Nordic Journal of Computing 1(1994), 389–401.

A NEW ALGORITHM FOR THE CONSTRUCTION OF OPTIMAL B-TREES

PETER BECKER
Wilhelm-Schickard-Institut für Informatik,
Universität Tübingen
Sand 13, 72076 Tübingen, Germany
becker@informatik.uni-tuebingen.de
Fax: +49 7071 / 295958

Abstract. In this paper the construction of optimal B-trees for n keys, n key weights, and $n+1$ gap weights, is investigated. The best algorithms up to now have running time $O(k\,n^3 \log n)$, where k is the order of the B-tree. These algorithms are based on dynamic programming and use step by step construction of larger trees from optimal smaller trees. We present a new algorithm, which has running time $O(k\,n^\alpha)$, with $\alpha = 2 + \log 2/\log(k+1)$. This is a substantial improvement to the former algorithms. The improvement is achieved by applying a different dynamic programming paradigm. Instead of step by step construction from smaller subtrees a decision model is used, where the keys are placed by a sequential decision process in such a way into the tree, that the costs become optimal and the B-tree constraints are valid.

CR Classification: E.1, H.2.2, I.2.8

Key words: B-Tree, optimization, dynamic programming

1. Introduction

In this article the problem of constructing an optimal B-tree for n keys, n key weights, and $n+1$ gap weights is considered. The best algorithms up to now have running time $O(k\,n^3 \log n)$, where k is the order of the B-tree, see [7, 3]. These algorithms are adaptations of the well known algorithm of Knuth [4] for the construction of optimal binary search trees with n key weights and $n+1$ gap weights. Furthermore, in [7, 2] algorithms for the construction of optimal multiway trees of order t are presented. These algorithms also adapt Knuth's dynamic programming scheme and have running time $O(t\,n^3)$. All these algorithms use step by step construction of larger trees from optimal smaller trees, that means an optimal tree for the set $\{k_i, \ldots, k_j\}$ is constructed from optimal trees for $\{k_i, \ldots, k_{b-1}\}$ and $\{k_{b+1}, \ldots, k_j\}$ for some b $(i \leq b \leq j, i \neq j)$.

In this article a new algorithm for the problem of constructing an optimal B-tree is presented. The improvement is achieved by applying a different dynamic programming paradigm. Instead of step by step construction from

Received May 1994. Accepted September 1994.

smaller subtrees a decision model is used, where the keys are placed by a sequential decision process in such a way into the tree, that the search costs become optimal and the B-tree constraints are valid. This new approach results in a running time of $O(k\,n^{\alpha})$ with $\alpha = 2 + \log 2/\log(k+1)$, which is a substantial improvement to the former algorithms, especially for k of realistic size. Common values of k in database applications are between 20 and 50. This implies values of 2.23 resp. 2.18 for the exponent α.

The rest of the paper is structured in the following way: in Section 2 a formal description of the problem is given. Section 3 reviews the best algorithms up to now. In Section 4 the new approach is presented: the decision model is explained, the attached dynamic program is formulated and the solution algorithm is stated. Section 5 gives the complexity results and Section 6 presents some ideas about further improvements.

2. The problem

First, we review the definition of a B-tree, cf. [1].

Definition 1. *A B-tree of order k is a multiway search tree that satisfies the following conditions:*

(1) Each node has at most $2k$ keys.

(2) Each node, except the root, has at least k keys.

(3) The root has at least one key.

(4) A nonleaf node with d keys has exactly $d+1$ children.

(5) All leaves are on the same level.

The following well known theorem (cf. [1]) is crucial for our complexity results in Section 5:

Theorem 1. *Let h_u be the maximum height of a B-tree of order k with n keys and let h_l be its minimum height. For the height h of a B-tree of order k with n keys the following equation is valid:*

$$h_l = \left\lceil \frac{\log(n+1)}{\log(2k+1)} \right\rceil \le h \le \left\lfloor 1 + \log(\frac{n+1}{2})/\log(k+1) \right\rfloor = h_u$$

Now we give the problem formulation. We have keys $k_1 < k_2 \ldots < k_n$, an order k and positive weights $q_0, p_1, q_1, p_2, \ldots, p_n, q_n$. p_i are the *key weights* and q_j are the *gap weights*. We define the sum of all weights as $w := \sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_j$. We may interpret the value $\alpha_i := p_i/w$ as the probability that key k_i is requested and the value $\beta_j := q_j/w$ as the probability, that a search is made for a key d with $k_j < d < k_{j+1}$. We assume that we have artificial keys $k_0 = -\infty$ and $k_{n+1} = \infty$. We define $pq(i,j) := (q_i, p_{i+1}, q_{i+1}, \ldots, p_j, q_j)$ as the sequence of gap and key weights

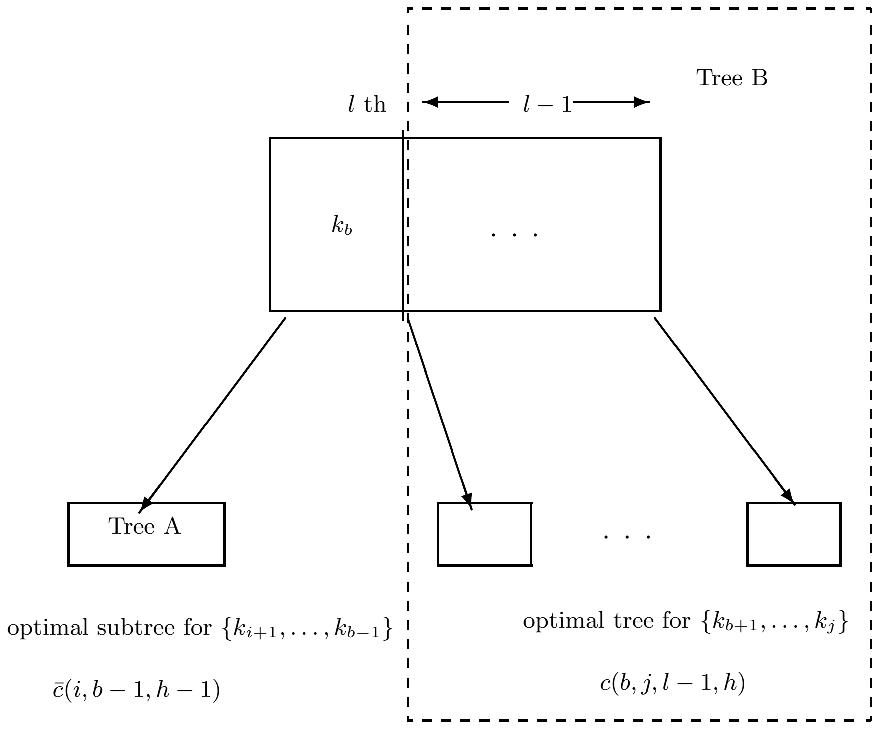
Tree B
l th
$l-1$
k_b
. . .
Tree A
. . .
optimal subtree for $\{k_{i+1},\ldots,k_{b-1}\}$
$\bar{c}(i,b-1,h-1)$
optimal tree for $\{k_{b+1},\ldots,k_j\}$
$c(b,j,l-1,h)$

Fig. 1: Construction principle for optimal trees

from i to j and $w(i,j) := \sum_{\nu=i}^{j} q_\nu + \sum_{\nu=i+1}^{j} p_\nu$ is the weight of such a sequence. For a B-tree b with n keys and height h we define the *weighted path length* wpl(b) by

$$\mathrm{wpl}(b) := \sum_{i=1}^{n} p_i \,\mathrm{level}(k_i) + h \sum_{i=0}^{n} q_i$$

where level(k_i) is the level of the node which contains key k_i. The level of the root is defined to be one. The weighted path length is the expected number of visits in a search multiplied by w. The problem is now to find a B-tree b of order k for n keys, that minimizes the weighted path length wpl(b). Such a tree is defined to be an *optimal B-tree.*

3. Review of existing algorithms

The best algorithms known so far for the problem defined in Section 2 are adaptations of the corresponding algorithms to construct optimal multiway search trees, cf. [7, 2]. These algorithms themselves are extensions of Knuth's algorithm for the construction of optimal binary search trees.

Figure 1 shows the basic principle underlying all these algorithms. Trees for larger sets of keys are constructed by joining trees for smaller sets. An optimal tree for the set $\{k_i,\ldots,k_j\}$ with exactly l keys in the root is constructed by joining an optimal tree for the set $\{k_{b+1}, \ldots, k_j\}$ with exactly $l-1$ keys in the root and an optimal tree for the set $\{k_i, \ldots, k_{b-1}\}$ in an optimal way. That means we have to find a value for b that minimizes the weighted path length of the resulting tree.

If we want to use this principle for B-trees, we have to notice some additional constraints. First, subtrees of any node must have the same height. This leads to the following restriction: if tree B of Figure 1 has height h we can use for tree A only trees of height $h-1$. Second, optimal B-trees may not have a fully occupied root in contrast to multiway trees. For them it can be proven that there always exists an optimal tree that has a fully occupied root. Third, an optimal B-tree does not necessarily have minimal height. So we have to take into consideration every possible height of a B-tree of order k with n keys, that means every possible value between h_l and h_u.

Taking the basic principle shown in Figure 1 and paying attention to the additional constraints leads to the following recursive formulas. With $c(i,j,l,h)$ we denote the weighted path length of an optimal B-tree for the set $\{k_{i+1}, \ldots, k_j\}$ that has height h and exactly l keys in the root. This is the weighted path length for the sequence $pq(i,j)$. $\bar{c}(i,j,h)$ is the weighted path length of an optimal subtree with height h for $\{k_{i+1}, \ldots, k_j\}$. Subtrees must have at least k keys in their root, see point (2) in Definition 1. We define $\hat{c}(h)$ as the weighted path length of an optimal B-tree for $\{k_1, \ldots, k_n\}$ with height h. $\tilde{c}$ is the weighted path length of an optimal B-tree for $\{k_1, \ldots, k_n\}$. We have the following formulas (cf. [7]):

(i) $c(i,i,l,1) = \infty, 0 \le i \le n, 1 \le l \le 2k$

(ii) $c(i,j,l,1) = \begin{cases} w(i,j) & \text{if } l = j-i \\ \infty & \text{otherwise} \end{cases} \quad 0 \le i < j \le n, 1 \le l \le 2k$

(iii) $\bar{c}(i,j,h) = \min_{k \le l \le 2k} c(i,j,l,h) + w(i,j), 0 \le i < j \le n, 1 \le h \le h_u$

(iv) $c(i,j,1,h) = \min_{i<b<j} (\bar{c}(i,b-1,h-1) + p_b + \bar{c}(b,j,h-1)),$
$0 \le i < j \le n, 2 \le h \le h_u$

(v) $c(i,j,l,h) = \min_{i<b<j} (\bar{c}(i,b-1,h-1) + p_b + c(b,j,l-1,h)),$
$0 \le i < j \le n, 2 \le l \le 2k$

(vi) $\hat{c}(h) = \min_{1 \le l \le 2k} c(0,n,l,h), 1 \le h \le h_u$

(vii) $\tilde{c} = \min_{h_l \le h \le h_u} \hat{c}(h)$

Formulas (i) and (ii) initialize the recursion. Formulas (iv) and (v) state the construction principle. Formula (iii) computes the weighted path length of an optimal subtree for $\{k_i, \ldots, k_j\}$. We have to add $w(i,j)$, because the level of each key and gap weight in the subtree will be increased by one, if we use an optimal tree as subtree of another tree. (vi) and (vii) are the definitions of $\hat{c}$ and $\tilde{c}$. To get the minimum weighted path length in (vii), we have to examine all possible heights between h_l and h_u.

Now we give the algorithm (cf. [7, 3]):

ALGORITHM 1:

```
for $i \leftarrow 1$ to $n$ do
    $w(i,i) \leftarrow q_i$
    $c(i,i,l,1) \leftarrow \infty$
    for $j \leftarrow i+1$ to $n$ do
        $w(i,j) \leftarrow w(i,j-1) + p_j + q_j$
        for $l \leftarrow 1$ to $2k$ do
            if $l = j - i$ then
                $c(i,j,l,1) \leftarrow w(i,j)$
                $r[i,j,l,1] \leftarrow i+1$
            else
                $c(i,j,l,1) \leftarrow \infty$
$\hat{c}(1) \leftarrow min_{1 \le l \le 2k}\, c(0,n,l,1)$
$h_u \leftarrow 1 + \log((n+1)/2)/\log(k+1)$
for $h \leftarrow 2$ to $h_u$ do
    for $i \leftarrow n$ downto $0$ do
        for $j \leftarrow i$ to $n$ do
            $\bar{c}(i,j,h-1) \leftarrow \min_{k \le l \le 2k} c(i,j,l,h-1) + w(i,j)$
            forall $i < b < j$ do
                compute $\bar{b}$ which minimizes
                $\bar{c}(i,b-1,h-1) + p_b + \bar{c}(b,j,h-1)$
            $r[i,j,1,h] \leftarrow \bar{b}$
            for $l \leftarrow 2$ to $2k$ do
                forall $i < b < j$ do
                    compute $\bar{b}$ which minimizes
                    $\bar{c}(i,b-1,h-1) + p_b + c(b,j,l-1,h)$
                $c(i,j,l,h) \leftarrow \bar{c}(i,\bar{b}-1,h) + p_{\bar{b}} + c(b,j,l-1,h)$
                $r[i,j,l,h] \leftarrow \bar{b}$
    $\hat{c}(h) \leftarrow \min_{1 \le l \le 2k} c(0,n,l,h)$
$\tilde{c} \leftarrow \min_{h_l \le h \le h_u} \hat{c}(h)$
```

After the termination of the algorithm the multidimensional array r contains the information to construct the optimal B-tree. $r[i,j,l,h]$ defines the leftmost key in the root of an optimal tree of height h for $\{k_{i+1},\ldots,k_j\}$ with exactly l keys in the root. From the nesting of the loops it is clear that the algorithm has a time complexity of $O(k\,n^3 \log n)$.

4. The new approach

As mentioned before the construction principle underlying the algorithm of the last section has originally been used for optimal multiway trees. In contrast to multiway trees the height of B-trees is bounded by $O(\log n)$ and their structure is much more restricted than the structure of multiway trees. These aspects are not considered by Algorithm 1. We present a sequential decision approach where keys are placed directly on some level $l (1 \le l \le h_u)$

of the tree. Using this approach yields a linear iteration over the keys instead of a cubic one (i, j, b) as in Algorithm 1. The crucial question is whether the number of trees that have to be examined for an optimal assignment of k_i is bounded drastically enough by the B-tree restrictions, so that overall we get an improved running time.

We now model the process of constructing an optimal B-tree as a decision problem with n stages. For every key k_i we have to decide, on which level this key should be placed. Whether placing on some level is feasible, depends on the former decisions for the keys k_1 to k_{i-1}, which define a certain state in the decision process. Then placing the key k_i on any level results in an increasing weighted path length and a new state. The amount of increasing as well as the new state depend on our decision.

Using this approach, the optimal B-tree is the result of a sequence of optimal decisions starting in a unique initial state. This leads to a dynamic program DP of the form $DP = (S_\nu, A_\nu, D_\nu, T_\nu, c_\nu, C_{n+1})$,where n is the number of the stages of DP, S_ν is the *state set* of stage $\nu, 1 \leq \nu \leq n+1$, and A_ν is the *decision set* of stage $\nu, 1 \leq \nu \leq n$. The sets $D_\nu \subseteq S_\nu \times A_\nu$ define the *feasible decisions* for the states of stage ν. It holds: $(s, a) \in D_\nu$, if and only if a is feasible in state s on stage ν. The set $D_\nu(s) := \{a \in A_\nu | (s, a) \in D_\nu\}$ contains all feasible decisions for state s on stage ν. $T_\nu : D_\nu \to S_{\nu+1}$ is the *transition function.* Making decision a in state s at stage ν results in state $T_\nu(s, a)$ at stage $\nu + 1$. $c_\nu : D_\nu \to \mathbb{R}$ is the *cost function* of stage ν. $c_\nu(s, a)$ gives the costs that arise if we decide to make decision a in state s on stage ν. $C_{n+1} : S_{n+1} \to \mathbb{R}$ is the *terminal cost* function. $C_{n+1}(s)$ gives the costs that arise if our final state is s.

Now we have to define the components of the dynamic program in such a way that the decision process models the construction of a B-tree.

First we give the definition of the states. For motivation take a look at Figure 2. Suppose we have $k = 2$. Then we have to place the first two keys in the first block. This results in (a) of Figure 2. Now for k_3 we have two possibilities. We may place k_3 in the same block as the former keys (b) or we may create a new root, which leads to (c). In the latter case it is not feasible to place one of the following keys in the block of $\{k_1, k_2\}$. Instead we have to place k_4 and k_5 in a new block on leaf level. Doing this we get (d). Now there are again two choices for k_6. On the other hand in situation (e) we have to place k_8 in the root, because the leaf already contains $2k$ keys. For assigning k_{10} it doesn't matter if we have tree (f) or tree (g). For both trees we have the choices to create a new root with k_{10} as the first key or to place k_{10} on the actual root or leaf level. The rules that we have implicitly used in this example have led to trees that are valid B-trees with the exception of the rightmost path. Such trees will be filtered on the last stage $(n + 1)$ by the terminal cost function C_{n+1}. For instance tree (c) gets the terminal costs ∞ for $n = 3$.

From the examples above we can deduce, that for a correct placing of a key in the partial tree only the occupation of the blocks in the rightmost path from the root to the leaf is relevant. Due to this fact we can represent

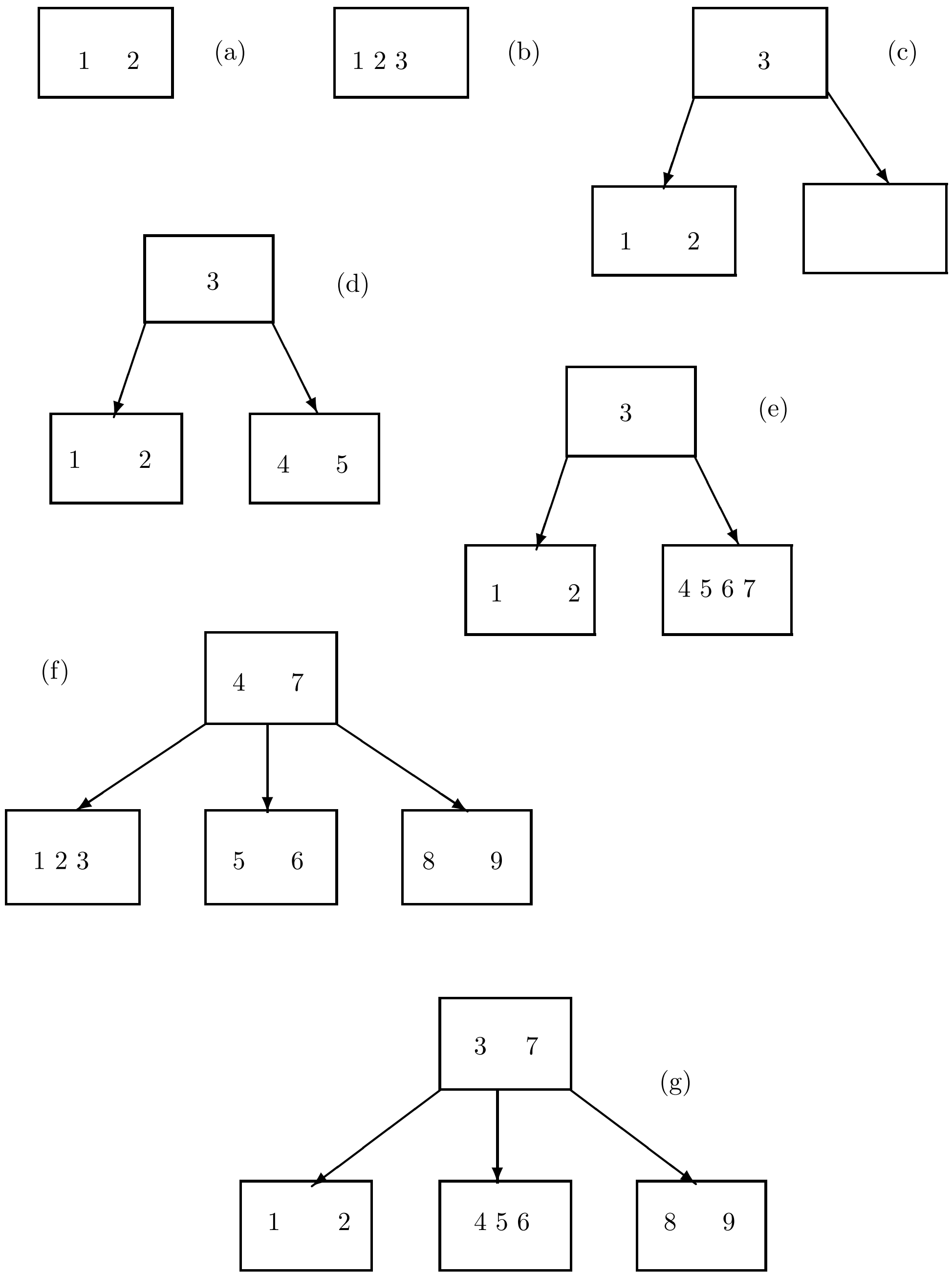

Fig. 2: B-tree states in the construction process

a state $s \in S_\nu$ by a vector with h_s components, that means $s = (s_1, \ldots, s_{h_s})$ with $0 \leq s_i \leq 2k$ resp. $s \in \mathbb{N}_{2k}^{h_s}$. In the following we will use h_s as the denotation for the *component number* of a state. Each component s_i gives the number of keys in the block of level i in the rightmost path of the tree. For instance the state resulting from tree (d) in Figure 2 is represented by $(1, 2)$ and the state resulting from tree (g) by $(2, 2)$. The set S_ν is defined to be the set of all vectors that are possible after the assignment of $\nu - 1$ keys. A more formal definition of the state sets S_ν follows below.

A decision is characterized by the level on which a key is placed. So we define $A = A_\nu = \{0, \ldots, h_u\}$. Making decision $a = 0$ means creating a new root, as in the transition from (a) to (c) of Figure 2. $a \geq 1$ means that the corresponding key is placed on level a. For instance, the tree (g) is constructed by the decision sequence $DS = (0, 1, 0, 2, 2, 2, 1, 2, 2)$, assuming that $s_1 = ()$ (i.e. $h_s = 0$) is the initial state. Why we start with $s_1 = ()$ and not with $s_1 = (0)$ is explained below.

Let $s = (s_1, \ldots, s_{h_s})$ be a state. A feasible decision a has to fulfill the following conditions:

(i) $0 \leq a \leq h_s$

(ii) $s_\mu \geq k, a + 1 \leq \mu \leq h_s$

(iii) $a = 0 \vee s_a < 2k$

Condition (ii) is due to (2) of Definition 1 and (iii) is due to (1) of Definition 1. (i) guarantees the consistency of state s. So we can define $D_\nu = \{(s, a) | s \in S_\nu, a \text{ fulfills (i) to (iii)}\}$. Observe that the feasible decisions of a state s are independent of the stage ν. So we define $D(s) = \{a \in A | a \text{ fulfills (i) to (iii)}\}$ as the *set of feasible decisions for state s*. For every B-tree there exists a unique feasible decision sequence that constructs the tree. As an example see the decision sequence to construct tree (g) above. Using these definition each feasible decision sequence leads to trees that are valid B-trees with the exception of the rightmost path. Such trees are filtered by the terminal cost function C_{n+1}.

As explained before, making a decision a has two effects. First, the block on level a of the rightmost path gets one additional key and second, the blocks on the levels from $a + 1$ to h_s become closed. That means we cannot assign following keys to these blocks. We get new empty blocks for these levels as in (c) of Figure 2. So the definition for the transition function is:

$$T_\nu(s, a) = T(s, a) = \begin{cases} (1, 0, \ldots, 0) \in \mathbb{N}_{2k}^{h_s+1} & \text{if } a = 0 \\ (s_1, \ldots, s_{a-1}, s_a + 1, 0, \ldots, 0) \in \mathbb{N}_{2k}^{h_s} & \text{if } 1 \leq a \leq h_s \end{cases}$$

With this definition of T_ν the state sets S_ν can be formally defined by:

$$\begin{aligned} S_1 &= \{()\} \\ S_{\nu+1} &= T(D_\nu),\ \nu = 1, \ldots, n \end{aligned}$$

The cost functions c_ν are defined by:

$$c_\nu(s, a) = \begin{cases} h_s\, q_\nu + a\, p_\nu & \text{if } a > 0 \\ (h_s + 1) q_\nu + p_\nu + w(0, \nu - 1) & \text{if } a = 0 \end{cases}$$

If we create a new root ($a = 0$), the tree constructed so far becomes a subtree of the new root. In that case the levels of the previous keys and gaps increase by one. This results in an additional weighted path length of $w(0, \nu - 1)$. Observe that by using $s_1 = ()$ the first gap weight q_0 is treated correctly.

The terminal costs C_{n+1} model whether our final state fulfills the B-tree conditions, especially condition (2) of Definition 1. For instance, tree (c) of Figure 2 is not a valid B-tree for $n = 3$. So we have to verify whether the rightmost path contains underfull nodes. We have:

$$C_{n+1}(s) = \begin{cases} 0 & \text{if } s_\nu \geq k, 2 \leq \nu \leq h_s \\ \infty & \text{otherwise} \end{cases}$$

Now the definition of the dynamic program DP is complete. Using this definition the optimization problem is

$$F := \sum_{\nu=1}^{n} c_\nu(s_\nu, a_\nu) + C_{n+1}(s_{n+1}) \rightarrow \min$$

subject to:

$$\begin{aligned} & s_1 = () \\ & a_\nu \in D(s_\nu), 1 \leq \nu \leq n \\ & s_{\nu+1} = T(s_\nu, a_\nu), 1 \leq \nu \leq n \end{aligned}$$

The value F of the objective function yields the minimum weighted path length. The tree is given by the optimal sequence of feasible decisions.

For the solution of this optimization problem we use a common dynamic programming algorithm, cf. [5].

Algorithm 2:

```
**forall** $s \in S_{n+1}$
    $W(s) \leftarrow C_{n+1}(s)$
**for** $\nu \leftarrow n$ **downto** 1 **do**
    **forall** $s \in S_\nu$ **do**
        **forall** $a \in D(s)$ **do**
            compute $\bar{a}$ that minimizes $c_\nu(s, a) + W(T(s, a))$
        $V(s) \leftarrow c_\nu(s, \bar{a}) + W(T(s, \bar{a}))$
        $\pi_\nu(s) \leftarrow \bar{a}$
    $W \leftarrow V$
$s \leftarrow ()$
$\tilde{c} \leftarrow 0$
**for** $\nu \leftarrow 1$ **to** $n$ **do**
    $a_\nu \leftarrow \pi_\nu(s)$
    $\tilde{c} \leftarrow \tilde{c} + c_\nu(s, a_\nu)$
    $s \leftarrow T(s, a_\nu)$
```

The result of the algorithm is a sequence $DS = (a_1, \ldots, a_n)$ that defines the optimal tree. Having DS we are able to build the corresponding tree in linear time, as for each key k_ν the level where k_ν has to be placed is given by the sequence of the a_ν.

5. Complexity results

Now we have to prove that solving our DP using Algorithm 2 yields the above mentioned running time. Cornerstones of this proof are bounds for the cardinalities of the state sets S_ν and the sets D_ν that define the feasible decisions.

THEOREM 2. *For all state sets S_ν we have:*

$$|S_\nu| = O(n^\beta) \text{ with } \beta = 1 + \frac{\log 2}{\log(k+1)}, \nu = 1, ..., n+1$$

PROOF. Let $S := \{0, \ldots, 2k\}^{h_u}$ and let $\tilde{S} := \cup_{\nu=1}^{n+1} S_\nu$. The function $f : \tilde{S} \to S$ defined by

$$f(s) = (0, \ldots, 0, s_1, \ldots, s_{h_s}) \in \mathbb{N}_{2k}^{h_u}$$

is an injective mapping from $\tilde{S}$ to S. The vector $f(s)$ is simply constructed from s by adding leading $h_u - h_s$ zeros. Observe that the first component s_1 of a state $s \in \tilde{S}$ is always positive. Due to the injectivity of the mapping we get

$$|S_\nu| \leq |S| = (2k+1)^{h_u}, \nu = 1, ..., n+1$$

Using Theorem 1 we get

$$\begin{aligned} |S| &\leq (2k+1)^{1+\log(\frac{n+1}{2})/\log(k+1)} \\ &= (2k+1) \exp\left(\log(\frac{n+1}{2}) \frac{\log(2k+1)}{\log(k+1)}\right) \end{aligned}$$

Using $\log(2k+1) \leq \log 2 + \log(k+1)$ we get

$$|S| \leq (2k+1) \left(\frac{n+1}{2}\right)^{1+\log(2)/\log(k+1)} . \quad \square$$

The next theorem states that the cardinality of the feasible decisions is bounded by the same function as the cardinality of the states.

THEOREM 3. *Let $D := \cup_{\nu=1}^{n+1} D_\nu$. Then we have:*

$$|D| = O(|S|)$$

PROOF. We have $D \subset \tilde{S} \times A$ and

$$|D| = \sum_{s \in \tilde{S}} |D(s)|$$

Now we define a partition of S by

$$R_d := \{s \in S | k \le s_{h_u-d+2}, \dots, s_{h_u} \le 2k \wedge (0 < s_{h_u-d+1} < k \vee h_u = d)\}, \quad d = 1, \dots, h_u$$

i.e. R_d contains the vectors of s that have exactly $d-1$ components equal to or greater than k at the backend. Observe that this is a necessary condition for a state to have d feasible decisions (see the definition of $D(s)$). Because the definition of R_d relaxes the conditions given in the definition of $D_\nu(s)$ we have

$$(\tilde{s} \in \tilde{S} \wedge |D(\tilde{s})| = d \wedge f(\tilde{s}) \in R_e) \Rightarrow d \le e$$

i.e. a state $s \in \tilde{S}$ which has exactly d feasible decisions is mapped to some vector $f(\tilde{s}) \in R_e$ with $d \le e$. Using that the mapping is injective yields

$$|D| \le \sum_{d=1}^{h_u} d\,|R_d|$$

Using the definition of R_d we get

$$|R_d| = \frac{k}{2k+1}\left(\frac{k+1}{2k+1}\right)^{d-1} |S|, d = 1, \dots, h_u - 1$$

and $|R_{h_u}| = ((k+1)/(2k+1))^{h_u}|S|$. Extending the upper bound of the sum to ∞ yields

$$|D| \le |S| \frac{k}{2k+1} \sum_{d=0}^{\infty} (d+1) \left(\frac{k+1}{2k+1}\right)^d$$

$1/2$ is an upper bound for the second term and $2/3$ is an upper bound for $(k+1)/(2k+1)$. To compute the sum we examine the power series $\sum_{\nu=0}^{\infty} (\nu+1)x^\nu$. We have

$$\sum_{\nu=0}^{\infty} (\nu+1)x^\nu = \frac{d}{dx} x \sum_{\nu=0}^{\infty} x^\nu = \frac{1}{(1-x)^2}$$

Using the upper bounds we get:

$$|D| \le \frac{9}{2}|S| . \ \square$$

Now we consider Algorithm 2. There are nested loops over ν, S_ν, and $D(s)$ for each $s \in S_\nu$. Using Theorems 2 and 3 we get the result that Algorithm 2 will have running time $O(k\,n^{1+\beta})$ if we are able to compute the remaining statements efficiently enough. But this is simple. We map the states to components of an array AS. The function

$$\mathrm{index}(s) = \sum_{\nu=1}^{h_u} s_\nu (2k+1)^{h_u - \nu}$$

defines a bijective mapping between S and the index domain of AS. Each component $AS(i)$ of AS represents a certain state s. Attached to this component is an array AD that represents $D(s)$. Each component of AD contains a pointer to another state. This pointer implements the transition function $T(s, a)$. Due to this, we are able to compute $W(T(s, a))$ in $O(1)$. Initialization of these data structures needs time $O(|D|)$. Moreover, it is not necessary to use exactly the set S_ν at stage ν. Because of the fact that the states $\bar{S}_\nu \setminus S_\nu$ can never be reached in the forward computation of Algorithm 2, the Algorithm remains correct if some superset $\bar{S}_\nu \supset S_\nu$ is used, for instance $\bar{S}_\nu := S$. For this reason we do not need any precomputation of the state sets S_ν. But even if we did, we would not need more than $O(k\,n^{1+\beta})$ time by using the above described data structure. Putting all together yields:

Theorem 4. *Algorithm 2 has running time $O(k\,n^{\alpha})$ with $\alpha = 2 + \log 2/\log(k+1)$.*

6. Summary

We have presented a new algorithm to construct optimal weighted B-trees. The key to the improvement has been the formulation of a dynamic program that models the construction of the tree in a decision-oriented way. In the model we have to decide key by key, the level on which the key should be placed. The B-tree conditions are included by additional constraints and a terminal cost function.

One idea for further improvement is to use an A^* algorithm in combination with a strong admissible estimation function, cf. [6]. In this case we would use a forward computation instead of the backward computation used in Algorithm 2. A state would be modelled as in our model but with an additional component that gives the number of keys already assigned. That means a state would correspond to a state-stage pair of our model. Expanding a state then means adding a key to the tree that is modelled by the state. The main difference to the former approach is that we may get a strong lower bound for the cost of the final tree by using an adequate admissible estimation function. In this way we may achieve a better performance, because a lot of states would never be expanded. Of course we can use $w(.,.)$ as estimate function, but the performance of $w(.,.)$ is poor, especially for increasing n.

References

[1] R. Bayer, E. M. McCreight, Organization and Maintenance of Large Ordered Indexes, *Acta Informatica* **1**, pp. 173–189, 1972.

[2] L. Gotlieb, Optimal Multi-Way Search Trees, *SIAM Journal of Computing* **10**, pp. 422–433, 1981.

[3] S.-H. S. Huang, V. Viswanathan, On the Construction of Weighted Time-Optimal B-Trees, *BIT* **30**, pp. 207–215, 1990.

[4] D. E. Knuth, Optimum Binary Search Trees, *Acta Informatica* **1**, pp. 14-25, 1971.

[5] K. Neumann, M. Morlock, *Operations Research*, Hanser, Munich, 1993.

[6] J. Pearl, *Heuristics — Intelligent Search Strategies for Computer Problem Solving*, Addison-Wesley, 1984.

[7] V. K. Vaishnavi, H. P. Kriegel, D. Wood, Optimum Multiway Search Trees, *Acta Informatica* **14**, pp. 119–133, 1980.