + \delta^{h-1}_{i,k'-1} + \delta^h_{k'j} + w_{k'j} \\
&= \delta_k(i,j+1) + \delta_{k'}(i,j).
\end{aligned}$$

Hence,

$$\delta_k(i,j) - \delta_{k'}(i,j) \leq \delta_k(i,j+1) - \delta_{k'}(i,j+1).$$

From the equation above it follows:

$$\delta_{k'}(i,j) \leq \delta_k(i,j) \implies \delta_{k'}(i,j+1) \leq \delta_k(i,j+1).$$

Recall that if $\delta_k(i,j)$ is minimum for several distinct values of k, we take the largest as the index of the pivot item; therefore, if the pivot item of S_{ij} were k', then no $k < k'$ could be the pivot item for $S_{i,j+1}$. By the definition of b^h_{ij} that means that

$$b^h_{ij} \leq b^h_{i,j+1}.$$

The monotonicity of b^h along columns ($b^h_{ij} \leq b^h_{i+1,j}$) can be proved in a similar way.

Appendix B. Optimal Skip Lists: The Algorithm

We describe here a full implementation of the algorithm discussed in Section 4. The code is written in C.

The procedure `compute_optimal` is given `p`, the access probability distribution, and `n`, the size of the skip list. It returns an array `level` of size `n`, with the levels that should be assigned to each item in an optimal skip list.

The main loop of the procedure computes in each iteration the matrix of "denormalized" horizontal costs `delta` for an optimal skip list of the current height `h`. To compute the matrix of horizontal costs `delta`, we need the matrix of horizontal costs previously computed `delta_prev` and the matrix of weights w_{ij} called `w`. While computing `delta`, a matrix of pivots `B[h]` is also used and updated.

If the cost `val` of the optimal skip list of fixed height `h` (accounting both the horizontal cost and the height) is better than the best one found up to the moment, `min`, then the matrix `delta_prev` is set to `delta`, and `min` is set to `val`. Afterwards, the next height is tried. To avoid the expensive copy of a matrix to another, both `delta` and `delta_prev` are actually pointers to the matrices `D1` and `D2`.

⟨ *⟩≡
⟨*definitions*⟩

```
void compute_optimal(float p[], int n, int level[])
{
   int i, j, u, umin, h;
   float **tmp, **delta, **delta_prev;
   float val, min, fmin;
   fmatrix w;
   fmatrix D1, D2;
   pivot_matrix B;

  delta = (float **) D1; delta_prev = (float **) D2;
  ⟨compute array w⟩
  ⟨initialize basic cases of D1 and D2⟩
  ⟨initialize the cost matrix delta_prev⟩
  ⟨initialize min to the cost of the 1-optimal skip list⟩
  ⟨initialize the pivots matrix B[1]⟩
  h = 2;
  for (;;)
   {
      ⟨compute an optimal skip list of level h and its cost matrix delta⟩
      if (val < min) {
        tmp = delta; delta = delta_prev; delta_prev = tmp;
        min = val;
      }
      else break;
      h++;
    }
  ⟨reconstruct the levels using B⟩
}
```

Before we describe in depth each part of the algorithm, let's introduce some definitions needed by the program.

⟨*definitions*⟩≡
```
#define MAX 1000          /* maximum number of items allowed */
#define LOGMAX 10         /* maximum height */

typedef float fmatrix[MAX][MAX];
typedef int pivot_matrix[LOGMAX][MAX][MAX];
```

The initialization of the matrix `w` is trivially easy, since $w_{ij} = w_{i,j-1} + p_j$ for $j > i$ and $w_{ii} = p_i$:

⟨*compute array w*⟩≡
```
  for(i = 0; i <= n; i++)
    w[i][i] = p[i];
  for(i = 0; i <= n; i++)
    for(j = i + 1; j <= n; j++)
      w[i][j] = w[i][j - 1] + p[j];
```

No matter what the height of the skip list is, we know the values of δ^h_{ii} and $\delta^h_{i,i+1}$ and initialize `D1` and `D2` accordingly:

⟨*initialize basic cases of D1 and D2*⟩≡
```
for(i = 0; i < n; i++) {
  D1[i][i] = D2[i][i] = 0;
  D1[i][i + 1] = D2[i][i + 1] = p[i + 1];
}
D1[n][n] = D2[n][n] = 0;
```

Moreover, for `h`=1, the matrix `delta_prev` must be initialized with the values δ^1_{ij} for $0 \le i + 2 \le j \le n$ as described in Section 4:

⟨*initialize the cost matrix delta_prev*⟩≡
```
for(i = 0; i < n - 1; i++)
  for(j = i + 2; j <= n; j++)
    delta_prev[i][j] = delta_prev[i][j - 1] + p[j] * (j - i);
```

The horizontal cost of the 1-optimal skip list is given by δ^1_{0n} and hence

⟨*initialize min to the cost of the 1-optimal skip list*⟩≡
```
min = delta_prev[0][n] + 1;
```

The kernel of the optimization procedure computes the h-optimal skip list for the current height h. For each i and j, $0 \leq i \leq j \leq n$ we compute the horizontal cost for an h-optimal skip list δ_{ij}^h from the previously computed costs for $(h-1)$-optimal skip lists and previous δ_{rs}^h. The recurrence in Equation 1 requires that all δ_{rs}^{h-1} have been computed —these are stored in `delta_prev`— and that the matrix `delta` for the current height is filled by rows, from bottom ($i = n$) to top ($i = 0$) and from left ($j = i + 1$) to right ($j = n$). In each iteration, the cost δ_{ij}^h is computed by a linear search of the pivot, with the lower and upper range of the search given by the matrix of pivots `B[h]`. Once the cost and the pivot are found, they are stored in `delta` and `B[h]` respectively.

⟨*compute an optimal skip list of level h and its cost matrix delta*⟩≡
```
⟨initialize the pivots matrix B[h]⟩
for(i = n; i >= 0; i--)
  for(j = i + 2; j <= n; j++)
   {
     dmin = (float) n * n;
     for(u = B[h][i][j - 1]; u <= B[h][i + 1][j]; u++)
       {
         val = delta_prev[i][u - 1] + delta[u][j] + w[u][j];
         if (val <= dmin) { dmin = val; umin = u; }
       }
     delta[i][j] = dmin;
     B[h][i][j] = umin;
   }
val = delta[0][n] + h;
```

The initialization of each matrix `B[h]` is not difficult. By definition, b_{ij}^h makes sense only if $i < j$, i.e. there exists a least one item to be the pivot. In particular, $b_{i,i+1}^h = i + 1$, since there is only one element that we consider. If $h = 1$ then the pivot must always be the first item so $b_{ij}^1 = i + 1$. We initialize `B[1]` separately in account of this observation and to guarantee that the optimal skip list —in fact, the levels of the items— is appropiately reconstructed.

⟨*initialize the pivots matrix B[h]*⟩≡
```
for (i = 0; i < n; i++) B[h][i][i+1] = i + 1;
```

⟨*initialize the pivots matrix B[1]*⟩≡
```
for (i = 0; i < n; i++)
 for (j = i + 1; j <= n; j++)
   B[1][i][j] = i + 1;
```

Finally, the reconstruction of the optimal skip list is just a call to a recursive procedure `compute_level(h,i,j)`: at level h, all pivots (i.e., items with level h) between x_i and x_j are marked as so using the matrix `B[h]`; if x_a is one of such pivots and x_b is the next one, with $b \neq a+1$, we recursively call to `compute_level(h - 1, a, b - 1)`.

⟨*reconstruct the levels using B*⟩≡
```
compute_level(h, 0, n, level, B);
```

⟨*definitions*⟩+≡
```
void compute_level(int h, int i, int j, int level[], pivot_matrix B)
{
 int a, b;

 for(a = i; a < j; a = B[h][a][j]) {
  b = B[h][a][j];
  level[b] = h;
  if (b > a + 1) compute_level(h - 1, a, b - 1, level, B);
 }
}
```

Appendix C. Nearly Optimal Skip Lists: The Algorithm

This Appendix describes the full implementation of the algorithm discussed in Section 5. The code is written in C.

The procedure `compute_optimal` is given `p`, the access probability distribution, and `n`, the size of the skip list. It returns its result in an array `level` of size `n`, such that `level`$[i]$ is the level that should be assigned to the i-th item in the nearly optimal skip list.

⟨ **⟩≡
```
⟨definitions⟩
void compute_near_optimal(float p[], int n)
{
  int j;
  ⟨compute array w⟩
  ⟨initial call to construct_near_optimal⟩
}
```

In the definitions, the global arrays `w` and `level` are declared:

⟨*definitions*⟩≡
```
#define MAX 1000
int level[MAX];
float w[MAX];
```

The initialization of the array `w` is straightforward, since $w_j = w_{j-1} + p_j$ for $j > 0$ and $w_0 = p_0$:

⟨*compute array w*⟩≡
```
w[0] = p[0];
for(j = 1; j <= n; j++)
  w[j] = w[j - 1] + p[j];
```

The procedure `construct_near_optimal(i,j,h,pl,gamma)` constructs a nearly optimal skip list for the items x_{i+1} to x_j such that its height is h; the reference point is given by `gamma` and the resolution is given by `pl`. In fact, if the resolution is ℓ then `pl`$=2^{-\ell}$. The first call to the recursive procedure must be therefore with `i`$=0$, `j`$=n$, `h`$=\Theta(\log n)$, and `gamma`$=0$. The initial value of `pl` must be 0.5, since the initial resolution is $\ell = 1$. We assume that `log2(n)` computes $\lfloor \log_2 n \rfloor$.

⟨*initial call to construct_near_optimal*⟩≡
```
construct_near_optimal(0, n, log2(n+1), 0.5, 0.0);
```

The construction of the nearly optimal skip list follows the description given in Section 5. If $h = 1$ then the recursion ends and all items from x_{i+1} to x_j are given level 1. Otherwise, a suitable pivot k is sought using `search_pivot`, the item x_k is given level h, and skip lists to the left and to the right are recursively built when appropiate. The parameters in the recursive calls to `construct_near_optimal` have already been discussed in Section 5 and are more or less self-evident.

⟨*definitions*⟩+≡
```
void construct_near_optimal(int i, int j, int h, float pl, float gamma)
{
 int k;

 if (h == 1)
    for(k= i+1; k <= j; k++) level[k] = 1;
 else {
     k = search_pivot(i, j, gamma + pl);
     level[k] = h;

     if (i < k - 1) construct_near_optimal(i, k - 1, h - 1, pl / 2, gamma);
     if (k < j)     construct_near_optimal(k, j, h - 1, pl / 2, gamma + pl);
 }
}
```

In order to find a pivot, that is, the greatest index k such that $i+1 < k < j$ and $w_{k-1} \leq \texttt{cut} \leq w_k$, where $\texttt{cut} = \gamma + 2^{-\ell}$, we use binary search over the array `w`. Before applying the binary search, we check the two extreme cases $w_i > \texttt{cut}$ and $w_j \leq \texttt{cut}$; the function returns $i+1$ and j respectively.

⟨*definitions*⟩+≡

```
int search_pivot(int i, int j, float cut)
{
 int a, b, k;

 if (w[i + 1] > cut) return i + 1;
 if (w[j] <= cut)    return j;

 a = i + 1; b = j;
 while (b > a + 1) {
     k = (a + b) / 2;
     if (w[k] > cut) b = k;
       else          a = k;
 }
 return b;
}
```

Departament de Llenguatges i Sistemes Informàtics.Universitat Politècnica de Catalunya.Pau Gargallo 5, 08028-Barcelona, Spain.

E-mail: {conrado,roura}@goliat.upc.es