arXiv:1901.03783v3 [cs.DS] 9 Mar 2021

On Huang and Wong's Algorithm for Generalized Binary Split Trees

Marek Chrobak* Mordecai Golin† J. Ian Munro‡ Neal E. Young§

March 10, 2021

Abstract

Huang and Wong [6] proposed a polynomial-time dynamic-programming algorithm for computing optimal generalized binary split trees. We show that their algorithm is incorrect. Thus, it remains open whether such trees can be computed in polynomial time.

Spuler [12, 13] proposed modifying Huang and Wong's algorithm to obtain an algorithm for a different problem: computing optimal two-way-comparison search trees. We show that the dynamic program underlying Spuler's algorithm is not valid, in that it does not satisfy the necessary optimal-substructure property and its proposed recurrence relation is incorrect. It remains unknown whether the algorithm is guaranteed to compute a correct overall solution.

1 Introduction

Given an ordered set K of n keys, a *generalized binary split tree* T is a form of binary search tree where each node N has two associated keys in K: an *equality-test* key and a *split* key [6]. For any query $v \in \mathsf{K}$, a *search* for v in T starts at the root. If v equals the root's equality-test key, then the search halts. Otherwise, the search recurses in the left or right subtree, depending on whether or not v is less than the root's split key. A correct tree T must have n nodes, and the search for each query $v \in \mathsf{K}$ must halt at the node whose equality-test key is v. (There must be exactly one such node for each $v \in \mathsf{K}$.) Given also a probability distribution p on K, the *cost* of a tree T is the expected number of nodes visited when searching in T for a random query v drawn from p. The goal, given K and p, is to compute a tree T of minimum cost (thus minimizing, over any tree T of this form, the expected number of two-way comparisons made when searching in T). We denote this problem GBSPLIT. See Figure 1 for an example.

Note that this definition is for the so-called *successful-queries* variant, in which the tree only supports queries for values in K. In contrast, the *general* variant concerns trees that also support "unsuccessful" (non-key) queries. The search for any non-key query $v \notin \mathsf{K}$ must identify its neighboring key(s).

Huang and Wong [6] proposed a polynomial-time algorithm for GBSPLIT. We show (in Theorem 1, Section 2) that their algorithm and claimed proof of correctness are wrong. The reason is that their dynamic

*University of California at Riverside. Research supported by NSF grants CCF-1217314 and CCF-1536026.

†Hong Kong University of Science and Technology. Research funded by HKUST/RGC grant FSGRF14EG28 and RGC CERG Grant 16208415.

‡University of Waterloo. Research funded by NSERC and the Canada Research Chairs Programme.

§University of California at Riverside. Research supported by NSF grant IIS-1619463.

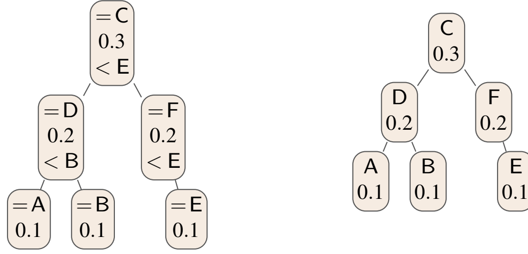

Figure 1: The picture on the left shows an example of a generalized binary split tree for key interval $\{\mathsf{A},\mathsf{B},\mathsf{C},\mathsf{D},\mathsf{E},\mathsf{F}\}$. Each node is labeled with its equality key and its probability, as well as the node's split key (except that split keys are omitted at leaves, where they are irrelevant). The total cost of this tree is $0.3\cdot 1+2\cdot(0.2\cdot 2)+3\cdot(0.1\cdot 3)=2$. In all figures in the paper we use a more compact representation, shown on the right, where split keys are omitted. (Each node's split key can be any key that separates the equality keys in the left subtree from those in the right subtree.)

program does not satisfy the claimed optimal-substructure property. Consequently, as far as we know, it is not known whether GBSPLIT has a polynomial-time algorithm.

A closely related problem is to find an optimal *two-way comparison* search tree, in which each node is associated with just one key and one binary comparison operator — equality or less-than. We use 2WCST to denote this problem. (See Figure 5 for an example.) Spuler [12, 13] proposed several 2WCST algorithms. He described two of his proposed 2WCST algorithms (for the successful-queries and general variants, respectively) as "straightforward" modifications of Huang and Wong's GBSPLIT algorithm, but he gave no formal proof of correctness, explaining only that correctness follows from the dynamic-programming formulation, in particular from the underlying recurrence relation.

We show (Theorem 3, Section 3) that this recurrence relation is wrong, and his algorithm computes incorrect solutions to some subproblems in the dynamic program. Here also the dynamic program does not satisfy the assumed optimal-substructure property. This counterexample is only for a subproblem, not a full instance, so the overall correctness of his proposed algorithm remains open.

Historical context. The study of optimal binary search trees began with *three-way comparison* search trees. These have only one key associated with each node, and comparing the given query to that key has *three* possible outcomes — less than, equal to, or greater than. Knuth's classical dynamic-programming algorithm computes a minimum-cost tree of this kind (supporting both successful and unsuccessful queries) in time $O(n^2)$ [8].

Following Knuth's suggestion [9, §6.2.2 ex. 33], various authors began exploring trees based on two-way (binary) comparisons. Sheil [11] introduced *median split trees* — generalized binary split trees where the split key at each node N must be a median key among the set K_N of keys whose search visits node N, and the equality-test key must be a most likely key among K_N. He gave an $O(n\log n)$-time algorithm to compute a median split tree (for the successful-queries variant). Other authors [7, 10, 5] then introduced *binary split trees* — generalized binary split trees with the added restriction that the equality-test key at each node must be a most likely key among keys reaching the node. These trees can be thought of as a relaxation of median split trees, without the restriction that the split key has to be a median key. Their algorithms compute minimum-cost binary split trees in $O(n^5)$ time for both the successful-queries and general variants. (See also the note at the end of this paper.) Huang and Wong [6] then introduced GBSPLIT (generalized binary split trees) as defined above, and proposed an $O(n^5)$-time algorithm for the problem, the one we

show here to be incorrect.

Subsequently, the algorithm was extended by Chen and Liu to MULTIWAY GBSPLIT, a variant of GBSPLIT that requires multiple split keys per node [2]. Chen and Liu's algorithm and proof of correctness are directly patterned on Huang and Wong's. Their proof is invalid (and we believe their algorithm to be incorrect) for the same reason that Huang and Wong's proof and algorithm fail. (See the remark at the end of Section 2.)

As mentioned above, Spuler [12, 13] proposed several 2WCST algorithms without proof of correctness. Anderson et al. [1] gave the first proof that 2WCST is in polynomial time. Their algorithm runs in time $O(n^4)$ and is restricted to the successful-queries variant. Chrobak et al. [3] gave a somewhat simpler $O(n^4)$-time algorithm for the general variant.

Beyond pointing out errors in the literature on binary search trees, we hope that the constructions underlying our counter-examples will contribute to a better understanding of the difficulties involved in designing algorithms for GBSPLIT and 2WCST, leading to better algorithms or even new hardness results.

2 Huang and Wong's GBSPLIT algorithm is incorrect

This section gives our first main result: a proof that Huang and Wong's proposed GBSPLIT algorithm [6] has a fundamental flaw.

Theorem 1. *Huang and Wong's* GBSPLIT *algorithm [6] is incorrect. There is a* GBSPLIT *instance* (K, p) *for which it returns a non-optimal tree.*

We summarize their algorithm and analysis, give the intuition behind the failure, then prove the theorem. The basic intuition is that, for the dynamic program that Huang and Wong define, the optimal-substructure property fails. The proof gives a specific counter-example and verifies it. The counter-example can also be verified computationally by running the Python code for Huang and Wong's algorithm in Appendix A.

Fix any GBSPLIT instance (K, p). Assume without loss of generality that the keys are $\mathsf{K} = \{1, 2, \ldots, n\}$. Regarding the probability vector p, for convenience, throughout the paper we drop the constraint that the probabilities must sum to 1, and we use "probabilities" and "weights" synonymously, allowing their values to be arbitrary non-negative reals. (To represent probabilities, these values can be appropriately normalized.)

During a search, the outcome of each less-than comparison narrows the current search interval, while the outcome of each (failed) equality test removes one key within the interval from consideration. Thus, at each node in any search tree, the set of keys reaching the node consists of some interval of keys, minus some so-called *holes* — keys removed from consideration by previous equality tests. Next we formally define an exponentially large (!) class of subproblems that arise in this way, along with a natural recurrence relation for their cost. We then discuss how Huang and Wong attempt to reduce the number of subproblems to $O(n^3)$.

Abusing notation, a *query interval* $I = [i, j]$ is the set of contiguous keys $\{i, i+1, \ldots, j\}$. Given any query interval I and any subset $H \subseteq I$ of "hole" keys, consider the subproblem (I, H) formed by the subset of keys $I \setminus H$, with the weight distribution obtained from p by restricting to $I \setminus H$. Let $\mathsf{opt}(I, H)$ denote the minimum cost of any generalized binary split tree for this subproblem. Let $p(I \setminus H) = \sum_{k \in I \setminus H} p_k$ denote the total weight of its keys.

If $H = I$ then the subproblem can be handled by an "empty" tree, so $\mathsf{opt}(I, H) = 0$. Otherwise, letting $I = [i, j]$, the definition of generalized binary split trees gives the recurrence

$$\mathsf{opt}(I, H) \; = \; p(I \setminus H) + \min_{s \in I;\, e \in I \setminus H} \Big(\mathsf{opt}([i, s-1], H_e \cap [i, s-1]) + \mathsf{opt}([s, j], H_e \cap [s, j]) \Big)$$

where $H_e = H \cup \{e\}$. (Here $s \in I$ ranges over the possible split keys; $e \in I \setminus H$ ranges over the possible equality keys.)

The goal is to compute $\mathsf{opt}(\mathsf{K},\emptyset)$. The recurrence above allows arbitrary equality keys e, so it gives rise to exponentially many hole sets H, resulting in a dynamic program with exponentially many subproblems. Huang and Wong propose a dynamic program with $O(n^3)$ subproblems (I,h), one for each interval I and integer $h \leq |I|$. Specifically, they define

$$\mathsf{opt}^*(I,h) \;=\; \min\{\,\mathsf{opt}(I,H) : H \subseteq I,\, |H| = h\,\},$$

which is the minimum cost of any tree for interval I minus *any* hole set of size h. Each such tree will have $|I|-h$ nodes. (Their paper uses "$p[i-1,j,h]$" to denote $\mathsf{opt}^*([i,j],h)$.) We refer to any such subproblem (I,h) as an *HW-subproblem*.

They develop a recurrence for $\mathsf{opt}^*(I,h)$ as follows. For any node N in an optimal tree, define N's *interval* I_N and *hole set* H_N in the natural way so that interval I_N contains those key values that, if searched for in T with the equality tests ignored, would reach N, and $H_N \subseteq I_N$ contains those keys *in interval* I_N that are equality keys at ancestors of N. Hence, the set of keys reaching N is $I_N \setminus H_N$, and the subtree rooted at N is a solution for the subproblem (I_N,H_N), as well as the HW-subproblem $(I_N,|H_N|)$, that we refer to as *the HW-subproblem arising at* N. Huang and Wong's Lemma 1 states:

Lemma 1 from [6] (ambiguous) *"Subtrees of an optimal generalized binary search tree are optimal generalized binary search trees."*

This statement is ambiguous in that it doesn't specify *for which subproblem* the subtree is optimal. Consider any subtree T' of an optimal tree T^*. Let T' have root N, interval I_N and hole set H_N. The first interpretation of their Lemma 1 is that T' must be an optimal solution for (I_N,H_N). With this interpretation (following the first recurrence above), the lemma is indeed true. But another interpretation is that T' must be an optimal solution for the HW-subproblem $(I_N,|H_N|)$ arising at N. This interpretation is not the same — the HW-subproblem specifies only the *number* of holes, and choosing different holes can give a cheaper tree, so it can be that $\mathsf{opt}^*(I_N,|H_N|) < \mathsf{opt}(I_N,H_N)$. As we shall see below, it is the second interpretation that underlies the recurrence relation that Huang and Wong propose, but, with that interpretation, as our Theorem 2 will show, the above lemma is false because the HW-subproblems do not have optimal substructure.

The ambiguity in Lemma 1 appears to be their first misstep. They follow it with the following (correct) observation:

Lemma 2 from [6] (correct) *Let N be the root of a subtree T' with interval I in an optimal generalized binary split tree T^*. The equality-test key e_N of N must be the least frequent key among those in N's interval I_N that do not occur (as an equality-test key) in the left and right subtrees of N.*[1]

Proof. The proof is a simple exchange argument. Suppose for contradiction that e_N is more likely than some key k in I_N and k does not occur as an equality-test key in the left and right subtrees of N. Then k is a hole at N, so it must be the equality-test key $k = e_{N'}$ of some ancestor N' of N. A contradiction is obtained by observing that exchanging e_N and $e_{N'}$ gives a correct tree cheaper than T^*. $\square$

Huang and Wong's Lemma 2 above (with the second, incorrect interpretation of their Lemma 1) suggests the following idea. To find a hopefully optimal tree $\tau(I,h)$ for the HW-subproblem (I,h), consider each possible root split key and each possible split of the h hole slots. For each, first find optimal left and right subtrees for their respective subproblems, and then take the equality key at the root of $\tau(I,h)$ to be the least-likely key in I that is not an equality test in either subtree. Among trees obtained in this way, take $\tau(I,h)$

[1] To avoid confusion, note that the lemma does not preclude a descendant D of N from having an equality-test key e_D that is more likely than e_N, because e_N might not be in D's interval. So it does not imply that the equality-test key e_N at N is as likely as all equality-test keys in the subtree rooted at N. For example, see keys A2 and D1 in Figure 2 (a).

to be one of minimum cost. Following this idea, their algorithm solves any given HW-subproblem (I,h), where $I=[i,j]$, as follows:

---

1. For each triple (s,h_1,h_2) where $s\in I$ is a valid split key, and h_1 and h_2 are non-negative integers such that $h_1+h_2=h+1$, $h_1\le s-i$, and $h_2\le j-s+1$, construct one possible *candidate tree* $T(s,h_1,h_2)$ as follows:

   1.1. Give $T(s,h_1,h_2)$ left and right subtrees $\tau([i,s-1],h_1)$ and $\tau([s,j],h_2)$.

   1.2. Give the root of $T(s,h_1,h_2)$ split key s and equality-test key e, *where e is a least-likely key in I that is not an equality-test key in either subtree.*

2. Among trees $T(s,h_1,h_2)$ so constructed, take $\tau(I,h)$ to be one of minimum cost.

The algorithm is not hard to implement. Appendix A gives Python code for it (40 lines).

Note that, by their Lemma 2, the choice for e in Line 1.2. would be correct *if* the second interpretation of their Lemma 1 was correct. We surmise that this line of thinking led Huang and Wong to their algorithm.

To justify the algorithm, Huang and Wong proceed as follows. Fix any possible execution of the algorithm. For any HW-subproblem (I,h) that it solves, let $\mathsf{opt}^*(I,h)$ denote the minimum cost of any tree for the subproblem. Recall that $\tau(I,h)$ denotes the algorithm's solution (tree) for the subproblem, presumably of cost $\mathsf{opt}^*(I,h)$. Let $w(I,h)$ denote the total weight of the equality keys in $\tau(I,h)$. Huang and Wong first state a correct base case:

Lemma 3 from [6] (correct) $\mathsf{opt}^*(\emptyset,0)=w(\emptyset,0)=0$.

But their Lemma 4 then claims that, for any non-empty interval $I=[i,j]$ and any number of holes $h\le|I|$, the following recurrence relation holds:

Lemma 4 from [6] (incorrect)

$$\mathsf{opt}^*(I,h) = \min_{s,h_1} \big(w(I,h)+\mathsf{opt}^*([i,s-1],h_1)+\mathsf{opt}^*([s,j],h-h_1+1) \big)$$

where the minimum is over all all legal combinations of s and h_1.

Remark about ties. During the execution of the algorithm, in Lines 1.2 and 2, ties may arise in choosing a minimizer. Different choices will lead to different subtrees $\tau(I,h)$. In this way, in principle, the value of $w(I,h)$ can depend on how ties are broken. Huang and Wong do not explicitly discuss tie-breaking, but, for their Lemma 4 to be correct, all choices must lead to the same value for $w(I,h)$.

Next we show, as discussed previously, that the optimal-substructure property assumed by Lemma 4 for the HW-subproblems simply does not hold. It will follow that Lemma 4 and the algorithm fail on (K,p), in any execution of the algorithm on that input, regardless of how ties are broken.

Theorem 2. *There exists a* GBSPLIT *instance* (K,p) *with the following property. In every optimal tree* T^* *for* (K,p)*, there is at least one node N such that the subtree* T^*_N *rooted at N in* T^* *is not optimal for the HW-subproblem* $(I_N,|H_N|)$ *arising at N. (The tree* T^*_N *has cost strictly larger than* $\mathsf{opt}^*(I_N,|H_N|)$*.)*

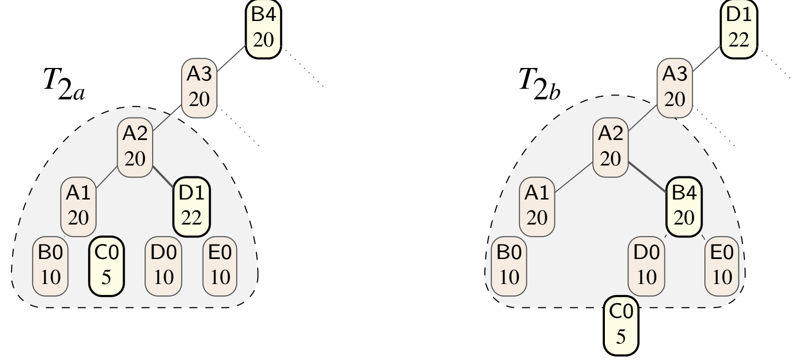

Figure 2: Subtrees T_{2_a} and T_{2_b} for 9-key interval I_9 with $h = 2$. T_{2_a} is missing the two keys $\mathsf{A3}, \mathsf{B4}$; T_{2_b} is missing $\mathsf{A3}, \mathsf{D1}$. Each node shows its equality key and the frequency of that key; split keys are not shown. (For each node, take the split key to be any key that separates the keys in the left and right subtrees.) The costs of T_{2_a} and T_{2_b} are 209 and 210, respectively, but in T_{2_a}, the total weight of the keys is larger by 2.

Proof. Before we describe (K, p), we first describe an HW-subproblem for which using a minimum-cost tree T' can be a bad choice globally. The HW-subproblem is $(I_9, 2)$, with $h = 2$ holes and interval I_9 consisting of nine keys $I_9 = \{\mathsf{A1, A2, A3, B0, B4, C0, D0, D1, E0}\}$, ordered lexicographically, with weights as follows:

key	A1	A2	A3	B0	B4	C0	D0	D1	E0
weight	20	20	20	10	20	5	10	22	10

Figure 2 shows two possible subtrees T_{2_a} and T_{2_b} for $(I_9, 2)$, each with seven nodes. By calculation, subtree T_{2_b} costs 1 more than subtree T_{2_a} for $(I_9, 2)$. (Indeed, key C0 contributes 5 units more to T_{2_b} than to T_{2_a}, while key B4 contributes 4 units less to T_{2_b} than key D1 contributes to T_{2_a}.)

Although T_{2_b} costs 1 more than T_{2_a}, choosing subtree T_{2_b} instead of T_{2_a} can decrease the cost of the overall tree! To see why, suppose that T_{2_a} occurs as a subtree of some tree T^*, in which T_{2_a} has parent A3 and grandparent B4 as shown in the figure. (See also Figure 3.) Consider replacing T_{2_a} and its two hole keys A3 and B4 by T_{2_b} and its two hole keys A3 and D1. This replacement decreases the cost of the tree by 1 unit, because the contribution of C0 increases by 5, swapping B4 and D1 decreases the cost by 6, and the contributions of other nodes do not change. But a different calculation gives better intuition why Huang and Wong's algorithm fails: the cost of T_{2_b} is 1 larger than T_{2_a}'s and the contribution of the root D1 is 2 more than B4's; however, this increase of 3 is more than compensated by decreasing the total weight of keys in T_{2_b} by 2 units, as this weight is counted with factor 2 towards the cost.

Next we use this HW-subproblem to obtain the complete instance (K, p) for Theorem 2. The instance has a 31-key interval I_{31}, which extends the previously considered interval I_9 by appending two "neutral" subintervals, with 7 and 15 keys. Figure 3 shows two trees T_{3_a} and T_{3_b} for (K, p). As shown there, the new keys are given weights so that each of the two added subintervals (without any holes) has a self-contained, optimal balanced subtree. To finish proving Theorem 2, we prove that (K, p) has the necessary properties:

Lemma 5. *Let T^* be any optimal tree for this* GBSPLIT *instance (K, p). At some node N of T^* the HW-subproblem $(I_9, 2)$ arises, but the subtree T_N rooted at N has cost at least 210 for $(I_9, 2)$, while* $\mathsf{opt}^*(I_9, 2) \leq 209$.

To bound tree costs, define a *key placement* (for a tree T) to be an assignment of the equality-test keys in T to distinct nodes in the infinite rooted binary tree T_∞. Define the *cost* of the placement to be the average

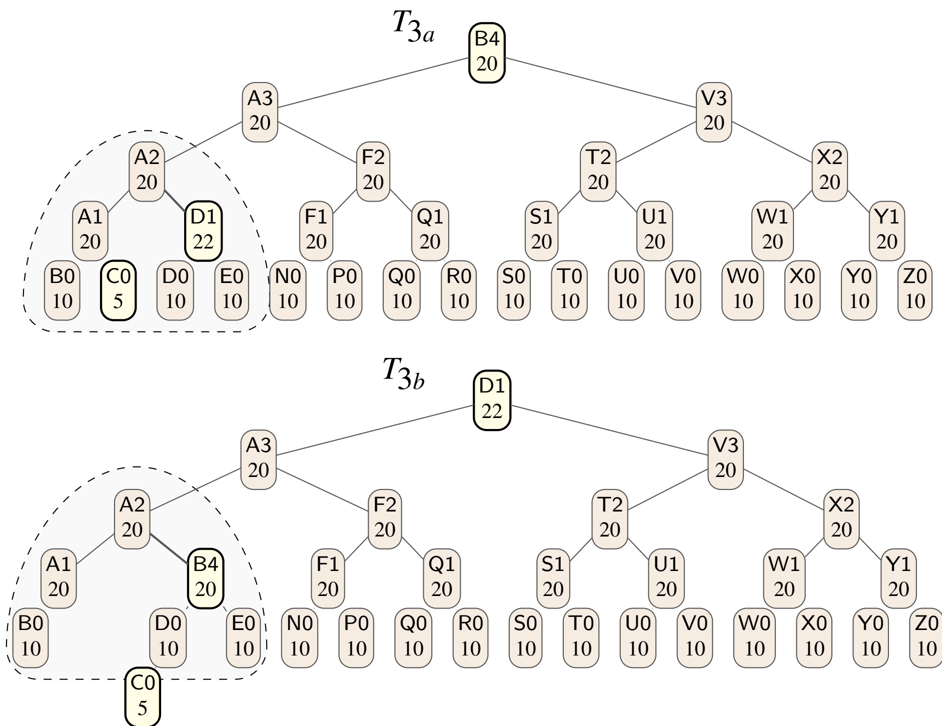

Figure 3: Trees T_{3_a} and T_{3_b} for an instance of GBSPLIT with 31-key interval I_{31}. Key order is lexicographic: $\mathsf{A0} < \mathsf{A1} < \mathsf{A2} < \mathsf{A3} < \mathsf{B0} < \cdots$. As in Figure 2, split keys are not shown. Huang and Wong's algorithm gives a tree of cost 1763, such as T_{3_a}, but tree T_{3_b} costs 1762.

weighted depth of the placed keys, weighted according to the key weight-vector p. Each correct GBSPLIT tree T yields a placement of equal cost by placing each equality-test key in the same place in T_∞ that it occupies in T. The converse does not hold, partly because placements can ignore the ordering of keys.

By an exchange argument, a placement has minimum cost if and only if it puts the weight-22 key D1 at depth 0, the fourteen weight-20 keys at depths 1–3, and the sixteen remaining (weight-10 and weight-5) keys at depth 4. By calculation, such a placement costs 1757. No placement costs less, so no tree costs less. Tree T_{3_b} almost achieves a minimum-cost placement — it fails only in that it places the weight-5 key at depth 5, so costs 1762, just 5 units more than the minimum placement cost.

Claim 6. *T^* has the following structure:*

(i) It places the fifteen keys of weight 20 or more at depths 0–3.

(ii) It places the fifteen weight-10 keys at depth 4.

Next we prove the claim. Since T^* is optimal it costs at most 1762 (the cost of T_{3_b}), so its placement also costs at most 1762. Suppose for contradiction that (i) doesn't hold. Then T^* places a key k of weight 20 or more at depth at least 4. Also, in depths 1–3, it either places at least one key k' of weight 10, or places fewer than fifteen keys. In either case, by exchanging k and k', or just re-placing k in depth 1–3, we can obtain a key placement that costs at least 10 units less than 1762. But this is impossible, as the minimum placement cost is 1757. So (i) holds. Now suppose for contradiction that (ii) doesn't hold. Then there is a weight-10 key k' at depth 5 or more, and at most fifteen keys at depth 4, so k' can be re-placed in depth 4, yielding a key placement that costs 10 less, which is impossible. This proves the claim.

Key placements ignore the ordering of keys. The following *order property* captures the restrictions on key placements due to the ordering.

> *Let T be any correct* GBSPLIT *tree. Let P and P' be nodes in T with equality-test keys k and k'. Let Q be the least-common ancestor of P and P'. If P is in Q's left subtree, and P' is in Q's right subtree, then $k < k'$.*

The property holds simply because k and k' are separated by M's split key.

Fix any optimal tree T^* for (K, p). Claim 6 imposes stringent constraints on the depth of all keys in T^*, except for the weight-5 key C0. There are two cases:

Case 1: T^* places C0 *at depth 4.* With Claim 6, this implies that T^* is a complete balanced binary tree of depth 4 (like T_{3_a}), whose sixteen depth-4 nodes hold the fifteen weight-10 keys and C0. By the order property, these depth-4 keys are ordered left to right, just as they are in T_{3_a}, with the left-most four nodes at depth 4 having keys B0, C0, D0, and E0.

The left spine has only five nodes. By the order property, all five keys less than C0 cannot be elsewhere than on the spine. So D1 *is not on the left spine.*

Let M be the parent of sibling leaves D0 and E0. Since D0 $<$ D1 $<$ E0, by the order property, D1 must lie on the path from M to the root. Since D1 is not on the left spine, and M is the only node on this path that is not on the left spine, D1 must be M. So D1 has depth 3 in T^*. Now exchanging D1 with the root key gives a placement that costs at least 6 less, that is, at most $1762 - 6 < 1757$, which is impossible as the minimum placement cost is 1757. So Case 1 cannot happen.

Case 2: T^* places C0 *at depth 5.* Let $L_0, L_1, \ldots, L_\ell$ be the left spine of T^*, starting at the root. Take T' to be the subtree of T^* rooted at L_2. By Claim 6, T^* has fifteen depth-4 nodes, holding the fifteen weight-10

keys. By the order property, these depth-4 keys are ordered left to right within their level and at most twelve of them are not in T'. This implies that the weight-10 keys B0, D0 and E0 must be in T'.

The next larger weight-10 key, N0, cannot be in T'. Indeed, if it were, then by the order property, all keys less than or equal to N0 would be in $T' \cup \{L_1, L_0\}$. But there are twelve keys less than or equal to N0 and at most eight keys in T'.

We now focus on the cost of T'. By the previous two paragraphs, T' has exactly three keys at depth 2, namely B0, D0, and E0. By the order property and the assumption for Case 2, C0 must be (the only key) at depth 3 in T' (as the child of either B0 or D0). By Lemma 8, the three keys at depths 0 and 1 in T' have weight 20 or 22. By calculation, *the cost of T' is therefore at least 210 (see Figure 2).*

Since E0 is in T', by the order property, all eight keys less than E0 are in $T' \cup \{L_0, L_1\}$. That is, $T' \cup \{L_0, L_1\}$ contains at least the 9 keys in I_9. But (as observed above) T' has seven nodes. So $T' \cup \{L_0, L_1\}$ contains exactly the 9 keys in I_9, and the HW-subproblem solved by T' must be $(I_9, 2)$. As observed above, T' costs at least 210. But tree T_{2a} (Figure 2) of cost 209 also solves $(I_9, 2)$, so $\mathsf{opt}^*(I_9, 2) \leq 209$.

This proves Lemma 5 and Theorem 2. □

We prove one final utility lemma before we prove Theorem 1. Consider any execution of Huang and Wong's algorithm on the input (K, p) defined in the proof of Theorem 2. Let $T = \tau(I_{31}, 0)$ be the algorithm's solution.

Lemma 7. *If T contains a node N whose HW-subproblem is $(I_9, 2)$, then the subtree $\tau(I_9, 2)$ rooted at N costs at most 209 for $(I_9, 2)$.*

Proof. Abusing notation, for $1 \leq i < j \leq 9$, let $[i, j]$ denote the ith through jth keys in interval I_9, as shown in Figure 3. (See also Figure 2 for intuition.) Consider the following table:

I, h	s, h_1, h_2	left	right	cost for $\tau(I, h)$	holes
$[1,5], 4$				10	A1, A2, A3, B4
$[6,6], 0$	base cases			5	none
$[7,8], 1$				10	D1
$[9,9], 0$				10	none
$[1,6], 3$	C0, 4, 0	$[1,5], 4$	$[6,6], 0$	$10+5+35 = 50$	three of A1, A2, A3, B4
$[7,9], 0$	E0, 1, 0	$[7,8], 1$	$[9,9], 0$	$10+10+42 = 62$	none
$[1,9], 2$	D0, 3, 0	$[1,6], 3$	$[7,9], 0$	$\leq 50+62+97 = 209$	two of A1, A2, A3, B4

Each row of the table is for one HW-subproblem (I, h) (shown in the leftmost column), and demonstrates that the cost of the tree $\tau(I, h)$ computed by the algorithm for that subproblem is as shown in the fifth column ("cost for subproblem"). The last column lists the keys that are holes in $\tau(I, h)$. The first four rows are base cases (key sets of size one), and hold by inspection. For each subsequent row, the second column gives one of the triples (s, h_1, h_2) considered by the algorithm for the given HW-subproblem (I, h), where s is the split key, and h_1 and h_2 are the number of holes allocated to the left and right subtrees. Columns "left" and "right" show the left and right HW-subproblems that follow from that choice of (s, h_1, h_2), and "cost for $\tau(I, h)$" gives the corresponding cost resulting from that choice. Likewise the final column "holes" describes the possible hole sets (in order to achieve the given cost, depending on how ties are broken). Each row can be verified by inspection, assuming inductively that the previous rows are correct. For the first six HW-subproblems (each with one or three keys), by inspection, the costs shown are optimal. For the seventh subproblem, the cost of 209 is an upper bound (in fact it is optimal, but we don't need that here). □

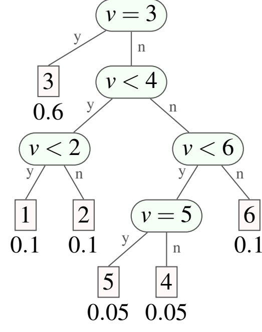

Figure 4: A two-way-comparison search tree with keys $\mathsf{K}=\{1,2,3,4,5,6\}$. Below each leaf is its weight. The cost of this tree is $0.6\cdot 1+0.1\cdot 3+0.1\cdot 3+0.05\cdot 4+0.05\cdot 4+0.1\cdot 3=1.9$.

By Lemmas 5 and 7, the tree T computed by Huang and Wang's algorithm for (K,p) cannot be optimal: Lemmas 5 states that *all* optimal trees for (K,p) contain a node with a certain property, while Lemma 7 states that T does not contain such a node. This proves Theorem 1.

Note that executing their algorithm on (K,p), via the Python code in the appendix, returns a tree of cost 1763 (that does have the HW-subproblem $(I_9,2)$), while T_{3_b} costs 1762.

Remark on Chen and Liu's algorithm for MULTIWAY GBSPLIT **[2].** Chen and Liu's algorithm [2] and analysis are patterned directly on Huang and Wong's, and the proofs they present also conflate (their equivalents of) $\mathtt{opt}^*(I,h)$ and $\mathtt{opt}(I,H)$, leading to the same problems with optimal substructure. For example, Property 1 of [2] states *"Any subtree of an optimal $(m+1)$-way generalized split tree is optimal."* They do not define "optimal", so their Property 1 has the same problem as Huang and Wong's Lemma 1: it is true if "optimal" means "with respect to their equivalent of $\mathtt{opt}(I,H)$", but does not necessarily hold if "optimal" means "with respect to their equivalent of $\mathtt{opt}^*(I,h)$". Lemmas 2, 3 and 4 of [2], which state the recurrence relations for their dynamic program, are direct generalizations of Huang and Wong's Lemma 4. Their recurrence chooses equality keys by first finding optimal subtrees for the children, then taking the equality keys to be the least-likely keys that are not equality keys in the children's subtrees. As pointed out in the proof of Theorem 1, correctness of this approach requires the optimal-substructure property to hold with respect to $\mathtt{opt}^*(I,h)$. But it does not. For these reasons, their proof of correctness is not valid. We believe that their algorithm for MULTIWAY GBSPLIT is also incorrect, but describing their algorithm and analysis in detail, and giving a complete counter-example, are out of the scope of this paper.

3 A 2WCST algorithm by Spuler fails on some subproblems

This section concerns 2WCST, the problem of computing an optimal *two-way comparison* search tree, given a set K of n keys and their weight distribution p. Such a tree T is a rooted binary tree, where each non-leaf node N has two children, as well as a key $k_N\in\mathsf{K}$ and a binary comparison operator (equality or less-than). Denote such a node by $\langle v=k_N\rangle$ or $\langle v<k_N\rangle$, depending on which comparison operator is used. The tree T has n leaves, each labeled with a unique key in K.

The search for a query v in T starts at the root. If the root is a leaf, the search halts. Otherwise, it compares v to the root's key using the root's comparison operator, then recurses left if the comparison succeeds, and right otherwise. For the tree to be correct, the search for any query $v\in\mathsf{K}$ must end at the leaf

that is labeled with v. (For simplicity, we discuss here only the successful-queries variant, in which only queries in K are allowed.) Given an instance (K, p), the problem is to find a tree that minimizes the weighted average depth of the leaves (in the case that p is a probability distribution, this is the expected number of comparisons in a search for a query v drawn randomly according to p). Figure 4 shows an example.

Spuler's thesis proposed various algorithms for 2WCST and for GBSPLIT, for both the successful-queries variant and the general variant [13].[2] Here we discuss the (successful-queries) 2WCST algorithm that Spuler presented as a modification of Huang and Wong's GBSPLIT algorithm in Section 6.4.1 of his thesis [13, Section 6.4.1]. That section starts with the following remark:

> *"The changes to the optimal generalized binary split tree algorithm of Huang and Wong [6] to produce optimal generalized two-way comparison trees are quite straight forward."*

("Generalized two-way comparison trees" in the thesis are two-way comparison search trees as defined herein.) The remainder of his Section 6.4.1 sketches the code for the algorithm. His Appendix A.4.1. gives complete code. Spuler does not explicitly define the dynamic program or recurrence that he has in mind, however, it is implicitly defined by his algorithm as described below. In addition to lacking proofs of correctness, these algorithms have not appeared in any peer-reviewed publication, although Spuler did refer to them in his journal paper [12], and they have been cited in the literature as the first polynomial-time algorithms for 2WCST [1].

Following Huang and Wong, Spuler's algorithms are based on a dynamic program where each subproblem is specified by an interval of keys and a number of holes, and solves each subproblem using a recurrence relation. In the remainder of the section, we prove that the dynamic program is flawed:

Theorem 3. *There is an instance (K, p) of 2WCST for which the dynamic program used by Spuler's 2WCST algorithm [13, Section 6.4.1] has the following flaws: for some subproblems, the recurrence relation is incorrect and the algorithm computes non-optimal solutions.*

Note that Theorem 3 does not imply that the algorithm is incorrect, in the sense that it gives an incorrect solution to some full instance (where the number h of holes is 0).

Following Huang and Wong, the dynamic program implicit in Spuler's algorithm has a subproblem (I, h) for each query interval I and number of holes h. In what follows we call any such subproblem (I, h) an *S-subproblem*. The definition of a correct tree for an S-subproblem is a natural extension of the definition for full instances: a correct tree for (I, h) must have exactly $|I| - h$ leaves, each labeled with a unique key from I; however, all keys in I can be used as inequality-comparison keys. We use $\mathsf{opt}^*(I, h)$ to denote the minimum cost of any tree for S-subproblem (I, h). The underlying flaw is the same as in Huang and Wong's dynamic program — S-subproblems do not have optimal substructure.

Given any S-subproblem (I, h), where $I = [i, j]$ and the subproblem size $|I| - h$ is more than one, Spuler's algorithm computes a tree $\tau(I, h)$ for S-subproblem (I, h) by combining trees $\tau(I', h')$ that it has computed for smaller S-subproblems, as follows:

[2] We remark that Spuler [13, Section 4.8] pointed out, and claimed to fix, several flaws in the *pseudo-code* that Huang and Wong gave for their GBSPLIT algorithm. Those flaws are relatively minor and do not include the deeper errors discussed in Section 2.

1. Construct one *candidate tree* $T_=$ with an equality test at the root, as follows:

   1.1. Let e be a least-likely key in I that is not a leaf in $\tau(I, h+1)$.

   1.2. Make a candidate tree $T_=$ with root $\langle v = e \rangle$ and right subtree $\tau(I, h+1)$.

2. For all $s \in I$ and (h_1, h_2) s.t. $h_1 + h_2 = h$, $s - i - h_1 \geq 1$ and $j - s + 1 - h_2 \geq 1$, do:

   2.1. Give candidate tree $T_{<s}$ root $\langle v < s \rangle$ and subtrees $\tau([i, s-1], h_1)$, $\tau([s, j], h_2)$.

3. Among the candidate trees so constructed, let $\tau(I, h)$ be one of minimum cost.

Remarks. The algorithm is not hard to implement. Appendix B gives Python code (42 lines). As noted earlier, Spuler does not explicitly define his dynamic program or recurrence relation for $\mathsf{opt}^*(I, h)$, however, it is implicitly defined by his algorithm and his assumption that each tree $\tau(I, h)$ is optimal for S-subproblem (I, h) (so has cost $\mathsf{opt}^*(I, h)$).

Although ties may arise in choosing the minimizers in Lines 1.1 and 3, Spuler does not discuss ties. We'll show that his recurrence relation is incorrect no matter how ties are broken.

Given an S-subproblem (I, h), Spuler's algorithm constructs its tree $\tau(I, h)$ out of trees $\tau(I', h')$ that it built for smaller S-subproblems. This only works if S-subproblems have optimal substructure. To complete the proof of Theorem 3, we show that they do not:

Theorem 4. *There exists a* 2WCST *S-subproblem* (I, h) *with the following property. In every optimal tree* T^* *for* (I, h)*, there is at least one node* N *such that, for the S-subproblem* (I_N, h') *arising at* N*, the subtree* T^*_N *rooted at* N *in* T^* *does* not *have minimum cost,* $\mathsf{opt}^*(I_N, h')$*, for that S-subproblem.*[3]

Proof. Before we describe the full S-subproblem (I, h), we describe one smaller S-subproblem (I', h') for which using a minimum-cost tree T' can be a bad choice globally. It is $(I_8, 1)$, with one hole and interval I_8 having keys $\{1, 2, \ldots, 8\}$ whose weights are as follows:

key	1	2	3	4	5	6	7	8
weight	7	5	0	5	0	5	0	5

Figure 5 shows three possible subtrees T_{5_a}, T_{5_b}, and T_{5_c} for $(I_8, 1)$. By inspection, T_{5_a} has cost 49 for S-subproblem $(I_8, 1)$, while T_{5_b} and T_{5_c} cost 50 but weigh 2 units less. Suppose, in a larger tree, that T_{5_a} occurs as the left child of a node N, as shown in Figure 5. Let T_N be the subtree rooted at N. Suppose that each hole of T_{5_a} is also a hole at N. (For example, N might be $\langle v < 9 \rangle$.) Then replacing T_{5_a} by T_{5_b} would reduce the overall cost by at least 1 unit. This is because the contribution of T_{5_a} to the cost of T_N is not the cost of T_{5_a}; rather, it is its cost *plus its weight*, and the cost plus weight of T_{5_b} is 1 unit less.

Next we construct the full S-subproblem $(I, h) = (I_{15}, 2)$ for Theorem 4. It has two holes, and extends the above S-subproblem $(I_8, 1)$ to a larger interval $I_{15} = \{1, 2, \ldots, 15\}$ with the following symmetric weights:

key	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
weight	7	5	0	5	0	5	0	5	0	5	0	5	0	5	7

[3] Note that h' is h plus the number of equality tests on the path from the root of T^* to N. The algorithm does not determine which keys are used in those equality tests until after it solves (I_N, h').

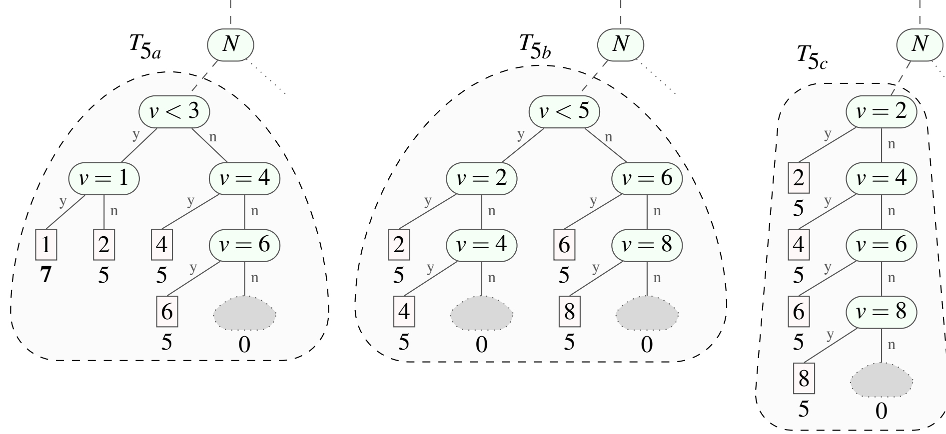

Figure 5: Three trees (circled and lightly shaded) for S-subproblem $(I_8,1)$. T_{5_a} has cost 49 and weight 22. T_{5_b} and T_{5_c} have cost 50 but weight 20. T_{5_a} is optimal for $(I_8,1)$. Among trees that don't contain the weight-7 key 1, trees T_{5_b} and T_{5_c} have minimum cost. Subtrees marked with 0 (dark shaded) contain keys of weight 0.

We use the following terminology to distinguish the different types of keys in a subtree. Given an S-subproblem (I',h') of (I,h), and a tree T' for (I',h'), the keys of I' that appear in the leaves of T' are *T'-queries*. The other keys in interval I', which are holes in T', are *T'-holes*. (We don't introduce new terminology for the comparison keys in T'.) We drop the prefix T' from these terms when it is understood from context.

To analyze $(I_{15},2)$ we need some utility lemmas. We start with one that will help us characterize how weight-0 queries increase costs. This lemma (Lemma 8 below) is in fact general and it holds for S-subproblems of an arbitrary instance of 2WCST. Define two integer sequences $\{d_m\}$ and $\{e_m\}$, as follows: $d_1=0$, $d_2=3$, $e_1=0$, $e_2=2$, $e_3=6$, and

$$
\begin{aligned}
d_m &= m+\min\{d_i+d_{m-i} : 1\le i<m\} \quad \text{for } m\ge 3,\\
e_m &= m+\min\{d_i+e_{m-i} : 1\le i<m\} \quad \text{for } m\ge 4.
\end{aligned}
$$

By calculation, $d_3=6$, $d_4=10$, $d_5=14$, $e_4=9$, $e_5=13$, and $e_6=18$.

Consider a tree T' for an S-subproblem of some arbitrary instance of 2WCST (not necessarily our specific instance (K,p)). A subset Q of T'-queries will be called *T'-separated* (or simply *separated*, if T' is understood from context) if for any two $k,k'\in Q$, with $k<k'$, there is a T'-query k'' that separates them, that is $k<k''<k'$. Also, if $Q\setminus\{f\}$ is T'-separated for some $f\in Q$, then we say that Q is *nearly T'-separated*.

Lemma 8. *Let T be a tree for an S-subproblem of some arbitrary instance of* 2WCST. *Let Q be a set of T-queries and $m=|Q|$.* (i) *If Q is T-separated then the total depth (i.e., the sum of the depths) in T of the keys in Q is at least d_m.* (ii) *If Q is nearly T-separated then the total depth in T of the keys in Q is at least e_m.*

The proof of Lemma 8 is a straightforward induction — we postpone it to the end of this section, and proceed with our analysis.

Now we focus our attention on our instance (K,p), and we characterize the weights and costs of optimal subtrees for certain subproblems. For $1\le\ell\le 14$, let $I_\ell=\{1,2,\ldots,\ell\}$ denote the subinterval of I_{15} containing

its first ℓ keys. These keys have ℓ weights (in order) $\{7,5,0,5,\ldots\}$: one key of weight 7, then $\lfloor \ell/2 \rfloor$ even keys of weight 5, separated by odd keys of weight 0. Let $\ell_+ = 1 + \lfloor \ell/2 \rfloor$ be the number of positive-weight keys in I_ℓ. Note that each S-subproblem (I_ℓ, h') can be solved by a tree with $\ell_+ - h'$ positive-weight queries, having h' (positive-weight) hole keys.

Lemma 9. *Consider any S-subproblem (I_ℓ, h') with $\ell \leq 14$ and $\ell_+ - h' = 4$. Let T' be an optimal tree for (I_ℓ, h'). Then T' has weight 22 and cost 49 (like T_{5a}).*

Proof. As T' is fixed throughout the proof, the terms *holes*, *queries*, and *separated*, mean *T'-holes*, *T'-queries*, and *T'-separated* as defined earlier, unless otherwise specified.

Let h_0 be the number of weight-0 holes and q_+ the number of queries with positive weight. We use the following facts about T'.

(F1) *T' costs at most 49.* Indeed, one way to solve (I_ℓ, h') is as follows: take the h' rightmost weight-5 keys in I_ℓ to be the holes, then handle the remaining $\ell_+ - h' = 4$ queries with positive weight (queries $1,2,4,6$), along with any weight-0 queries $3,5,7,\ldots$, using tree T_{5a}, at cost 49.

(F2) $q_+ = 4 + h_0$. This follows by simple calculation: $q_+ = \ell_+ - (h' - h_0) = 4 + h_0$.

(F3) *T' does not contain four separated weight-5 queries.* Indeed, otherwise, by Lemma 8, T' would cost at least $5 \cdot d_4 = 50 > 49$, contradicting (F1).

To finish we show that T' costs at least 49. Along the way we show it has weight 22.

Case 1: First consider the case that $h_0 = 0$. By (F2), *there are* 4 *positive-weight queries in T'*. Since $h_0 = 0$, all weight-0 keys are queries in T', so the set of all weight-5 queries in T' is separated, and by (F3), *there are at most three such queries*. The fourth positive-weight query must be the weight-7 query, query 1. So *the positive-weight queries in T' are the weight-7 query and three separated weight-5 queries.*

So T' has total weight 22, as desired. Further, by Lemma 8, the four positive-weight queries in T' have total depth at least e_4 in T'. So T' costs at least $5 \cdot e_4 + (7-5) \cdot j = 45 + 2j$, where j is the depth of the weight-7 query. If $j \geq 2$, by the previous bound, T' costs at least 49, and we are done. In the remaining case we have $j = 1$ (as $j = 0$ is impossible), so the weight-7 query is a child of the root. The three weight-5 queries are in the other child's subtree (and are a separated subset there), so by Lemma 8 have total depth at least $d_3 = 6$ in that subtree, and therefore total depth at least 9 in T'. So the total cost of T' is at least $7 + 5 \cdot 9 > 49$, contradicting (F1).

Case 2: In the remaining case $h_0 \geq 1$. By (F2), there are $q_+ = 4 + h_0$ positive-weight queries in T'. Let $q_5 \geq q_+ - 1$ be the number of weight-5 queries in T'. Since all but h_0 of the weight-0 queries are in T', there is a separated set of $q_5 - h_0$ weight-5 queries in T'. By (F3), $q_5 - h_0 \leq 3$.

This (with $q_+ = 4 + h_0$ and $q_5 \geq q_+ - 1$) implies $q_5 = h_0 + 3 = q_+ - 1$. This implies that the weight-7 query is in T', along with some $q_5 - h_0 = 3$ separated weight-5 queries. Reasoning as in Case 1, the cost of these four queries alone is at least 49. But T' contains at least one additional weight-5 query (as $q_5 = 3 + h_0 > 3$), so T' costs strictly more than 49, contradicting (F1). Thus Case 2 cannot actually occur. □

Lemma 10. *Consider any S-subproblem (I_ℓ, h') with $\ell \leq 14$ and $\ell_+ - h' = 5$. Let T' be an optimal tree for (I_ℓ, h'). Then T' has weight 27 and cost 69 (like T_{6a} in Figure 6).*

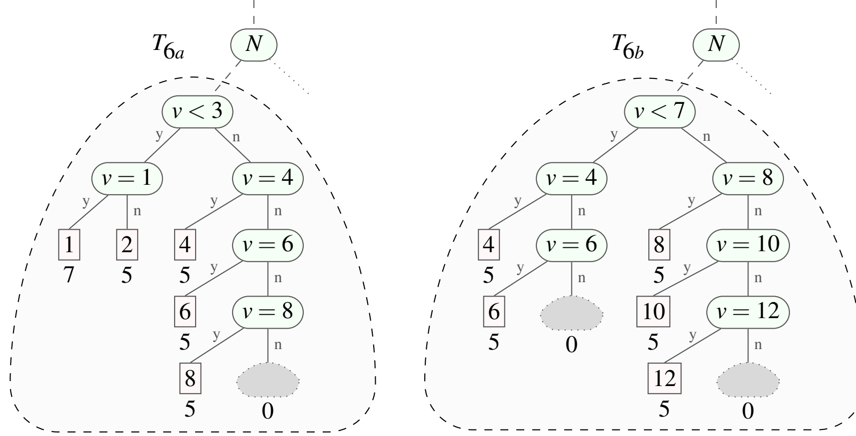

Figure 6: Trees T_{6a} and T_{6b}, with five positive weight queries. Tree T_{6a} has cost 69 and weight 27. Tree T_{6b} has cost 70 and weight 25.

Proof. Again, throughout the proof, unless otherwise specified, the terms *holes*, *queries*, and *separated*, are all with respect to T'. Let h_0 be the number of weight-0 holes and q_+ the number of queries with positive weight. We use the following facts about T'.

(F4) *T' costs at most 69.* Indeed, one can solve (I_ℓ, h') is as follows: take the h' rightmost weight-5 keys in I_ℓ to be the holes, then handle the remaining $\ell_+ - h' = 5$ queries with positive weight (queries $1,2,4,6,8$), along with any weight-0 queries $3,5,7,\ldots$, using tree T_{6a} at cost 69.

(F5) $q_+ = 5 + h_0$. This follows by straightforward calculation: $q_+ = \ell_+ - (h' - h_0) = 5 + h_0$.

(F6) *T' does not contain five separated weight-5 queries.* Indeed, otherwise, by Lemma 8, T' would cost at least $5 \cdot d_5 = 70 > 69$, a contradiction.

To finish, we show that T' has cost at least 69. Along the way we show it has weight 27.

<u>Case 1</u>: First consider the case that $h_0 = 0$. By (F5), *there are* 5 *positive-weight queries in T'*. Also, since $h_0 = 0$, all weight-0 keys are queries in T', so the set of all weight-5 queries in T' is separated, and by (F6), *there are at most four of them*. The fifth positive-weight query must be the weight-7 query, query 1. So *the positive-weight queries in T' are the weight-7 query and four separated weight-5 queries.*

So T' has total weight 27. Further, by Lemma 8, the five positive-weight queries in T' have total depth at least e_5 in T'. So T' costs at least $5 \cdot e_5 + (7-5) \cdot j = 65 + 2j$, where j is the depth of the weight-7 query. If $j \geq 2$ then, by the previous bound, T' costs at least 69, and we are done. In the remaining case we have $j = 1$ (as $j = 0$ is impossible), so the weight-7 query is a child of the root. The four weight-5 queries are in the other child's subtree (and form a separated set there), so by Lemma 8 have total depth at least $d_4 = 10$ in that subtree, and therefore total depth at least 14 in T'. So the total cost of T' is at least $7 + 5 \cdot 14 > 69$, contradicting (F4).

<u>Case 2</u>: In the remaining case, $h_0 \geq 1$. By (F5), there are $q_+ = 5 + h_0$ positive-weight queries in T'. Let $q_5 \geq q_+ - 1$ be the number of weight-5 queries in T'. Since all but h_0 of the weight-0 queries are in T', there is a separated set of $q_5 - h_0$ weight-5 queries in T'. By (F6), $q_5 - h_0 \leq 4$.

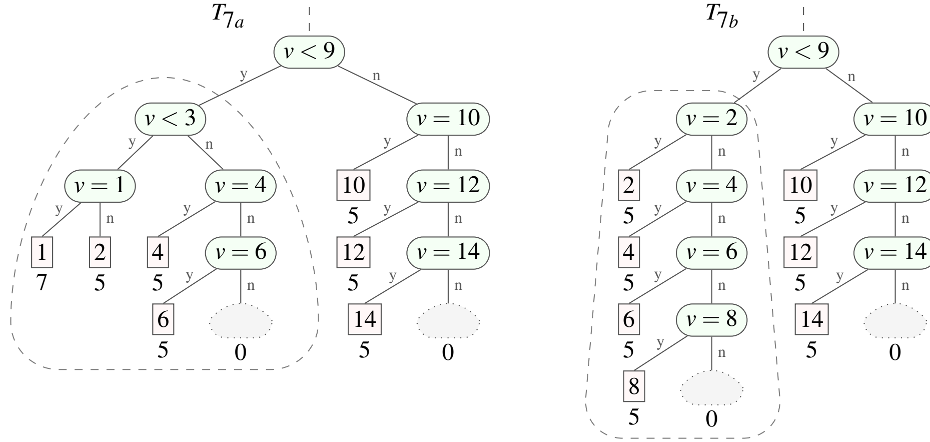

Figure 7: Spuler's algorithm fails on the S-subproblem $(I_{15},2)$. The algorithm computes a tree of cost 116, such as T_{7_a} above, but there are trees, such as T_{7_b}, of cost 115. The two trees' left subtrees are T_{5_a} and T_{5_c}.

This (with $q_+ = 5+h_0$ and $q_5 \geq q_+ - 1$) implies $q_5 = h_0 + 4 = q_+ - 1$. This implies that the weight-7 query is in T', along with some separated set of $q_5 - h_0 = 4$ weight-5 queries. Reasoning as in Case 1, the cost of these five queries alone is at least 69. But T' contains at least one additional weight-5 query (as $q_5 = 4 + h_0 > 4$), so T' costs strictly more than 69, contradicting (F4). Thus Case 2 cannot actually occur.

□

To conclude the proof of Theorem 4, we prove that $(I_{15},2)$ has the necessary properties:

Lemma 11. *Let T^* be any optimal tree for S-subproblem $(I_{15},2)$. Then T^* has at least one node N such that, for the S-subproblem $(I_N, |H_N|)$ arising at N, the subtree T^*_N rooted at N in T^* does* not *have minimum cost,* $\mathsf{opt}^*(I_N, |H_N|)$*, for that S-subproblem.*

Throughout the proof, unless otherwise specified, the terms *holes*, *queries*, and *separated*, are all with respect to T^*. We use the following properties of T^*:

(P1) *T^* costs at most 115.* Indeed, one way to solve $(I_{15},2)$ is to take the two weight-7 keys as holes, then use tree T_{7_b} in Figure 7, of cost 115. As T^* is optimal, it costs at most 115.

(P2) *The root of T^* does a less-than comparison.* Indeed, by [1, Theorem 5], since T^* is optimal for its queries, if T^* does an equality-test at the root, then the total query weight in T^* is at most four times the maximum query weight. But the total query weight in T^* is at least $7 \cdot 5 = 35$, while the maximum query weight is at most 7.

(P3) *In T^* there are seven positive-weight queries, and the set of weight-5 queries is separated (by weight-0 queries).* To show this, we show that no weight-0 key is a hole. Suppose otherwise for contradiction. Let k' be a weight-0 hole. We can assume without loss of generality that k' is not used in any node of T^* as an inequality key, for otherwise we can modify T^* to not use it, without changing its cost, by replacing it with the weight-5 key $k'' = k' + 1$ (which could be a hole or a query). Since k' is a T^*-hole, by definition, k' also cannot be used as an equality key. So we can assume that k' does not appear as a comparison key in T^*. Let $k \in \{k' \pm 1\}$ be a weight-5 query in T^*. (Query k exists in T^* — otherwise

$\{k'-1, k', k'+1\}$ would all be holes.) Replace k throughout T^* by k'. As k' and k are adjacent keys and k' does not occur in T^*, the resulting tree $\bar{T}$ still solves $(I_{15}, 2)$, and $\bar{T}$ costs less than T^* (as $\bar{T}$ uses the weight-0 key k' instead of the weight-5 key k). This contradicts the optimality of T^*.

By (P3), T^* has seven positive-weight queries. Using condition (P2) and left-right symmetry of subproblem $(I_{15}, 2)$, we can assume that the left subtree of T^* has at least four of the seven. (Note that "flipping" the tree, namely replacing each key k by $16-k$ and swapping the yes and no-subtrees, would map each inequality comparison $\langle v < k \rangle$ to $\langle v \leq 16-k \rangle$, while our model uses only strict inequalities. However, this latter comparison is equivalent to $\langle v < 17-k \rangle$.) Let T' be the left subtree. Denote the S-subproblem that T' solves by (I_ℓ, h'). To prove the lemma, assume for contradiction that T' is optimal for its S-subproblem, and proceed by cases:

Case 1: T' has four positive-weight queries. That is, T' solves an S-subproblem (I_ℓ, h') where $\ell_+ - h' = 4$. By Lemma 9, T' has cost 49 and weight 22. The right subtree T'' of T^* has the three remaining positive-weight queries, the leftmost two of which are separated in T'' by a zero-weight query (using (P3)). By Lemma 8 (ii), T'' has cost at least $5 \cdot e_3 = 30$ and weight at least 15. The cost of T^* is its weight plus the costs of T' and T''. By the above observations, this is at least $(22+15)+49+30 = 116$, contradicting (P1).

Case 2: T' has five positive-weight queries. That is, T' solves an S-subproblem (I_ℓ, h') where $\ell_+ - h' = 5$. By Lemma 10, T' has cost 69 and weight 27. The right subtree T'' of T^* has the two other positive-weight queries, which have total depth at least $1+1=2$ in T'', and each has weight at least 5. So T'' has cost, and weight, at least $5 \cdot 2 = 10$. The cost of T^* is its weight plus the costs of T' and T''. By the above observations, this is at least $(27+10)+69+10 = 116$, contradicting (P1).

Case 3: T' has six or seven positive-weight queries. Let set S consist of just the first six of these queries. Since T' is the left subtree of T^* (which has seven positive-weight queries) S does not contain the last key, 15. So (using (P3)) all queries in S, except possibly $\{1, 2\}$, are separated by weight-zero queries in T'. By Lemma 8 (ii), T' has cost at least $5 \cdot e_6 = 90$. The cost of T^* is its weight (at least $7 \cdot 5 = 35$), plus the cost of its left and right subtrees (at least 90, counting T' alone). So T^* costs at least $35+90 = 125$, contradicting (P1).

This proves the lemma and Theorem 4. □

Finally we prove Theorem 3.

Theorem 3. Consider any execution of Spuler's algorithm on the S-subproblem (I, h) from Theorem 4, breaking ties arbitrarily. Let T be the tree it computes for that S-subproblem. By Theorem 4, either T is not optimal for (I, h), or some subtree T' of T is not optimal for its S-subproblem (I', h'). So Spuler's algorithm must compute a non-optimal solution to at least one S-subproblem. □

In fact, for this instance (I, h), Spuler's algorithm (as implemented via the Python code in Appendix B) computes a non-optimal tree of cost 116, such as T_{7_a} in Figure 7. By inspection, tree T_{7_b} in that figure costs 115, so T_{7_a} is not optimal.

Discussion. As mentioned earlier, this counterexample is just for a subproblem. This subproblem has $h = 2$ holes, so it does not represent a complete instance of 2WCST for which Spuler's algorithm would give an incorrect *final* result. However, this counterexample does demonstrate that Spuler's algorithm solves some *subproblems* incorrectly, so that the recurrence relation underlying its dynamic program is incorrect. At a minimum, this suggests that any proof of correctness for Spuler's algorithm would require a more delicate approach. Anderson et al. [1] establish some conditions on the weights of equality-test keys in

optimal trees. It may be possible to leverage the bounds from [1] to show that bad subproblems — those that are not solved correctly by the algorithm — never appear as subproblems of an optimal complete tree. For example, per Anderson et al's Theorem 5 for any equality-test node in any optimal tree, the weight of the node's key must be at least one quarter of the total weight of the keys that reach the node. Hence, if a subproblem (I',h') is solved by some subtree T' of an optimal tree T^*, then each hole key in T' must have weight at least one third of the total weight of the queries in T'. This implies that the subproblem $(I_{15},2)$ in the proof of Theorem 3 cannot actually occur in any optimal tree for $(I_{15},0)$.

While the question of correctness of Spuler's algorithm is somewhat intriguing, it should be noted that showing its correctness will not improve known complexity bounds for 2WCST, as there are faster 2WCST algorithms that are known to be correct [1, 3].

3.1 Proof of Lemma 8.

Here is the promised proof of Lemma 8.

Lemma 8. Recall that T is a tree for some S-subproblem and Q is a subset of the queries in T, with $m = |Q|$.

Part (i). Assume that Q is separated. Our goal is to show that the total depth in T of queries in Q is at least d_m, as defined before Lemma 8. It is convenient to recast the problem as follows. Change the weight of each query in Q to 1. Change the weight of each query not in Q to 0. We will refer to the resulting cost of a tree as *modified cost*. Now we need to show that the modified cost of T is at least d_m. The proof is by induction on m.

The base cases (when $m = 1,2$) are easily verified, so consider the inductive step, for some given $m \geq 3$. We assume that T and Q are chosen to minimize the modified cost of T, subject to $|Q| = m$. Call this the *minimality assumption*.

Suppose T does an inequality test at the root. Let T_1 and T_2 be the left and right subtrees of T, and for $a \in \{1,2\}$ let $Q_a \subseteq Q$ contain the queries in Q that fall in T_a. Let $i = |Q_1|$, so that $|Q_2| = m - i$. For $a \in \{1,2\}$, query set Q_a is T_a-separated. By the minimality assumption, $0 \notin \{i, m-i\}$. The modified cost of T is its weight (m), plus the modified costs of T_1 and T_2. By the inductive assumption, this is at least $m + d_i + d_{m-i} \geq d_m$, as desired.

Suppose T does an equality test at the root. The minimality assumption implies that the equality-test key has non-zero (modified) weight. (This follows via the argument given for Property (P2) in the proof of Lemma 11, using Anderson et al's Theorem 5 or Corollary 3.) So the equality-test key is in Q. Let T_1 be the no-subtree of T and let $Q_1 \subseteq Q$ contain the queries in Q that fall in T_1; so we have $|Q_1| = m - 1$. Set Q_1 is T_1-separated, so by the inductive assumption, T_1 has modified cost at least d_{m-1}. So the modified cost of T is at least $m + d_{m-1} = m + d_1 + d_{m-1} \geq d_m$, as desired.

Part (ii). The proof of Part (ii) follows the same inductive argument as above. The base cases for $m = 1,2$ are trivial. The verification of the base case for $m = 3$ is by straightforward case analysis. In the inductive step, the only significant difference is in the case when T does an inequality test at the root. Since Q is now only nearly separated, Q_1 will be T_1-separated while Q_2 will be nearly T_2-separated (or vice versa), giving us that the modified cost of T is at least $m + d_i + e_{m-i} \geq e_m$. ☐

Note: We would like to use this opportunity to acknowledge yet another error in the literature on binary split trees, this one in our own paper [3]. In that paper we introduced a perturbation method that can be used to extend algorithms for binary search trees with keys of distinct weights to instances where key-weights need not be distinct, and we claimed that this method can be used to speed up the computation of optimal binary split trees to achieve running time $O(n^4)$. (Recall that in binary split trees from [7, 10, 5], the equality-test key in each node must be a most likely key among keys reaching the node.) As it turns out, this claim is not valid. In essence, the perturbation approach from [3] does not apply to binary split trees because such perturbations affect the choice of the equality-test key and thus also the validity of some trees. See [4] for an erratum, full proofs of the remaining results, and pointers to follow-up work.

References

[1] R. Anderson, S. Kannan, H. Karloff, and R. E. Ladner. Thresholds and optimal binary comparison search trees. *Journal of Algorithms*, 44:338–358, 2002.

[2] G-H. Chen and L-T. Liu. Optimal multiway generalized split trees. *International Journal of Computer Mathematics*, 41(1-2):39–47, January 1991.

[3] M. Chrobak, M. J. Golin, J. I. Munro, and N. E. Young. Optimal search trees with 2-way comparisons. In Khaled Elbassioni and Kazuhisa Makino, editors, *Algorithms and Computation. ISAAC 2015*, volume 9472 of *Lecture Notes in Computer Science*, pages 71–82. Springer Berlin Heidelberg, 2015. See [4] for erratum. `doi:10.1007/978-3-662-48971-0_7`.

[4] M. Chrobak, M. J. Golin, J. I. Munro, and N. E. Young. Optimal search trees with two-way comparisons, 2021. Includes erratum for [3]. `arXiv:1505.00357`.

[5] J. H. Hester, D. S. Hirschberg, S. H. Huang, and C. K. Wong. Faster construction of optimal binary split trees. *Journal of Algorithms*, 7:412–424, 1986.

[6] S-H. Huang and C. K. Wong. Generalized binary split trees. *Acta Informatica*, 21(1):113–123, 1984.

[7] S-H. Huang and C. K. Wong. Optimal binary split trees. *Journal of Algorithms*, 5:69–79, 1984.

[8] D. E. Knuth. Optimum binary search trees. *Acta Informatica*, 1:14–25, 1971.

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

[10] Y. Perl. Optimum split trees. *Journal of Algorithms*, 5:367–374, 1984.

[11] B. A. Sheil. Median split trees: a fast lookup technique for frequently occurring keys. *Communications of the ACM*, 21:947–958, 1978.

[12] D. Spuler. Optimal search trees using two-way key comparisons. *Acta Informatica*, 31(8):729–740, 1994.

[13] D. A. Spuler. *Optimal search trees using two-way key comparisons*. PhD thesis, James Cook University, 1994.

APPENDIX

A Python code for Huang and Wong's GBSPLIT algorithm

See Figure 8.

B Python code for Spuler's 2WCST algorithm

See Figure 9.

```
#!/usr/bin/env python3.6
from collections import namedtuple
from functools import lru_cache

memoize = lru_cache(maxsize=None)
Tree = namedtuple('Tree', 'cost weight holes')

def size(i, j, h):
    return j - i + 1 - h

def huang1984(weights):
    '''Returns cost as computed by Huang and Wong's GBSPLIT algorithm (1984).'''

    wts = [weights[k] for k in sorted(weights.keys())]

    @memoize
    def tree(i, j, h):
        '''Returns tree for opt([i,j], h)'''

        interval = frozenset(range(i, j+1))

        if size(i, j, h) == 0:
            return Tree(cost=0, weight=0, holes=interval)

        def candidate(k, h_l, h_r):
            left = tree(i, k-1, h_l)
            right = tree(k, j, h_r)
            holes = left.holes | right.holes
            eq_key = min(holes, key=lambda k: wts[k])
            weight = left.weight + right.weight + wts[eq_key]
            return Tree(cost=weight + left.cost + right.cost,
                        weight=weight,
                        holes=holes - {eq_key})

        return min(candidate(k, h_l, h-h_l+1)
                   for k in interval
                   for h_l in range(h+2)
                   if size(i, k-1, h_l) >= 0 and size(k, j, h-h_l+1) >= 0)

    return tree(0, len(weights)-1, 0).cost

# The instance (K, p) from the proof of Theorem 1:
weights = dict(B4=20,
               A3=20, V3=20,
               A2=20, F2=20, T2=20, X2=20,
               A1=20, D1=22, F1=20, Q1=20, S1=20, U1=20, W1=20, Y1=20,
               B0=10, C0= 5, D0=10, E0=10, N0=10, P0=10, Q0=10, R0=10,
               S0=10, T0=10, U0=10, V0=10, W0=10, X0=10, Y0=10, Z0=10)

assert huang1984(weights) == 1763

# Increasing a weight _lowers_ the cost computed by the algorithm.
weights['D1'] += 0.99
assert huang1984(weights) < 1763
```

Figure 8: Python code for Huang and Wong's GBSPLIT algorithm

```python
#!/usr/bin/env python3.6
from collections import namedtuple
from functools import lru_cache

memoize = lru_cache(maxsize=None)
Tree = namedtuple('Tree', 'cost weight holes')

def size(i, j, h):
    return j - i + 1 - h

def spuler1994(wts, n_holes):
    '''Returns cost as computed by Spuler's 2WCST algorithm (1994)'''

    @memoize
    def tree(i, j, h):
        '''Returns tree for opt([i, j], h).'''

        interval = frozenset(range(i, j+1))

        if size(i, j, h) == 1:
            k = min(interval, key=lambda k: wts[k])
            return Tree(cost=0, weight=wts[k], holes=interval - {k})

        def equality_candidate():
            right = tree(i, j, h+1)
            eq_key = min(right.holes, key=lambda k: wts[k])
            return Tree(cost=wts[eq_key] + right.weight + right.cost,
                        weight=wts[eq_key] + right.weight,
                        holes=right.holes - {eq_key})

        def less_than_candidate(k, h_l, h_r):
            left, right = tree(i, k-1, h_l), tree(k, j, h_r)
            return Tree(cost=left.weight + right.weight + left.cost + right.cost,
                        weight=left.weight + right.weight,
                        holes=left.holes | right.holes)

        return min(equality_candidate(),
                   min(less_than_candidate(k, h_l, h-h_l)
                       for k in interval
                       for h_l in range(h+1)
                       if size(i, k-1, h_l) >= 1 and size(k, j, h-h_l) >= 1))

    return tree(0, len(wts)-1, n_holes).cost

# The instance (K, p) and subproblem (I_15, 2) from the proof of Theorem 2:
weights1 = [7, 5, 0, 5, 0, 5, 0, 5, 0, 5, 0, 5, 0, 5, 7]
assert spuler1994(weights1, 2) == 116

# Increasing some weights _lowers_ the cost computed by the algorithm:
weights2 = [9, 5, 0, 5, 0, 5, 0, 5, 0, 5, 0, 5, 0, 5, 9]
assert spuler1994(weights2, 2) == 115
```

Figure 9: Python code for Spuler's 2WCST algorithm