A Simple Algorithm for Optimal Search Trees with Two-way Comparisons

MAREK CHROBAK, Department of Computer Science and Engineering, University of California at Riverside, USA
MORDECAI GOLIN, Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Hong Kong
J. IAN MUNRO, Cherton School of Computer Science, University of Waterloo, Canada
NEAL E. YOUNG, Department of Computer Science and Engineering, University of California at Riverside, USA

We present a simple $O(n^4)$-time algorithm for computing optimal search trees with two-way comparisons. The only previous solution to this problem, by Anderson et al., has the same running time but is significantly more complicated and is restricted to the variant where only successful queries are allowed. Our algorithm extends directly to solve the standard full variant of the problem, which also allows unsuccessful queries and for which no polynomial-time algorithm was previously known. The correctness proof of our algorithm relies on a new structural theorem for two-way-comparison search trees.

CCS Concepts: • **Theory of computation → Sorting and searching;**

Additional Key Words and Phrases: Data structures, algorithms, optimal search trees

ACM Reference format:
Marek Chrobak, Mordecai Golin, J. Ian Munro, and Neal E. Young. 2021. A Simple Algorithm for Optimal Search Trees with Two-way Comparisons. *ACM Trans. Algorithms* 18, 1, Article 2 (December 2021), 11 pages.
https://doi.org/10.1145/3477910

See Reference [5, Theorem 1] for a conference version of the result in this article. The algorithm in that version is more complicated, and its analysis is less informative. See References [6, 8, 9] for improved versions of other results in Reference [5].

Marek Chrobak research supported by NSF Grants No. CCF-1217314 and No. CCF-1536026.
Mordecai Golin research funded by HKUST/RGC Grant No. FSGRF14EG28 and RGC CERG Grants No. 16208415 and No. 16213318.
J. Ian Munro research funded by NSERC and the Canada Research Chairs Programme.
Neal E. Young research supported by NSF Grant No. IIS-1619463.

Authors' addresses: M. Chrobak and N. E. Young, Department of Computer Science and Engineering, University of California at Riverside, Riverside, CA 92521, USA; emails: marek@cs.ucr.edu, neal.young@ucr.edu; M. Golin, Department of Computer Science and Engineering, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong; email: golin@cse.ust.hk; J. I. Munro, Cherton School of Computer Science, University of Waterloo, Waterloo, ON N2L3G1, Canada; email: imunro@uwaterloo.ca.

1549-6325/2021/12-ART2
https://doi.org/10.1145/3477910

1 INTRODUCTION

Search trees are among the most fundamental data structures for storing and accessing data. *Static* search trees are used in applications where the set of possible queries and their frequencies are known in advance, in which case a single tree can be precomputed and used throughout the application to handle lookup queries. Such a tree is *optimal* if it minimizes the expected search cost. The problem of computing optimal search trees has been studied extensively in various forms since the 1960s. Perhaps the most famous example is Knuth's $O(n^2)$-time algorithm [19], which is widely considered one of the gems of algorithmics and has inspired the discovery of general techniques for speeding up dynamic-programming algorithms [3, Section 7.1].

But most optimal-search-tree results, including Knuth's, consider only trees based on three-way comparisons [19]. In contrast, standard high-level programming languages implement two-way comparisons (e.g., $=$, $\leq$, $<$). As pointed out by Knuth in the second edition of *"The Art of Computer Programming,"* it is desirable to understand search trees that use only two-way comparisons [20, Section 6.2.2, Example 33]. Yet they are still not well understood: There is only one published proof that optimal two-way-comparison trees can be computed in polynomial time, given decades after Knuth's result, in a breakthrough by Anderson et al. [1], who gave an $O(n^4)$-time algorithm for the *successful-queries* variant, in which only searches for keys stored in the tree are supported. Their work was motivated by an application to efficient message dispatching in object-oriented programming languages, addressed earlier by Chambers and Chen [4] (see also Reference [21]) who provided an $O(n^2)$-time heuristic (non-optimal) algorithm for constructing two-way-comparison trees. Independently, Andersson [2] had earlier looked at speeding up search trees by replacing three-way comparisons with two-way ones.

Difficulties introduced by two-way comparisons. Given a query q, the search for q in a search tree starts at the root and traces a path to a leaf, at each node comparing q to the node's key, then following the edge corresponding to the outcome. A *three-way* comparison has three possible outcomes: the query is less than, equal to, or greater than the node's key. In any three-way-comparison tree, the set of queries reaching any internal node N is an open interval between some two keys. Each subtree solves a subproblem defined by some such inter-key interval. This leads to a simple dynamic program with $O(n^2)$ subproblems and to a simple $O(n^3)$-time algorithm, the derivation of which is a standard exercise [10, Section 15.5], [11, Example 6.20]. Knuth's celebrated result improves the running time to $O(n^2)$.

In contrast (as noted, e.g., in Reference [1]), algorithms for *two-way*-comparison search trees face the following obstacle. After a few equality tests, the subproblem that remains is defined by some inter-key interval with *holes*—each hole corresponding to an earlier equality test. (For example, in tree (b) of Figure 1, the set of queries reaching node $\langle q < 5\rangle$ is $[3, \infty) - \{7\}$.) The resulting dynamic-programming formulation can have $\Theta(2^n)$ subproblems, as the query sets that can arise can contain any subset of the keys.

The most-likely-key property. To achieve their polynomial-time algorithm for two-way comparison trees, Anderson et al. circumvent this obstacle in two steps. First, using an elegant *side-weights* argument, they show that every instance has an optimal tree T with what we call the *most-likely-key property*:

(MLK) *In each equality-test node N in T, the test is to some key of maximum weight among keys reaching N.*

Their proof is for the successful-queries variant but extends directly to the unrestricted variant. The result implies that it suffices to consider only trees with the MLK property.

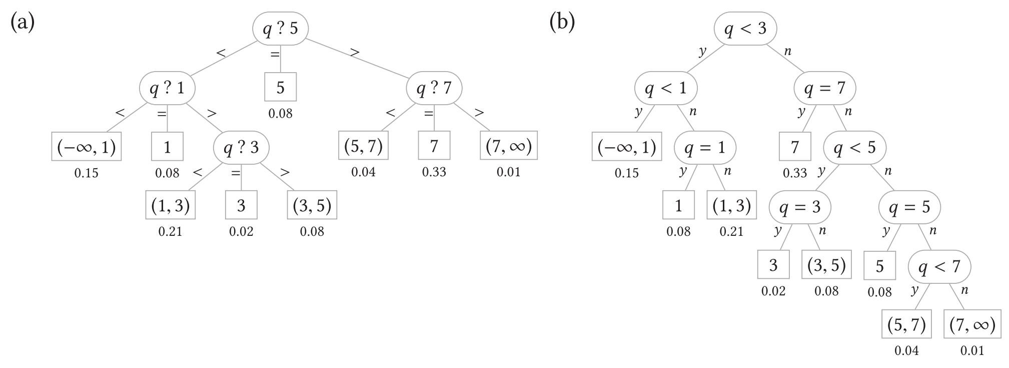

Fig. 1. (a) A *three-way-comparison* search tree and (b) a *two-way-comparison* tree. Both trees have keys $\{1, 3, 5, 7\}$. The number below each leaf represents its frequency. The cost of tree (a) is 2.23, the cost of tree (b) is 2.8.

If all key weights are distinct, then, at any node N in such a tree T, the set of queries that reach N is an inter-key interval with the g heaviest keys removed, for some $g \in \{0, 1, \ldots, n\}$. There are $O(n^3)$ such query sets (intervals with such keys removed) so this yields a dynamic program with $O(n^3)$ subproblems, and a simple $O(n^4)$-time algorithm for instances with distinct weights.

When the weights are not distinct, however, the MLK property gives no guidance about how to break ties. Even in trees with the MLK property, exponentially many distinct query sets can arise, so the dynamic program still has exponential size. To work around this counter-intuitive obstacle, Anderson et al. then resort to a lengthy argument that establishes "thresholds on the frequency of an item that can occur as an equality comparison at the root of an optimal tree." This significantly complicates their analysis and their algorithm.

New results. Our first result is a structural theorem (Theorem 1), which implies that there is an optimal tree with what we call the *refined MLK property*:

(RMLK) *In each equality-test node N in T, the test is to the key k_b of maximum weight among keys reaching N, breaking ties by maximizing the index b.*

Breaking ties by the index b (above) is for convenience, in fact any tie-breaking rule will work.

The precise statement of Theorem 1 is somewhat delicate. For example, it is *not* the case that any tree with an equality test at the root can be converted into an equally good tree with an equality test to any given maximum-weight key. (See the remarks following the proof of the theorem.) But, with the correct theorem statement in hand, the proof is straightforward, by augmenting the side-weights argument of Anderson et al. with a perturbation argument.

The theorem implies that, whether or not the weights are distinct, it suffices to consider trees with the RMLK property. Restricting to such trees yields a simple dynamic program with $O(n^3)$ subproblems, each solvable in $O(n)$ time, yielding an $O(n^4)$-time algorithm (Corollary 1). The algorithm is essentially the algorithm for distinct-weights instances, modified to break ties arbitrarily. Our formulation is simple and easy to implement—Appendix B gives a 22-line implementation in Python. This is the first polynomial-time algorithm to be proven correct for the unrestricted variant.

Other related work. In 1994, Spuler [22, 23] suggested an algorithm for computing optimal two-way comparison trees, based on a result by Huang et al. [17] for a different type of search tree. But Huang et al.'s related result, and the recurrences underlying Spuler's proposed algorithm, are demonstrably wrong [8]. Independently, Spuler also conjectured that optimal trees have the MLK property, and based on that conjecture proposed an additional algorithm, without proof of correctness, running in time $O(n^5)$. As discussed earlier, in 2002 Anderson et al. proved the conjecture for the successful-queries variant and obtained an $O(n^4)$-time algorithm for it. To our knowledge, this is the only previously published proof of correctness for any polynomial-time algorithm for optimal two-way-comparison search trees, for any variant with equality tests and inequality comparisons.

The technical obstacle introduced by non-distinct weights also arises in finding optimal so-called *binary split trees*: for distinct-weight instances, an optimal binary split tree can be found in $O(n^4)$ time, but for arbitrary instances the best bound known is $O(n^5)$ [14]. For *generalized binary split trees* (see References [8, 17]) no correct polynomial-time algorithm is yet known.

There are various other models for comparison-based search trees in the literature. As one example, one can consider only inequality comparisons. In trees with three-way comparisons that would correspond to giving every key weight zero (so only non-key queries have positive probability). As noted by Knuth, this model is equivalent to alphabetic coding, solved by Gilbert and Moore in $O(n^3)$ time [13], later improved to $O(n \log n)$ time by Hu and Tucker [15] and, independently, by Garsia and Wachs [12]. As noted by Anderson et al. this is also equivalent to finding the optimal two-way-comparison tree in the successful-queries case that only uses "<" comparisons. For more background information on different models and algorithms (including approximations) for search trees, we refer the reader to the 1997 tutorial by Nagaraj [21].

Formal problem statement. Let $k_1, k_2, \ldots, k_n$ denote the n given keys, in increasing order. Let $\beta_b \geq 0$, for $b = 1, 2, \ldots, n$, be the weight of key k_b and $\alpha_a \geq 0$, for $a = 0, 1, \ldots, n$, be the weight of interval (k_a, k_{a+1}), where for convenience k_0 and k_{n+1} (which are not keys) represent $-\infty$ and ∞. In standard applications the weights represent query frequencies or probabilities. Here each weight can be an arbitrary non-negative number.

As illustrated in Figure 1(b), a *two-way-comparison search tree* for the instance is a rooted binary tree T, where each non-leaf node N represents a comparison ($=$ or $<$)[1] to some key k_b. T has $2n + 1$ leaves, n of which are associated one-to-one with the keys k_b and the remaining $n + 1$ with the intervals (k_a, k_{a+1}). Given any value q, the search for q in T must correctly *identify* q, in the following sense. The search starts at the root and proceeds down the tree, following the edges corresponding to the outcomes of the comparisons, and it terminates either at the leaf associated with a key k_b, if $q = k_b$, or at the leaf associated with the interval (k_a, k_{a+1}) containing q, if q is not a key.

For each node N in T, let T_N denote the subtree rooted at N. The *weight of subtree* T_N, denoted w_N, is the total weight of the keys k_b and intervals (k_a, k_{a+1}) in the set of queries reaching N (these are the keys and intervals associated with the leaves of T_N).

The *cost* of T is the weighted depth of its leaves, that is,

$$\mathrm{cost}(T) = \sum_{\ell \in \mathrm{leaves}(T)} \mathrm{depth}_T(\ell) \times w_\ell = \sum_{N \in T - \mathrm{leaves}(T)} w_N, \tag{1}$$

where $\mathrm{leaves}(T)$ is the set of leaves in T and $\mathrm{depth}_T(\ell)$ is the number of edges on the path from the root to ℓ in T.

A search tree T is called *optimal* for a given instance if it has minimum cost among all search trees for the given instance. The objective is to compute such an optimal tree.

[1]For simplicity of presentation, we do not allow the $>$ (or equivalently $\leq$) comparison. See the discussion below.

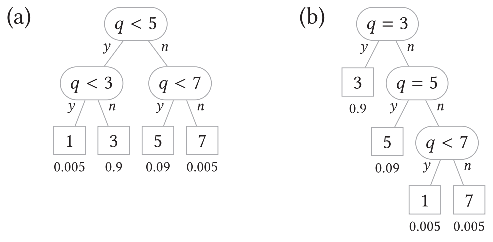

Fig. 2. Optimal two-way-comparison trees for the *successful-queries* variant (a) using only inequalities, and (b) when "="s are also permitted. Both trees have key set $\{1, 3, 5, 7\}$. The number below each leaf represents its frequency. The cost of tree (a) is 2, the cost of tree (b) is 1.11.

The successful-queries variant. The problem definition above is for the "full" variant, with unrestricted queries. In the *successful-queries* variant, illustrated in Figure 2, only queries to the given keys $k_1, k_2, \ldots, k_n$ are allowed. An instance is specified by just those keys and their frequencies $\beta_1, \beta_2, \ldots, \beta_n$, and the problem is to find the minimum cost of any tree that handles $\mathcal{S} = \{k_1, \ldots, k_n\}$. (Section 2 formally defines "handles.") While our algorithm's description in Section 2 assumes the full variant, it can be adapted directly to the successful-queries variant, retaining its running time of $O(n^4)$, by a straighforward modification of the dynamic program.[2] This adapted algorithm has the same asymptotic running time as Anderson et al.'s (who considered only the successful case), but is much simpler.

Other comparison operators. Similarly, for the sake of exposition, and following the model in Reference [1], we allow only the two comparison operators $\{<, =\}$. As explained later (see the end of Section 2), our algorithm and its analysis extend naturally to the model with comparison set $\{<, \leq, =\}$. Due to symmetries and equivalences, this is the only other set of operators of interest for the full variants of the problem (allowing unsuccessful queries) that include equality comparisons. For example, operator $\geq$ can be replaced by operator $<$ by swapping the subtrees of each node with the $\geq$ comparison. We note, in passing, that Hu and Tucker [16] have shown that the full variant with only comparison operators in $\{<, \leq\}$ allowed can be solved in time $O(n \log n)$ by reducing it to an instance of alphabetic coding with $2n + 1$ keys and applying the algorithm of Hu and Tucker [15].

2 THE DYNAMIC-PROGRAMMING ALGORITHM

Within any tree having the RMLK property, at any node N, the set $\mathcal{S}$ of queries reaching N has the following form: $\mathcal{S}$ consists of the queries lying in some inter-key interval, minus some number of heaviest keys removed. The algorithm will solve one subproblem for each set $\mathcal{S}$ of this form. Next, we explicitly define these sets and their subproblems.

Fix a permutation $k_{\pi(1)}, k_{\pi(2)}, \ldots, k_{\pi(n)}$ of the keys that orders them by nondecreasing weight, breaking ties arbitrarily. For each key $k_{\pi(r)}$, call r the *rank* of the key. A set $\mathcal{S}$ is *valid* if either (i) $\mathcal{S}$ is a singleton key, of the form $\mathcal{S} = \{k_b\}$, or (ii) $\mathcal{S}$ is an inter-key interval with some largest-rank keys removed—that is, for some $i, j \in \{0, 1, \ldots, n+1\}$ with $i < j$, and $h \in \{0, 1, \ldots, n\}$, the set $\mathcal{S}$ equals $S(i, j, h)$, defined as

$$S(i, j, h) \;=\; [k_i, k_j) - \left\{k_{\pi(h+1)}, k_{\pi(h+2)}, \ldots, k_{\pi(n)}, k_0\right\}.$$

[2]Note that the successful-queries variant is *not* a special case of the full variant, since, by definition, any search tree for this variant must contain leaves representing all inter-key intervals $(k_a,\, k_{a+1})$, even though their probabilities α_a are 0.

(These sets are not necessarily distinct—for example, it is possible that $S(i,j,h) = S(i,j,h')$ even though, say, $h \neq h'$. Removing k_0 is needed only to deal with the case when $i = 0$.)

Note that $S(i,j,h)$ is a union of some keys and "failure" intervals (k_a, k_{a+1}). Let $w(S(i,j,h))$ denote the total weight of these keys and intervals. Given a search tree T, if those keys and intervals are also the ones associated with T's leaves, say that T *handles* $S(i,j,h)$ (so, for each $q \in S(i,j,h)$, the tree correctly identifies q). Define $\text{opt}(i,j,h)$ to be the minimum cost of any tree that handles $S(i,j,h)$. Note that $S(0,n+1,n) = (-\infty,+\infty)$, so the goal is to return $\text{opt}(0,n+1,n)$. (If desired, then an optimal tree can also be constructed in the standard fashion.)

The algorithm computes $\text{opt}(0,n+1,n)$ using the following recurrence relation:

(a) If $j = i+1$ and $k_i \notin S(i,j,h)$, then $S(i,j,h) = (k_i, k_{i+1})$ so $\text{opt}(i,j,h) = 0$.
(b) If $h > 0$ and $k_{\pi(h)} \notin S(i,j,h)$, then $S(i,j,h) = S(i,j,h-1)$, so $\text{opt}(i,j,h) = \text{opt}(i,j,h-1)$.
(c) Else, $$\text{opt}(i,j,h) = w(S(i,j,h)) + \min \begin{cases} \text{opt}(i,j,h-1) \ \textit{(if } h > 0\textit{, else } +\infty\textit{)}, \\ \min\left\{ \text{opt}(i,b,h) + \text{opt}(b,j,h) : i < b < j \right\}. \end{cases}$$

Correctness. For any tree T that handles $S(i,j,h)$, note that $\text{cost}(T)$ is zero if T is a leaf, and otherwise $\text{cost}(T) = w(S(i,j,h)) + \text{cost}(T_L) + \text{cost}(T_R)$, where T_L and T_R are the left and right subtrees.

By inspection, the cases in the recurrence relation are exhaustive. Likewise, Cases (a) and (b) are correct. For Case (c), the two terms in the right-hand side correspond to creating a tree whose root does either (i) an equality-test with key $k_{\pi(h)}$ (if $h > 0$ so that $S(i,j,h)$ contains some keys, with $k_{\pi(h)}$ being the one of largest rank), or (ii) an inequality comparison with some key k_b. By Theorem 1 in Section 3 there is an optimal tree of this form that handles $S(i,j,h)$, so the recurrence is correct in Case (c). (Note that if $j = i+1$, the range of the minimum in the second term is empty, and there no trees of type (ii) above. In this case, we take the minimum to be infinite, so the recurrence remains correct.)

Running time. The number of subproblems $S(i,j,h)$ is $O(n^3)$. For each subproblem, by inspection of the recurrence, $\text{opt}(i,j,h)$ can be computed in $O(n)$ time. Thus, the running time is $O(n^4)$. This proves Corollary 1:

Corollary 1. *There is an $O(n^4)$-time algorithm for computing optimal two-way-comparison search trees (with unrestricted queries).*

Extension to operators $\{=,<,\leq\}$. The algorithm and its proof of correctness extend naturally to the model with three comparison operators, $\{=,<,\leq\}$, potentially allowing lower tree costs (even for the case $n = 1$). The overall principle of the algorithm remains the same, although the presence of two inequality operators introduces minor technical complications. The modified algorithm will have four types of subproblems that correspond to four types of inter-key intervals: (k_i, k_j), $[k_i, k_j)$, $(k_i, k_j]$, and $[k_i, k_j]$, each with some number h of holes. In the recurrence for each type of subproblem, the minimum in Case (c) will have three choices that correspond to operators $=$, $<$, and $\leq$. As for the correctness proof, the statement of Lemma 1 needs to be modified to say in part (b) that "The root of T does an inequality comparison." The proofs of Lemma 1, Lemma 2, and Theorem 1 apply as presented.

3 STATEMENT AND PROOF OF THEOREM 1

Fix the keys $k_1, k_2, \ldots, k_n$ and their associated weights. Let $\mathcal{S}$ be any valid set, and consider the subproblem of finding a minimum-cost (two-way-comparison) tree that handles $\mathcal{S}$.

Theorem 1. *Fix any key k_b of largest weight among keys in $\mathcal{S}$. Then some optimal tree T satisfies one of the following three conditions:*

(i) T consists of a single leaf.
(ii) The root of T does a less-than comparison.
(iii) The root of T does an equality test to k_b.

Proof. The rest of this section gives the proof. It uses the following lemma as a black box.

Lemma 1. *Some optimal tree T satisfies one of the following three conditions:*

(a) T consists of a single leaf.
(b) The root of T does a less-than comparison.
(c) The root of T does an equality test to some key k_b of maximum weight among keys in $\mathcal{S}$.

(Anderson et al. prove essentially the same lemma (their Corollary 3), but for the successful-queries variant. Their proof uses their side-weights technique and extends essentially unchanged to our setting. For completeness, we include the proof in Appendix A.)

The idea of the proof is to infinitesimally perturb the weights to make k_b the uniquely largest-weight key in $\mathcal{S}$. Lemma 1 implies that there is a tree T that is optimal for the perturbed instance and has one of the Properties (a)–(c), which (given the perturbation) implies that T has one of the Properties (i)–(iii). The perturbation is sufficiently small so that any optimal tree (T, in particular) for the perturbed instance must also be optimal for the original instance. (Similar arguments have been used for other problems, such as the Minimum Spanning Tree problem, to extend algorithms for distinct-weights instances to algorithms for general instances; see, e.g., Reference [18, Section 4.5].) Here are the details.

For each integer $i \geq 1$, define a new instance with the same set of keys but with the weight of key k_b increased by $1/i$. That is, let $\beta_b^i = \beta_b + 1/i$ and $\beta_a^i = \beta_a$ for all $a \neq b$. By Lemma 1, there is an optimal tree T^i for the modified instance such that T^i satisfies one of (a)–(c) of Lemma 1 for the modified instance, and therefore satisfies one of (i)–(iii) of the theorem for the original instance. Let tree T be such that $T = T^i$ for infinitely many i. (Such a T must exist, because the possible trees for each modified instance are the same, and there are finitely many of them.) Then T is a possible tree for the original instance and has one of Properties (i)–(iii), as desired. To finish, we show that T is optimal for the original instance.

Let T^* be an optimal tree for the original instance. For all i such that $T = T^i$,

$$\begin{aligned}
\text{cost}(T,\beta) &= \text{cost}(T,\beta^i) - \text{depth}_T(k_b)/i && \text{by definition of } \beta_a^i\text{'s and Equation (1)}\\
&\leq \text{cost}(T,\beta^i) && \\
&= \text{cost}(T^i,\beta^i) && \text{because } T = T^i \text{ for this } i\\
&\leq \text{cost}(T^*,\beta^i) && \text{because } T^i \text{ is optimal for } \beta_a^i\text{'s}\\
&= \text{cost}(T^*,\beta) + \text{depth}_{T^*}(k_b)/i && \text{by definition of } \beta_a^i\text{'s and Equation (1).}
\end{aligned}$$

(In this derivation, arguments β or β^i in the cost function indicate which key weights are used to compute it.) This holds as $i \to \infty$, so $\text{cost}(T,\beta) \leq \text{cost}(T^*,\beta)$, and thus T is also optimal for the original instance. □

Remarks. The correct statement of Theorem 1 is somewhat delicate. Consider the following statement: *If there is an optimal tree whose root does an equality test, then, for any maximum-weight key k in $\mathcal{S}$, there is an optimal tree whose root does an equality test to k.* This seemingly similar statement is false. Figure 3 gives a counterexample.

In that figure, keys 1 and 2 both have maximum weight among keys reaching the right child of the root. Trees (a) and (b) are optimal. In tree (a), the right child of the root does an equality test to key 1, but no optimal subtree for that subproblem (the subproblem with valid set $\mathcal{S} = [1, \infty)$) has

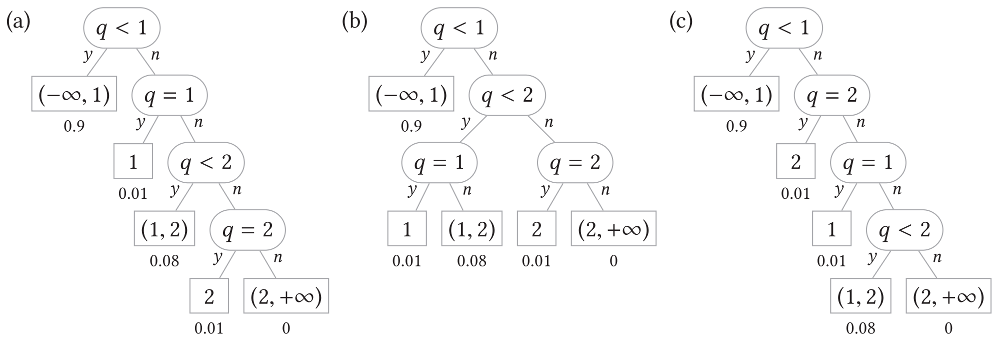

Fig. 3. Trees for a full instance with $\mathcal{K} = \{1, 2\}$, $(\beta_1, \beta_2) = (0.01, 0.01)$ and $(\alpha_0, \alpha_1, \alpha_2) = (0.9, 0.08, 0)$. As discussed in the remarks, the subproblem with valid set $\mathcal{S} = [1, \infty)$ has an optimal solution with an equality test to key 1, but none with an equality test to key 2, even though $\beta_1 = \beta_2$.

an equality test to key 2 at the root—tree (c) has minimum cost among trees with such a subtree. This counter-example also holds in the problem variant where all comparisons in $\{<, \leq, =\}$ are allowed.[3]

The proof of Theorem 1 perturbs the weights infinitesimally, giving k_b the unique largest weight and (via Lemma 1) forcing any root equality test to be to key k_b. It then observes that, because the minimum tree cost is a continuous function of the weights, any tree that is optimal for the (infinitesimally!) perturbed instance must also be optimal for the original instance. This infinitesimal perturbation can make *all* previously optimal trees with equality tests at the root non-optimal. (For example, in Figure 3, after an infinitesimal increase in β_2, only tree (b) would be optimal.)

The RMLK property from the introduction is a corollary of the theorem. To prove it, let T be the optimal tree for $\mathcal{S}$, with root as specified in the theorem. Then (using induction on n) replace each of the two subtrees by an optimal one (for its subproblem) having the RMLK property. The resulting tree T' is also an optimal tree for $\mathcal{S}$, and has the RMLK property.

In fact, there must be an optimal tree with the following stronger property. *At any node N that does an equality test, if there are d largest-weight keys in the search space reaching N, then the top d nodes on the right spine in T_N do equality tests to those d keys in some order (and permuting that order arbitrarily preserves optimality).* This can be shown directly by a careful treatment of ties arising in Anderson et al.'s side-weights argument. See Corollary 2 in Appendix B of [7] for details.

APPENDICES

A PROOF OF LEMMA 1

This Appendix proves Lemma 1. First, we state Anderson et al.'s side-weights lemma [1, Lemma 2], and prove it in our setting. (The proof is identical.) Fix the keys $k_1, k_2, \ldots, k_n$ and associated weights. Let $\mathcal{S}$ be any valid set and consider the problem of finding a minimum-cost tree among trees that handle $\mathcal{S}$. Let T be any optimal tree for this problem.

[3]For the successful-queries variant no such counter-example is possible; that is, it can be shown that if some optimal tree has an equality test at the root, then for *any* maximum-weight key there is an optimal tree that uses this key in the equality-test at the root. A proof can be found in Reference [7]. This stronger property, however, does not seem to have any algorithmic implications.

Define the *side-weight* of any node N in T, denoted $\mathrm{sw}(N)$, as follows. If N is a leaf, then $\mathrm{sw}(N) = 0$. If N does an equality test to a key, then $\mathrm{sw}(N)$ is the weight of that key. Otherwise (N is an inequality-comparison node) $\mathrm{sw}(N) = \min\{w_L, w_R\}$, where L and R are N's left and right children. The utility lemma states that side-weights are monotone along paths from the root:

Lemma 2 ([1, Lemma 2]). *If P is the parent of N in T, then $\mathrm{sw}(P) \geq \mathrm{sw}(N)$.*

Proof. If N is a leaf, then it has side-weight 0 and the property holds, so assume that N is not a leaf. There are four cases according to whether at each of P or N the comparison done is an equality or an inequality. Recall that T is an optimal tree.

Case 1: *Both P and N are inequality comparisons.* Assume without loss of generality that N is the right child of P. Let T_1 be the subtree rooted at the child of P that is not N. Let T_2 and T_3 be the subtrees rooted, respectively, at the left and right children of N. Let μ_i denote the weight of T_i (for $i \in \{1,2,3\}$). Then $\mathrm{sw}(P) = \min\{\mu_1, \mu_2 + \mu_3\}$ and $\mathrm{sw}(N) = \min\{\mu_2, \mu_3\}$. Now do a left rotation at P, so that N takes the place of P, P becomes N's left child, and T_2 becomes the right subtree of P. While T_2 stays at the same depth, T_1 moves down and T_3 moves up, each by one level. The increase in cost is $\mu_1 - \mu_3$. This must be non-negative, as T is optimal, so $\mu_1 \geq \mu_3$. This implies $\min\{\mu_1, \mu_2 + \mu_3\} \geq \min\{\mu_2, \mu_3\}$, that is, $\mathrm{sw}(P) \geq \mathrm{sw}(N)$.

Case 2: *Node P is an inequality comparison and N is an equality comparison.* Let T_1, of weight, say, μ_1, be the subtree rooted at the child of P that is not N. Let μ_2 be the weight of the left subtree, say T_2, of N (consisting of a leaf for the equality-test key of N). Let μ_3 be the weight of the right subtree, say T_3 of N. We have $\mathrm{sw}(P) = \min\{\mu_1, \mu_2 + \mu_3\}$ and $\mathrm{sw}(N) = \mu_2$. Modify T by moving the equality test at N just above P. Tree T_1 moves down one level, T_3 stays at the same level, yet T_2, of weight μ_2, moves up one level. The net increase in cost is $\mu_1 - \mu_2$. Since T is optimal, this is non-negative, so $\mu_1 \geq \mu_2$. Thus, $\min\{\mu_1, \mu_2 + \mu_3\} \geq \mu_2$. That is, $\mathrm{sw}(P) \geq \mathrm{sw}(N)$.

Case 3: *Both P and N are equality comparisons.* Swap the comparisons in P and N. The increase in cost is $\mathrm{sw}(P) - \mathrm{sw}(N)$. Since T is optimal, this is non-negative, so $\mathrm{sw}(P) \geq \mathrm{sw}(N)$.

Case 4: *Node P is an equality comparison and N is an inequality comparison.* Let μ_1 be the weight of P's left-subtree T_1 (consisting of a leaf for the equality-test key of P). Let T_2 and T_3 of weight μ_2 and μ_3, respectively, be the subtrees hanging off N. Move the equality-test node P down just above the appropriate child of N. Then exactly one of T_2 and T_3 moves up while T_1 moves down. The increase in cost is either $\mu_1 - \mu_2$ or $\mu_1 - \mu_3$. Since T is optimal, the increase is non-negative, so $\mu_1 \geq \min\{\mu_2, \mu_3\}$. That is, $\mathrm{sw}(P) \geq \mathrm{sw}(N)$. □

Next is the proof of Lemma 1. The proof is essentially the same as Anderson et al.'s proof of Reference [1, Corollary 3], which is for successful-queries instances but extends directly to our setting.

Lemma 1. *Some optimal tree T satisfies one of the following three conditions:*

(a) T consists of a single leaf.
(b) The root of T does a less-than comparison.
(c) The root of T does an equality test to some key k_b of maximum weight among keys in $\mathcal{S}$.

Proof. Let T be an optimal tree with root N. In the case that N does an inequality comparison, we are done. So assume N does an equality test to some key k_b, necessarily in $\mathcal{S}$. Let k_a be any largest-weight key in $\mathcal{S}$. The parent of the leaf for k_a is some node P in T. First, if P does not do an equality test to k_a, then replace the comparison at P by an equality test to k_a, without otherwise changing T. This preserves correctness and optimality. Now, in the case that P is the root N, we are done. In the remaining case, by Lemma 2, along the path from N to P, the side-weights are non-increasing, so $\beta_b = \mathrm{sw}(N) \geq \mathrm{sw}(P) = \beta_a$, and k_b is a largest-weight key in $\mathcal{S}$. □

B SOURCE CODE (PYTHON)

See Figure 4.

```
#!/usr/bin/env python3.9
from math import inf # infinity
from functools import lru_cache
memoize = lru_cache(maxsize=None)


def opt_cost(K, alpha, beta):
    'Return min cost of any 2-way-comparison tree for (K, alpha, beta).'

    K_by_wt = sorted(range(1, len(K)+1), key=lambda i: beta[i-1])

    @memoize
    def opt(i, j, h):

        K_in_A_ijh = [b for b in K_by_wt[:h] if i <= b < j]
        w_ijh = sum(alpha[i:j] + [beta[k-1] for k in K_in_A_ijh])

        return (
            0 if j == i+1 and i not in K_in_A_ijh else
            opt(i, j, h-1) if h > 0 and K_by_wt[h-1] not in K_in_A_ijh else
            w_ijh + min(
                opt(i, j, h-1) if h > 0 else inf,
                min(opt(i, b, h) + opt(b, j, h) for b in range(i+1, j))
                if i+1 < j else inf
            )
        )

    return opt(0, len(K)+1, len(K))


# Run the algorithm on the instance from Figure 1(b).

K = [1, 3, 5, 7]
beta = [0.08, 0.02, 0.08, 0.33]
alpha = [0.15, 0.21, 0.08, 0.04, 0.01]

assert opt_cost(K, alpha, beta) == 2.8
```

Fig. 4. Python code for the algorithm for the full variant. (Technical note: the memoization decorator uses a dictionary. To achieve a truly faithful implementation, it should be replaced by a three-dimensional array.)

ACKNOWLEDGMENTS

We are very grateful to the anonymous reviewers for insightful comments that helped us improve the presentation.

REFERENCES

[1] R. Anderson, S. Kannan, H. Karloff, and R. E. Ladner. 2002. Thresholds and optimal binary comparison search trees. *J. Algor.* 44, 2 (Aug. 2002), 338–358. https://doi.org/10.1016/S0196-6774(02)00203-1

[2] A. Andersson. 1991. A note on searching in a binary search tree. *Softw.: Pract. Exper.* 21, 10 (1991), 1125–1128. https://doi.org/10.1002/spe.4380211009

[3] W. Bein. 2013. Advanced techniques for dynamic programming. In *Handbook of Combinatorial Optimization*, Panos M. Pardalos, Ding-Zhu Du, and Ronald L. Graham (Eds.). Springer, New York, 41–92. https://doi.org/10.1007/978-1-4419-7997-1_28

[4] C. Chambers and W. Chen. 1999. Efficient multiple and predicated dispatching. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'99)*. ACM, New York, NY, 238–255. https://doi.org/10.1145/320384.320407

[5] M. Chrobak, M. Golin, J. I. Munro, and N. E. Young. 2015. Optimal search trees with two-way comparisons. In *Proceedings of the International Symposium on Algorithms and Computation (ISAAC'15) (Lecture Notes in Computer Science)*, Khaled Elbassioni and Kazuhisa Makino (Eds.), Vol. 9472. Springer, Berlin, 71–82. See Reference [9] for erratum. https://doi.org/10.1007/978-3-662-48971-0_7

[6] M. Chrobak, M. Golin, J. I. Munro, and N. E. Young. 2021. On the cost of unsuccessful searches in search trees with two-way comparisons. *Info. Comput.* (2021). https://doi.org/10.1016/j.ic.2021.104707

[7] M. Chrobak, M. Golin, J. I. Munro, and N. E. Young. 2021. A simple algorithm for optimal search trees with two-way comparisons. Retrieved from https://arxiv2103.01084.

[8] M. Chrobak, M. J. Golin, J. I. Munro, and N. E. Young. 2021. On Huang and Wong's algorithm for generalized binary split trees. Retrieved from https://arxiv1901.03783. To appear in *Acta Informatica*.

[9] M. Chrobak, M. J. Golin, J. I. Munro, and N. E. Young. 2021. Optimal search trees with two-way comparisons. Retrieved from https://arxiv1505.00357. Includes erratum for and pointers to journal versions of other results from Reference [5].

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. 2009. *Introduction to Algorithms* (3rd ed.). The MIT Press, Cambridge, MA.

[11] S. Dasgupta, C. Papadimitriou, and U. Vazirani. 2006. *Algorithms* (1st ed.). McGraw-Hill Education, Boston, MA.

[12] A. Garsia and M. Wachs. 1977. A new algorithm for minimum cost binary trees. *SIAM J. Comput.* 6, 4 (Dec. 1977), 622–642. https://doi.org/10.1137/0206045

[13] E. N. Gilbert and E. F. Moore. 1959. Variable-length binary encodings. *Bell Syst. Tech. J.* 38 (1959), 933–967.

[14] J. H. Hester, D. S. Hirschberg, S. H. Huang, and C. K. Wong. 1986. Faster construction of optimal binary split trees. *J. Algor.* 7, 3 (Sept. 1986), 412–424. https://doi.org/10.1016/0196-6774(86)90031-3

[15] T. C. Hu and A. C. Tucker. 1971. Optimal computer search trees and variable-length alphabetical codes. *SIAM J. Appl. Math.* 21, 4 (1971), 514–532. https://doi.org/10.1137/0121057

[16] T. C. Hu and P. A. Tucker. 1998. Optimal alphabetic trees for binary search. *Inform. Process. Lett.* 67, 3 (Aug. 1998), 137–140. https://doi.org/10.1016/S0020-0190(98)00101-X

[17] S.-H. Huang and C. K. Wong. 1984. Generalized binary split trees. *Acta Inform.* 21, 1 (May 1984), 113–123. https://doi.org/10.1007/BF00289143

[18] J. M. Kleinberg and É. Tardos. 2006. *Algorithm Design*. Addison-Wesley.

[19] D. E. Knuth. 1971. Optimum binary search trees. *Acta Inform.* 1, 1 (Mar. 1971), 14–25. https://doi.org/10.1007/BF00264289

[20] D. E. Knuth. 1998. *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.). Addison-Wesley Publishing Company, Redwood City, CA.

[21] S. V. Nagaraj. 1997. Optimal binary search trees. *Theoret. Comput. Sci.* 188, 1 (Nov. 1997), 1–44. https://doi.org/10.1016/S0304-3975(96)00320-9

[22] D. Spuler. 1994. Optimal search trees using two-way key comparisons. *Acta Inform.* 31, 8 (Aug. 1994), 729–740. https://doi.org/10.1007/BF01178732

[23] D. Spuler. 1994. *Optimal search trees using two-way key comparisons*. Ph.D. Dissertation. James Cook University.

Received July 2019; accepted July 2021