Classification via two-way comparisons

Marek Chrobak * Neal E. Young †

February 21, 2023

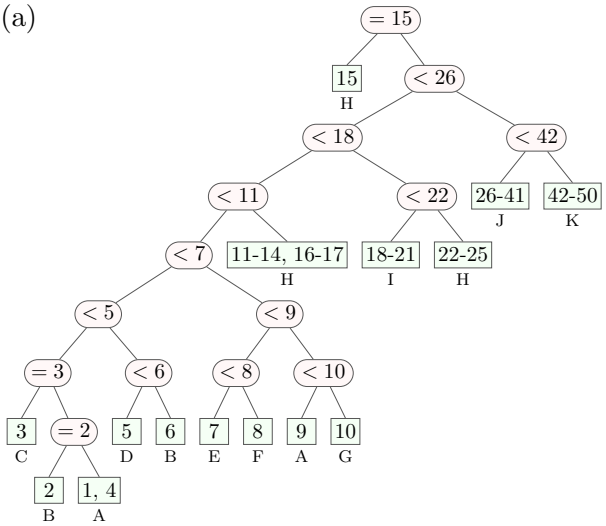
Abstract

Given a weighted, ordered query set Q and a partition of Q into classes, we study the problem of computing a minimum-cost decision tree that, given any query $q \in Q$, uses equality tests and less-than comparisons to determine the class to which q belongs. Such a tree can be much smaller than a lookup table, and much faster and smaller than a conventional search tree. We give the first polynomial-time algorithm for the problem. The algorithm extends naturally to the setting where each query has multiple allowed classes.

*University of California, Riverside. Research partially supported by National Science Foundation grant CCF-2153723.

†University of California, Riverside.

(a)



(b)

q	cls	wt	q	cls	wt	q	cls	wt
1	A	6	18	I	10	35	J	2
2	B	7	19	I	8	36	J	2
3	C	7	20	I	15	37	J	1
4	A	6	21	I	17	38	J	1
5	D	8	22	H	7	39	J	1
6	B	8	23	H	8	40	J	2
7	E	7	24	H	15	41	J	1
8	F	6	25	H	10	42	K	3
9	A	8	26	J	1	43	K	3
10	G	7	27	J	2	44	K	3
11	H	10	28	J	2	45	K	3
12	H	11	29	J	1	46	K	3
13	H	10	30	J	2	47	K	3
14	H	9	31	J	2	48	K	3
15	H	500	32	J	100	49	K	3
16	H	9	33	J	1	50	K	3
17	H	10	34	J	2			

Figure 1: An optimal two-way-comparison decision tree (2WCDT) for the problem instance shown on the right. The instance (but not the tree) is from [2, 3, Figure 6]. Each leaf (rectangle) is labeled with the queries that reach it, and below that with the class for the leaf. The table gives the class and weight of each query $q \in Q = [50] = \{1, 2, \ldots, 50\}$. The tree has cost 2055, about 11% cheaper than the tree from [2, 3], of cost 2305.

1 Introduction

Given a weighted, ordered *query* set Q partitioned into classes, we study the problem of computing a minimum-cost decision tree that uses equality tests (e.g., "$q = 4$?") and less-than tests (e.g., "$q < 7$?") to quickly determine the class of any given query $q \in Q$. (Here the cost of a tree is the weighted sum of the depths of all queries, where the depth of a given query $q \in Q$ is the number of tests the tree makes when given query q.) We call such a tree a *two-way-comparison decision tree (*2WCDT*)*. See Figure 1.

A main use case for 2WCDTs is when the number of classes is small relative to the number of queries. In this case a 2WCDT can be significantly smaller than a lookup table, and, likewise, faster and smaller than a conventional search tree, because a search tree has to identify a given query q (or the inter-key interval that q lies in) whereas a decision tree only has to identify q's class. Because they can be faster and more compact, 2WCDTs are used in applications such as dispatch trees, which allow compilers and interpreters to quickly resolve method implementations for objects declared with type inheritance [2, 3]. (Each type is assigned a numeric ID via a depth-first search of the inheritance digraph. For each method, a 2WCDT maps each ID to the appropriate method resolution.)

Chambers and Chen give a heuristic to construct low-cost 2WCDTs, but leave open whether minimum-cost 2WCDTs can be found in polynomial time [2, 3]. We give the first polynomial-time algorithm to find minimum-cost 2WCDTs. The algorithm runs in time $O(n^4)$, where $n = |Q|$ is the number of distinct query values, matching the best time bound known for a special type of 2WCDTs called *two-way-comparison search trees (*2WCSTs*)*, discussed below. The algorithm extends naturally to the setting where each query can belong to multiple classes, any one of which is acceptable as

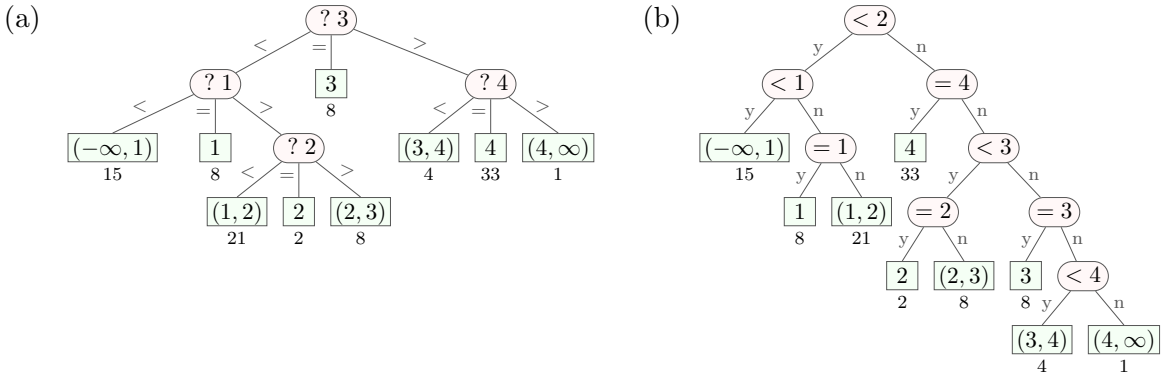


Figure 2: Tree *(a)* is a three-way-comparison search tree (3WCST). Tree *(b)* is a two-way-comparison search tree (2WCST) for the same instance. The query (or interval of queries) reaching each (rectangular) leaf is within the leaf. The weight of the query (or interval) is below the leaf.

an answer for the query. The extended algorithm runs in time $O(n^3m)$, where m is the sum of the sizes of the classes.

Related work. Various types of decision trees are ubiquitous in the areas of artificial intelligence, machine learning, and data mining, where they are used for data classification, clustering, and regression.

Here we study decision trees for one-dimensional data sets. In theoretical computer science, most work on such trees has focussed on search trees, that is, decision trees that must fully identify the query or the inter-key interval it lies in. Here is a brief summary of relevant work on such trees. One of our main contributions is to increase the understanding of trees based on two-way comparisons. These are not yet fully understood.

The tractability of finding a minimum-cost search tree depends heavily on the kind of tests that the tree can use. For some kinds of tests, the problem is NP-complete [12]. Early works considered trees in which each test compared the given query value q to some particular comparison key k, with *three* possible outcomes: the query value q is less than, equal to, or greater than k [6, §14.5], [14, §6.2.2]. (See Figure 4(a).) We call such trees *three-way-comparison search trees*, or 3WCSTs for short. In a 3WCST, the query values that reach any given node form an interval. This leads to a natural $O(n^3)$-time dynamic-programming algorithm with $O(n^2)$ subproblems for finding minimum-cost 3WCSTs [8]. Knuth reduced the time to $O(n^2)$ [13].

In practice each three-way comparison is often implemented by doing a less-than test followed by an equality test. Knuth [14, §6.2.2, Example 33] proposed exploring binary search trees that use these two tests directly in any combination. Such trees are called *two-way-comparison search trees (*2WCST*s)* [1]. For the so-called *successful-queries* variant, assuming that the query weights are normalized to sum to 1, there is always a 2WCST whose cost exceeds the entropy of the weight distribution by at most 1 [7]. The entropy is a lower bound on the cost of any binary search tree that uses Boolean tests of any kind. This suggests that restricting to less-than and equality tests need not be too costly [7].

Stand-alone equality tests introduce a technical obstacle not encountered with 3WCSTs. Namely, while (analogously to 3WCSTs) each node of a 2WCST is naturally associated with an interval of queries, not all queries from this interval necessarily reach the node. For this reason the dynamic-

programming approach for 3WCSTs does not extend easily to 2WCSTs. This led early works to focus on restricted classes of 2WCSTs, namely *median split trees* [16] and *binary split trees* [11, 15, 9]. These, by definition, constrain the use of equality tests so as to altogether sidestep the aforementioned technical obstacle. *Generalized binary split trees* are less restrictive, but the only algorithm proposed to find them [10] is incorrect [5]. Similarly, the first algorithms proposed to find minimum-cost 2WCSTs (without restrictions) were given without proofs of correctness [17, 18], and the recurrence relations underlying some of those proposed algorithms turned out to be demonstrably wrong [5].

In 1994, Spuler made a conjecture that leads to a natural dynamic program for 2WCSTs. Namely, that every instance admits a minimum-cost 2WCST with the *heaviest-first* property: that is, at any equality-test node $\langle = h \rangle$, *the comparison key h is heaviest among keys reaching the node* [18]. In a breakthrough in 2002, Anderson et al proved the conjecture for the so-called *successful-queries* variant, leading to an $O(n^4)$-time dynamic-programming algorithm to find minimum-cost 2WCSTs for that variant [1]. In 2021, Chrobak et al simplified their result (in particular, the handling of keys of equal weights, as discussed later) obtaining an $O(n^4)$-time algorithm for finding minimum-cost 2WCSTs [4].

Our contributions. Unfortunately these 2WCST algorithms don't extend easily to 2WCDTs. The main obstacle is that for some instances (e.g. in Figure 5) no minimum-cost 2WCDT has the crucial heaviest-first property. To overcome this obstacle we introduce a *splitting* operation (Definition 7), a correctness-preserving local rearrangement of the tree that can be viewed as an extension of the well-studied rotation operation to a more general class of trees, specifically, to trees whose allowed tests induce a laminar set family (Property 1).

We use splitting to identify an appropriate relaxation of the heaviest-first property that we call being *admissible* (Definition 4). Most of the paper is devoted to proving the following theorem:

Theorem 1. *If the instance is feasible, then some optimal tree is admissible.*

Section 3 gives the proof. Along the way it establishes new structural properties of optimal 2WCSTs and 2WCDTs. Section 4 shows how Theorem 1 leads to a suitable dynamic program and our main result:

Theorem 2. *There is an $O(n^3m)$-time algorithm for finding a min-cost* 2WCDT.

Remarks. The presentation above glosses over a secondary technical obstacle for 2WCSTs. For 2WCST instances with distinct query weights, the heaviest-first property uniquely determines the key of each equality test, so that the subset of queries that reach any given node in a 2WCST with the heaviest-first property must be *one of $O(n^4)$ predetermined subsets*. This leads to a natural dynamic program with $O(n^4)$ subproblems. (See Section 3.) But this does not hold for instances with non-distinct weights. This obstacle turns out to be more challenging than one might expect. Indeed, there are instances for which, for each of the 2^n subsets S of Q, there is a minimum-cost 2WCST, having the heaviest-first property, with a node u such that the set of queries reaching u is S. It appears that one cannot just break ties arbitrarily: it can be that, for two maximum-weight keys h and h' reaching a given node u, there is an optimal subtree in which u does an equality-test to h, but none in which u does an equality-test to h' [4, Figure 3]. Similar issues arise in finding

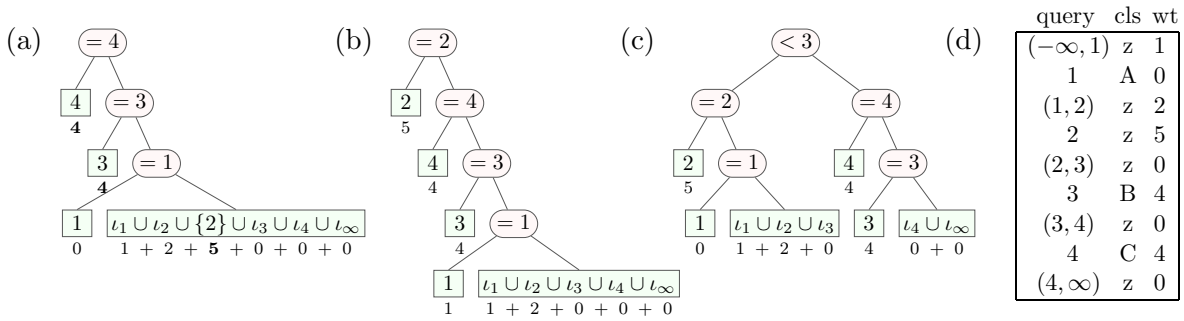


Figure 3: Three trees for the 2WCDT instance shown in *(d)*. The set of queries reaching each (rectangular) leaf is shown within the leaf (to save space, there ι_i denotes the inter-key open interval with right boundary i, e.g. $\iota_1 = (-\infty, 1)$, $\iota_2 = (1, 2)$). The associated weights are below the leaf. The optimal tree *(a)* has cost 36 and is not heaviest-first. Each heaviest-first tree (e.g. *(b)* of cost 41 or *(c)* of cost 56) is not optimal. These properties also hold if each weight is perturbed to make the weights distinct.

optimal *binary split trees*—these can be found in time $O(n^4)$ if the instance has distinct weights, while for arbitrary instances the best bound known is $O(n^5)$ [9].

Nonetheless, using a perturbation argument Chrobak et al show that an arbitrary 2WCST instance can indeed be handled as if it is a distinct-weights instance just by breaking ties among equal weights in an appropriate way [4]. We use the same approach here for 2WCDTs.

Most search-tree problems come in two flavors: the *successful-queries* variant and the *standard* variant. In the former, the input is an ordered set K of weighted keys, each comparison must compare the given query value to a particular key in K, and each query must be a value in K. In the latter, the input is augmented with a weight for each open interval between successive keys. Queries (called *unsuccessful queries*) to values in these intervals are also allowed, and must be answered by returning the interval in which the query falls. Our formal definition of 2WCDTs generalizes both variants.

The tractability of finding a minimum-cost search tree depends heavily on the kind of tests that the tree must use. For some kinds of tests, the problem is NP-complete [12]. Early works considered trees in which each test compared the given query value q to some particular comparison key k, with *three* possible outcomes: the query value q is less than, equal to, or greater than k [6, §14.5], [14, §6.2.2]. (See Figure 4(a).) We call such trees *three-way-comparison search trees*, or 3WCSTs for short. In a 3WCST, the query values that reach any given node form an interval. This leads to a natural $O(n^3)$-time dynamic-programming algorithm with $O(n^2)$ subproblems for finding minimum-cost 3WCSTs [8]. Knuth reduced the time to $O(n^2)$ [13].

Often, in practice, each three-way comparison is implemented by doing a less-than test followed by an equality test. Knuth [14, Section 6.2.2, Example 33] proposed exploring binary search trees that use these two tests directly. Such trees are called *two-way-comparison search trees (*2WCST*s)* [1]. For the so-called *successful-queries* variant, assuming that the query weights are normalized to sum to 1, there is always a 2WCST whose cost exceeds the entropy of the weight distribution by at most 1 [7]. The entropy is a lower bound on the cost of a binary search tree that uses Boolean tests of any kind. This suggests that restricting to less-than and equality tests need not be too costly [7].

But equality tests present a technical obstacle not encountered with 3WCSTs. Namely, while

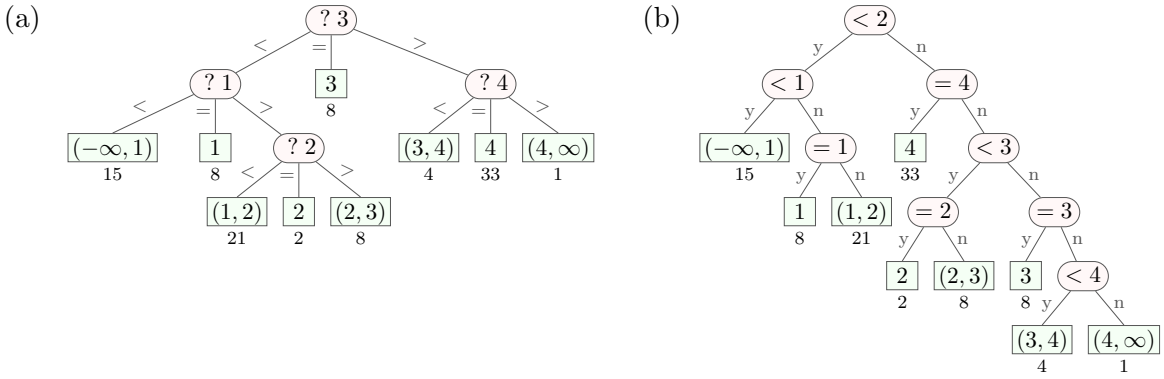


Figure 4: Tree *(a)* is a three-way-comparison search tree (3WCST). Tree *(b)* is a two-way-comparison search tree (2WCST) for the same instance. The query (or interval of queries) reaching each (rectangular) leaf is within the leaf. The weight of the query (or interval) is below the leaf.

(analogously to 3WCSTs) with each node of a 2WCST we can naturally associate an interval of queries, not all queries from this interval necessarily reach the node. For this reason the dynamic-programming approach for 3WCSTs does not extend easily to 2WCSTs. This led early works to focus on restricted classes of 2WCSTs, namely *median split trees* [16] and *binary split trees* [11, 15, 9]. These, by definition, constrain the use of equality tests so as to altogether sidestep the aforementioned technical obstacle. *Generalized binary split trees* are less restrictive, but the only algorithm proposed to find them [10] is incorrect [5]. Similarly, the first algorithms proposed to find minimum-cost 2WCSTs (without restrictions) were given without proofs of correctness [17, 18], and the recurrence relations underlying some of those proposed algorithms turned out to be demonstrably wrong [5].

In 1994, Spuler made a conjecture that leads to a natural dynamic program for 2WCSTs. Namely, that every instance admits a minimum-cost 2WCST with the *heaviest-first* property: that is, at any equality-test node $\langle = h \rangle$, *the comparison key* h *is heaviest among keys reaching the node* [18]. In a breakthrough in 2002, Anderson et al proved the conjecture for the so-called *successful-queries* variant, leading to an $O(n^4)$-time dynamic-programming algorithm to find minimum-cost 2WCSTs for that variant [1]. In 2021, Chrobak et al simplified their result (in particular, the handling of keys of equal weights, as discussed later) obtaining an $O(n^4)$-time algorithm for finding minimum-cost 2WCSTs [4].

Unfortunately these 2WCST algorithms don't extend easily to 2WCDTs. The main obstacle is that for some instances (e.g. in Figure 5) no minimum-cost 2WCDT has the crucial heaviest-first property. To overcome this obstacle we develop new machinery for reasoning about 2WCDTs. Using this machinery we identify an appropriate relaxation of the heaviest-first property, one that leads to the desired algorithm.

2 Definitions and technical overview

For the remainder of the paper, fix a 2WCDT instance $(Q, w, \mathcal{C}, K)$, where Q is a totally ordered finite set of *queries*, each with a weight $w(q) \geq 0$, the set $\mathcal{C} \subseteq 2^Q$ is a collection of *classes* of queries (where each class has a unique identifier), and $K \subseteq Q$ is the set of *keys*. Let $n = |Q|$ and $m = \sum_{c \in \mathcal{C}} |c|$. The problem is to compute a minimum-cost *two-way-comparison decision tree*

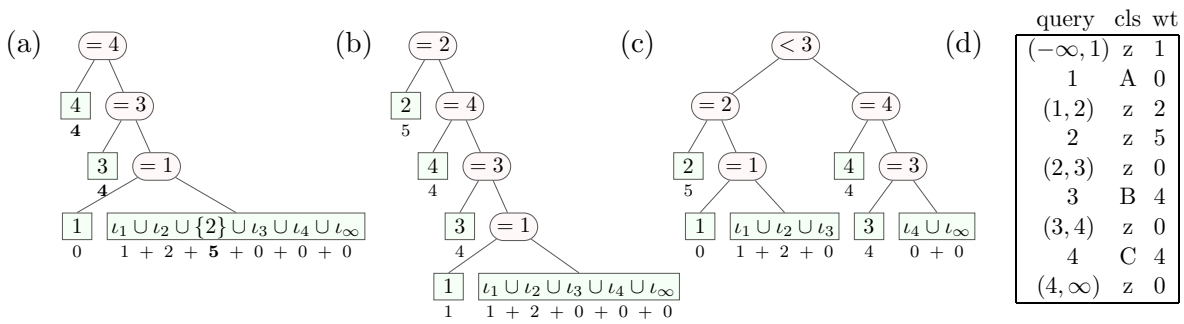


Figure 5: Three trees for the 2WCDT instance shown in *(d)*. The set of queries reaching each (rectangular) leaf is shown within the leaf (to save space, there ι_i denotes the inter-key open interval with right boundary i, e.g. $\iota_1 = (-\infty, 1)$, $\iota_2 = (1, 2)$). The associated weights are below the leaf. The optimal tree *(a)* has cost 36 and is not heaviest-first. Each heaviest-first tree (e.g. *(b)* of cost 41 or *(c)* of cost 56) is not optimal. These properties also hold if each weight is perturbed by a small amount to make the weights distinct.

*(*2WCDT*)* for the instance (as defined below).

To streamline presentation, throughout the paper we restrict attention to the model of decision trees that allows only less-than and equality tests. Our results extend naturally to decision trees that also use other inequality comparisons between queries and keys. See the end of Section 4 for details.

Definition 1 (2WCDT)**.** *A* two-way-comparison decision tree (2WCDT) *is a rooted binary tree T where each non-leaf node is a* test *of the form $\langle < k \rangle$ for some $k \in K$ such that $\min Q < k \leq \max Q$, or of the form $\langle = k \rangle$ for some $k \in K$, and the two children of the node are labeled with the two possible test outcomes ("yes" or "no"). Each leaf node is labeled with the identifier of one class in $\mathcal{C}$. This class must contain every query $q \in Q$ whose search (as defined next) ends at the leaf.*

For each $q \in Q$, the search for q in T *starts at the root, then recursively searches for q in the root's yes-subtree if q satisfies the root's test, and otherwise in the no-subtree. The search stops at a leaf, called the* leaf for q*. The path from the root to this leaf is called q's* search path*. We say that q* reaches *each node on this path, and q's* depth *in T is defined as the length of this path (equivalently, the number of comparisons when searching for q). The* cost *of T is the weighted sum of the depths of all queries in Q.*

A tree T is called irreducible *if, for each node u in T, (i) at least one query in Q reaches u, and (ii) if some class $c \in \mathcal{C}$ contains all the queries that reach u, then u is a leaf.*

For any $\ell, r \in Q$, let $[\ell, r]_Q$ and $[\ell, r]_K$ denote the query interval $\{q \in Q : \ell \leq q \leq r\}$ *and* key interval $\{k \in K : \ell \leq k \leq r\} = K \cap [\ell, r]_Q$.

Allowing K and Q to be specified as we do captures both the successful-queries and standard variants. The successful-queries variant corresponds to the case when $K = Q$. The standard variant is modeled by having one non-key query between every pair of consecutive keys, and before the minimum key and after the maximum key (so $|Q \setminus K| = |K| + 1$). Each such non-key query represents an interval between keys.

For ease of exposition, assume without loss of generality that each query belongs to some class, so $m \geq |Q|$ and the input size is $\Theta(n + m) = \Theta(m)$. Note that the instance is not necessarily

feasible, that is, it might not have a decision tree. (To be feasible, in addition to each query belonging to some class, each query interval that contains no keys must be contained in some class.) If the instance is feasible, some optimal tree is irreducible, so we generally restrict attention to irreducible trees. As we shall see, in an irreducible tree any given test is used in at most one node.

Definition 2 (ordering queries by weight)**.** *For any query subset $R \subseteq Q$ and integer $i \geq 0$ define* $\mathsf{heaviest}_i(R)$ *to contain the i heaviest queries in R (or all of R if $i \geq |R|$). For $q \in Q$, define* $\mathsf{heavier}(q)$ *to contain the queries (in Q) that are heavier than q. Define* $\mathsf{lighter}(q)$ *to contain the queries (in Q) that are lighter than q. Break ties among query weights arbitrarily but consistently throughout.*

Formally, we use the following notation to implement the consistent tie-breaking mentioned above. Fix an ordering of Q by increasing weight, breaking ties arbitrarily. For $q \in Q$ let $\tilde{w}(q)$ denote the rank of q in the sorted order. Throughout, given distinct queries q and q', define q to be lighter than q' if $\tilde{w}(q) < \tilde{w}(q')$ and heavier otherwise ($\tilde{w}(q) > \tilde{w}(q')$). So, for example $\mathsf{heaviest}_i(R)$ contains the last i elements in the ordering of R by increasing $\tilde{w}(q)$. The symbol $\perp$ represents the undefined quantity $\arg\max \emptyset$. Define $\tilde{w}(\perp) = w(\perp) = -\infty$, $\mathsf{heavier}(\perp) = Q$, and $\mathsf{lighter}(\perp) = \emptyset$.

Definition 3 (intervals and holes)**.** *Given any non-empty query subset $R \subseteq Q$, call* $[\min R, \max R]_Q$ *the* query interval *of R. Define $k^*(R)$ to be the heaviest key in R, if there is one (that is, $k^*(R) = \arg\max\{\tilde{w}(k) : k \in K \cap R\}$). Define also* $\mathsf{holes}(R) = [\min R, \max R]_Q \setminus R$ *to be the set of* holes *in R. We say that a hole $h \in \mathsf{holes}(R)$ is* light *if $\tilde{w}(h) < \tilde{w}(k^*(R))$, and otherwise* heavy.

The set of queries reaching a node u in a tree T is called u's query set*, denoted Q_u. The query interval, and light and heavy holes, for u are defined to be those for u's query set Q_u.*

Overview. Note that each hole $h \in \mathsf{holes}(Q_u)$ at a node u in a tree T must result from a failed equality test $\langle = h \rangle$ at a node v on the path from the root to u in T. In particular, $h \in K$. Further, if the hole is light, then h is not the heaviest key reaching v.

The problem has the following *optimal substructure* property. Any query subset $R \subseteq Q$ naturally defines the subproblem $\pi(R)$ induced by restricting the query set to R (that is, $\pi(R) = (R, w, \mathcal{C}_R, K)$ where $\mathcal{C}_R = \{c \cap R : c \in \mathcal{C}\}$). In any minimum-cost tree T for R, if T is not a leaf, then the yes-subtree and no-subtree of T must be minimum-cost subtrees for their respective subproblems.

Let $\mathsf{cost}(R)$ be the minimum cost of an irreducible tree for $\pi(R)$. (If R is empty, then $\mathsf{cost}(R) = \infty$, as no tree for R is irreducible.) Then the following recurrence holds:

Observation 1 (recurrence relation)**.** *Fix any $R \subseteq Q$. If $R = \emptyset$, then $\mathsf{cost}(R) = \infty$. Otherwise, if $(\exists c \in \mathcal{C})\, R \subseteq c$ (that is, R can be handled by a single leaf labeled c), then $\mathsf{cost}(R) = 0$. Otherwise, for any allowed test u, let $(R_u^{\mathsf{yes}}, R_u^{\mathsf{no}})$ be the bipartition of R into those queries that satisfy u and those that don't. Then*

$$\mathsf{cost}(R) = w(R) + \min_u \left(\mathsf{cost}(R_u^{\mathsf{yes}}) + \mathsf{cost}(R_u^{\mathsf{no}}) \right), \tag{1}$$

where the variable u ranges over the allowed tests such that R_u^{yes} and R_u^{no} are non-empty. (If there are no such tests then $\mathsf{cost}(R) = \infty$.)

The goal is to compute $\mathsf{cost}(Q)$ using a dynamic program that applies recurrence (1) recursively, memoizing results so that for each distinct query set R the subproblem for R is solved at most

once. (The algorithm as presented computes only $\mathsf{cost}(Q)$. It can be extended in the standard way to also compute an optimal tree.) The obstacle is that exponentially many distinct subproblems can arise.

Identity classification without equality tests. For intuition, consider first the variant of our problem in which $\mathcal{C}$ is the *identity classification*, that is $\mathcal{C} = \big\{\{q\} : q \in Q\big\}$, and only less-than tests $\langle < k\rangle$ are allowed (equality tests are not). In the absence of equality tests, there are no holes. When applying recurrence (1) recursively to $\mathsf{cost}(Q)$, each query set R that arises is a query interval. There are $O(n^2)$ such query intervals, and for each the right-hand side of the recurrence can be evaluated in $O(n)$ time. This yields an $O(n^3)$-time algorithm. This approach mirrors a classical dynamic-programming algorithm for 3WCSTS [8], as discussed in the introduction.

The algorithm extends easily to arbitrary classifications $\mathcal{C}$. Recall that a given query set R can be handled by a leaf (at zero cost) if and only if $R \subseteq c$ for some $c \in \mathcal{C}$. This condition can be checked in constant time given (ℓ, r) such that $R = [\ell, r]_Q$ (after an appropriate precomputation, e.g., for each ℓ, precompute the maximum r for which the condition holds).

Identity classification with equality tests allowed. Next consider the variant when $\mathcal{C}$ is the identity classification but both kinds of tests are allowed. This is essentially the problem of computing a minimum-cost 2WCST. In this variant, each node u in a tree T has query set $Q_u = [\min Q_u, \max Q_u]_Q \setminus \mathsf{holes}(Q_u)$. Applying recurrence (1) naively to $\mathsf{cost}(Q)$ can yield exponentially many subproblems because $\mathsf{holes}(Q_u)$ can be almost any subset of $[\min Q_u, \max Q_u]_Q$. However, as mentioned in Section 1, it is known that some optimal tree T has the *heaviest-first* property [1, 4]: for each node u in T that does an equality test $\langle = h\rangle$, the test key h is the heaviest key reaching u. (Our tie-breaking scheme makes h unique.) In such a tree there are no light holes. That is, the hole set of any given node u is the set of heavy holes at u:

$$\mathsf{holes}(Q_u) = [\min Q_u, \max Q_u]_K \cap \mathsf{heavier}(k^*(Q_u)).$$

(Note that, by the definition of $k^*(Q_u)$, no keys heavier than $k^*(Q_u)$ reach u, so the set $[\min Q_u, \max Q_u]_K \cap \mathsf{heavier}(k^*(Q_u))$ contains exactly the heavy holes at u.)

A non-empty query set R without light holes is determined by the triple $(\min R, \max R, k^*(R))$, so there are $O(n^3)$ query sets without light holes. This leads naturally to an $O(n^4)$-time algorithm for instances with distinct weights [1, 4]. (Specifically, redefine $\mathsf{cost}(R)$ to be the minimum cost of any *heaviest-first*, irreducible tree for $\pi(R)$. Then $\mathsf{cost}(R) = \infty$ if R has at least one light hole. Add this case as a base case to recurrence (1). Apply the modified recurrence recursively to calculate $\mathsf{cost}(Q)$. Then the number of distinct non-trivial subproblems that arise is $O(n^3)$, and each can be solved in $O(n)$ time.)

Allowing equality tests and an arbitrary classification. The existing results for 2WCSTS don't extend to 2WCDTS because, as shown in Figure 5, there are 2WCDT instances with distinct weights for which no optimal tree is heaviest-first. But, in some sense, the example in Figure 5 is as bad as it gets. There is an optimal tree in which an appropriate relaxation of the heaviest-first property holds, namely, that each node's query set is *admissible*. Roughly, this means that there are at most three light holes, and the light holes must be taken heaviest first from those keys that don't belong to some class $c \in \mathcal{C}$ that contains k^* (the heaviest key reaching the node). Here's the formal definition:

Definition 4 (admissible)**.** *Consider any query subset $R \subseteq Q$. The set R is called* admissible *if it is non-empty and the set of light holes in R is either empty or has the form*

$$\mathsf{heaviest}_b(\,[\min R, \max R]_K \cap \mathsf{lighter}(k^*(R)) \setminus c\,)$$

for some $b \in [3]$ and $c \in \mathcal{C}$ such that $k^*(R) \in c$.

A tree (or subtree) T is called admissible *if the query set of each node in T is admissible.*

Above (and within any mathematical expression), for any integer i, the notation $[i]$ denotes the set $\{1, 2, \ldots, i\}$.

To gain some intuition for the definition, note that, by definition, for any query set R its holes must be in $[\min R, \max R]_K$, and its light holes must be in $\mathsf{lighter}(k^*(R))$.

For the algorithm, roughly, we redefine $\mathsf{cost}(R) = \infty$ if R is not admissible, add a corresponding base case to recurrence (1), and then recursively apply the modified recurrence to compute $\mathsf{cost}(Q)$. Each admissible query set R with no light holes is determined by the triple $(\min R, \max R, k^*(R))$. Per Definition 4, each admissible query set R with at least one light hole is determined by a triple $(\min R, \max R, k^*(R), b, c)$, where $(b, c) \in [3] \times \mathcal{C}$ with $k^*(R) \in c$. It follows that there are $O(n^3 + n^2 m) = O(n^2 m)$ admissible query subsets, so that, in the recursive evaluation of $\mathsf{cost}(Q)$, $O(n^2 m)$ distinct non-trivial subproblems arise. These are solvable in total time $O(n^3 m)$. Section 4 gives the detailed proof.

3 Some optimal tree is admissible

This section proves Theorem 1: if the instance is feasible, then some optimal tree is admissible. Along the way we establish quite a bit more about the structure of optimal trees. We start with some general terminology for how pairs of tests can relate. Recall that $(Q, w, \mathcal{C}, K)$ is a problem instance with at least one correct tree. In any such tree, each edge $u \to v$ from a node to its child corresponds to one of the two possible outcomes of the test at u. We use $u \to v$ to denote both the edge and the associated outcome at u. For example, if u is the node $\langle < 3 \rangle$, and v is the no-child of u, then the outcome $u \to v$ means that the queried value is at least 3.

Definition 5. *Two such outcomes $u \to v$ and $x \to y$ are called* consistent *if Q contains a query value that satisfies them both. Otherwise they are* inconsistent*.*

Two tests are said to be equivalent *if either for all $q \in Q$ the two tests give the same outcome for q, or for all $q \in Q$ the two tests give opposite outcomes for q.*

For example, assume $Q = [4]$. The yes-outcome of $\langle < 3 \rangle$ is inconsistent with the yes-outcome of $\langle = 4 \rangle$ and with the no-outcome of $\langle < 4 \rangle$, but is consistent with both outcomes of $\langle = 2 \rangle$, and with both outcomes of $\langle < 2 \rangle$. The tests $\langle < 4 \rangle$ and $\langle = 4 \rangle$ are equivalent.

Most of the proof requires only the following property of tests:

Property 1 (*laminarity*)**.** *Let u and x be test nodes. If u and x do non-equivalent tests, then, among the four pairs of outcomes between the two nodes, exactly one pair is inconsistent, while the other three pairs are consistent. Formally, let $u \to v$, $u \to v'$, $x \to y$, and $x \to y'$ be the two outcomes from u and the two outcomes from x. Then exactly one pair in $\{u \to v, u \to v'\} \times \{x \to y, x \to y'\}$ is inconsistent.*

If u and x do equivalent tests, each outcome at u is consistent with a distinct outcome at x.

Property 1 is easily verified. (Note that, by the definition of 2WCDTs in Section 2, and assuming there is more than one test, each outcome of each test is satisfied by at least one query in Q.) We call Property 1 *laminarity* because it is equivalent to the laminarity of the collection of sets that has, for each possible test, one set containing the queries that satisfy the test. In our case this laminar collection is

$$\big\{\{q \in Q : q < k\} : k \in K,\, \min Q < k \leq \max Q\big\} \cup \big\{\{q\} : q \in K\big\}.$$

As an example, consider the query set $Q = [4]$. Then the yes-outcome of $\langle < 3\rangle$ and the yes-outcome of $\langle = 4\rangle$ are inconsistent, while every other pair of outcomes is consistent; e.g., the yes-outcome of $\langle < 3\rangle$ and the no-outcome of $\langle = 4\rangle$ are consistent, as they are both satisfied by the query value 2.

Throughout most of the rest of this section (including Sections 3.1 and 3.2), fix T to be an arbitrary irreducible tree.

Property 2. *(i) In T, if u is a proper ancestor of a test node v then the outcome of u leading to v is consistent with both outcomes at v, and the other outcome of u is consistent with exactly one outcome at v. (ii) No two nodes in T are equivalent.*

Property 2 follows quite easily from the irreducibility of T and Property 1: The irreducibility of T implies directly that the outcome of u leading to v is consistent with both outcomes at v. This implies that u and v are not equivalent, and then the second part of (i) then follows from laminarity (Property 1). To justify Property 2(ii), let x and y be two different test nodes in T. We have already established that if one of x, y is an ancestor of the other then they cannot be equivalent. Otherwise, let u be the lowest common ancestor of x and y. By (i), the outcome at u leading to x is consistent with both outcomes at x, but, using (i) and Property 1, it is inconsistent with one outcome at y. So x and y cannot be equivalent.

3.1 Two weight bounds, via splitting

This section introduces *splitting*—a correctness-preserving local rearrangement of the tree that can be viewed as an extension of the well-studied *rotation* operation to a more general class of trees, specifically, to trees whose admissible tests form a laminar set as described above. The section uses splitting to prove two weight bounds (Lemmas 3 and 4) that are used in subsequent sections.

Definition 6. *Let u be a node in T, T_u the subtree of T rooted at u, and x any allowed test (not necessarily in T). The* x-consistent path from u *is the maximal downward path from u in T_u such that each outcome along this path is consistent with both outcomes at x.*

The x-consistent path from u is unique because (by laminarity) at most one outcome out of any given node is consistent with both outcomes at x. In the case that T_u contains a node $\tilde{x}$ that is equivalent to x, the x-consistent path from u is the path from u to $\tilde{x}$ (using here the irreducibility of T and that neither outcome at $\tilde{x}$ is consistent with both outcomes at x). In the case that T_u contains no such node $\tilde{x}$, this x-consistent path from u ends at a leaf.

Fix a node u in T and a test node x, not necessarily in T. Informally, *splitting T_u around x* replaces subtree T_u of T by the subtree T'_x obtained by the following process: initialize T'_x to a subtree with root x, whose yes- and no-subtrees are each a copy of T_u, then splice out each redundant test (that is, each test w such that one of the outcomes at w is inconsistent with the

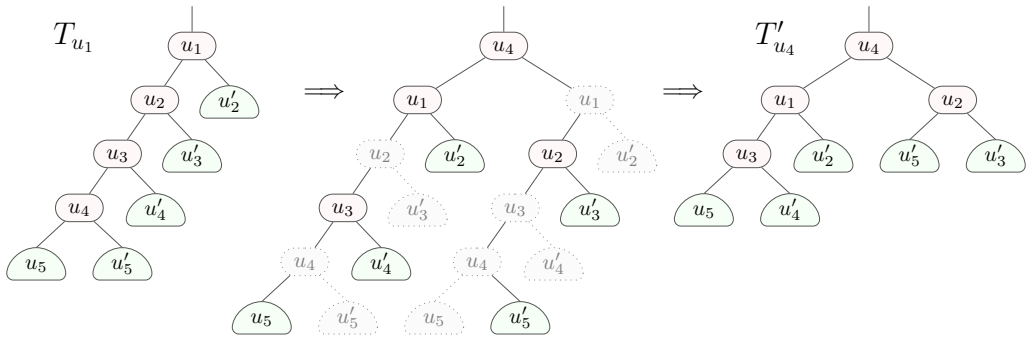


Figure 6: Splitting a subtree T_{u_1} around descendant u_4. The figures in this section draw T_u by drawing u_i and u_i' as the left and right children of their parent u_{i-1}, so that the u_4-consistent path from u_1 is drawn as a prefix of the left spine. Each rounded half-circle represents a subtree, labeled with its root. Here outcomes $u_1 \to u_2'$ and $u_3 \to u_4'$ are consistent with the outcome $u_4 \to u_5$ at u_4 while outcome $u_2 \to u_3'$ is consistent with the other outcome $u_4 \to u_5'$. In the notation of Lemma 3 (taking $j = 4$) $\delta_2 = \delta_3 = 1$ and $\beta = 2$, and the lemma gives the bound $w(u_2') \geq w(u_5) + 2w(u_5')$.

outcome at x that leads to w, so that all queries reaching w must satisfy the other outcome at w). The resulting subtree T'_x has a particular structure that we'll need to use. The formal definition, below, makes this structure explicit.

In this construction, and in the proofs that follow, we will consider and manipulate downward paths in T. For convenience, we adopt the following convention: If $u = u_1 \to u_2 \to \cdots \to u_j$ is any downward path in T then for each $i \in [j-1]$ by u_i' we denote the sibling of u_i, so each edge $u_i \to u_{i+1}'$ leaves this path.

Definition 7 (splitting)**.** Splitting T_u around x *yields the subtree* T'_x *produced by the following process. Let* $u = u_1 \to u_2 \to \cdots \to u_d$ *be the* x*-consistent path from* u*, as defined in Definition 6. Initialize* T'_x *to have root* x*, with yes- and no-subtrees, denoted* T_u^{yes} *and* T_u^{no}*, each a copy of* T_u*.*

For each outcome $\alpha \in \{\mathsf{yes}, \mathsf{no}\}$ *at* x*, modify* T_u^α *within* T'_x *as follows. For each* $i \in [d-1]$*, if outcome* $u_i \to u_{i+1}'$ *is inconsistent with the* α*-outcome at* x*, within* T_u^α*, delete node* u_i *and the subtree* $T_{u_{i+1}'}$*, making* u_{i+1} *the child of the current parent of* u_i *in place of* u_i*. For* $i = d$*, if* u_d *is a leaf, stop. Otherwise (*u_d *is a test node), let* $u_d \to y'$ *be the outcome at* u_d *that is inconsistent with the* α*-outcome at* x*. Within* T_u^α*, delete node* u_d *and the subtree* $T_{y'}$*, making the other child* y *of* u_d *the child of the current parent of* u_d *in place of* u_d*.*

Note that, for each $\alpha \in \{\mathsf{yes}, \mathsf{no}\}$, by the definition of the x-consistent path from u and Property 1 (laminarity), outcome $u_i \to u_{i+1}'$ is inconsistent with exactly one outcome at x. Also, if u_d is a test node then it must be equivalent to x, so exactly one outcome at u_d is inconsistent with the α-outcome at x. (See Lemmas 1 and 2 below for a more detailed characterization of the result of splitting.) Figures 6 and 7 give examples of splitting. In Figure 6, $d = 4$ and u_4 is a test node (in fact $x = u_4$). In Figure 7, x is a new node (not equivalent to any node in T_u), $d = 5$ and u_5 is a leaf.

Lemma 1. *For each query* $q \in Q_u$*, the search for* q *in* T'_x *ends at a leaf that is one of the two copies in* T'_x *of the leaf that the search for* q *in* T_u *ends at.*

Proof. The process that produces T'_x maintains this property as an invariant. The invariant holds initially when the yes- and no-subtrees of x in T'_x are each copies of T_u. Suppose the invariant holds

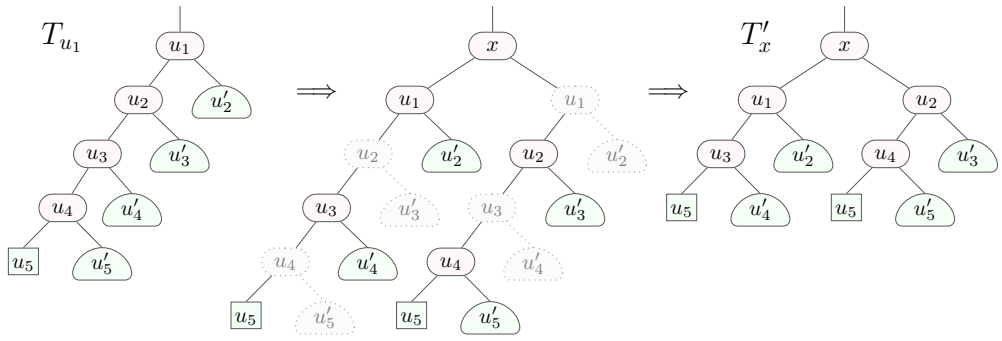


Figure 7: Splitting a subtree T_{u_1} around a new node x (not equivalent to any node in T_u). The x-consistent path from u_1 is $u_1 \to \cdots \to u_5$. Here $u_1 \to u'_2$ and $u_3 \to u'_4$ are consistent with the yes-outcome at x, while $u_2 \to u'_3$ and $u_4 \to u'_5$ are consistent with the no-outcome at x. In the notation of Lemma 4 (taking $j = 4$) $\delta_2 = \delta_3 = 1$ and $\beta' = 2$. The lemma gives the bound $w(u'_2) \geq w(u_4)$.

just before the process deletes a test node v and its subtree $T_{y'}$ from a subtree T_u^α of the current T'_x. The α-outcome at x is inconsistent with the $v \to y'$ outcome at v, and all queries that reach v in the current tree have outcome α at x, and therefore they all satisfy the opposite outcome $v \to y$ at v. So deleting v and $T_{y'}$ (replacing v by y) doesn't change the leaf that any search ends at, thus maintaining the invariant. □

Lemma 1 implies that T'_x is a correct subtree for query set Q_u.

Lemma 2. *(i) For each $i \in [d-1]$, outcome $u_i \to u'_{i+1}$ is inconsistent with exactly one outcome $\alpha \in \{\mathsf{yes}, \mathsf{no}\}$ at x. For this outcome α, node u_i and subtree $T_{u'_{i+1}}$ are deleted from the α-subtree T_u^α of x, and are not deleted from the other subtree $T_u^{\alpha'}$, where outcome α' is the opposite of α.*

(ii) If u_d is a test node, one outcome at u_d, say $u_d \to y$, is inconsistent with the yes-outcome at x, while the other outcome $u_d \to y'$ is inconsistent with the no-outcome at x. Then within T_u^{yes} node u_d and subtree T_y are deleted, while within T_u^{no} node u_d and subtree $T_{y'}$ are deleted.

(iii) For each leaf z in T_u except u_d (if u_d is a leaf), only one of the two copies of z remains in T'_x, and the query set of the remaining copy in T'_x is the same as the query set of z in T_u.

Proof. For $i < d$, by the definition of the x-consistent path from u, each outcome $u_i \to u_{i+1}$ is consistent with both outcomes at x, so, by laminarity, the outcome $u_i \to u'_{i+1}$ is inconsistent with exactly one outcome α at x. Inspecting the construction of T'_x, we obtain that u_i and its subtree $T_{u'_{i+1}}$ are deleted from T_u^α but not from $T_u^{\alpha'}$. This implies Part (i).

Recall that if the final node u_d on the x-consistent path from u is a test node, then by the definition of this path, u_d must be equivalent to x. So one outcome at u_d, say $u_d \to y$, is inconsistent with the yes-outcome at x, while the other outcome $u_d \to y'$ at u_d is inconsistent with the no-outcome at x. This and the definition of T'_x imply Part (ii).

To prove Part (iii), first consider the case that u_d is a test node. Each leaf in T_u is in one of the subtrees $T_{u'_i}$ $(2 \leq i \leq d)$ or in the yes- or no-subtree of T_{u_d}. (Note that all these subtrees are disjoint.) In T'_x, for each of these $d+1$ subtrees, one of the two copies of the subtree is deleted

from T'_x. So only one copy of each leaf remains in T'_x. In the case that u_d is a leaf, for each leaf other than u_d, the same reasoning applies (minus the subtrees of T_{u_d}), to show that only one copy of each leaf other than u_d remains in T'_x. Part (iii) then follows from Lemma 1. □

If u_d is a leaf in T_u, then both copies of u_d remain in T'_x, although one can have an empty query set. (In general, T'_x might not be irreducible, but this does not affect the proofs below.)

Now we prove the weight bounds that are used in later sections. The proofs of these bounds takes advantage of laminarity. Specifically, as T is irreducible, Property 2(i) implies that if u_i is a proper ancestor of u_j then outcome $u_i \to u'_{i+1}$ is consistent with one outcome at u_j and inconsistent with the other.

Lemma 3. *Suppose T is optimal. Let $u_1 \to \cdots \to u_{j+1}$ be any downward path in T. For $1 \le i \le j-1$, let δ_i be the number of ancestors u_s of u_i on the path such that outcomes $u_s \to u'_{s+1}$ and $u_i \to u'_{i+1}$ are consistent with opposite outcomes at u_j. Let β be the number of ancestors u_s of u_{j-1} whose outcome $u_s \to u'_{s+1}$ is consistent with outcome $u_j \to u_{j+1}$ (so $0 \le \beta \le j-1$). Then*

$$w(u'_2) \;\ge\; (j-1-\beta)\, w(u_{j+1}) + \beta w(u'_{j+1}) + \textstyle\sum_{i=3}^{j}(\delta_{i-1}-1)w(u'_i).$$

Proof. Consider splitting subtree T_{u_1} around u_j. Because T is irreducible, both outcomes at u_j are consistent with each outcome along the path $u_1 \to \cdots \to u_j$, so this path is the u_j-consistent path from u_1 used in splitting. By Lemma 2, for each i with $2 \le i \le j$, each descendant of u'_i gains one new ancestor (u_j) and loses δ_{i-1} ancestors, namely those ancestors u_s of u_{i-1} such that outcomes $u_{i-1} \to u'_i$ and $u_s \to u'_{s+1}$ are consistent with opposite outcomes at u_j. Each descendant of u_{j+1} loses $j-1-\beta$ ancestors, namely the ancestors u_s of u_{j-1} whose outcome $u_s \to u'_{s+1}$ is inconsistent with $u_j \to u_{j+1}$. Each descendant of u'_{j+1} loses β ancestors, namely the ancestors u_s of u_{j-1} whose outcome $u_s \to u'_{s+1}$ is inconsistent with $u_j \to u'_{j+1}$. (Here we use that descendants of u_j already had u_j as an ancestor in T_u.) So (using Lemma 2(iii) and that u_j is not a leaf) splitting increases the cost by $-(j-1-\beta)w(u_{j+1}) - \beta w(u'_{j+1}) + \sum_{i=2}^{j}(1-\delta_{i-1})w(u'_i)$. By the optimality of T, this quantity must be non-negative. Substituting $\delta_1 = 0$ and rearranging gives the desired bound. □

Lemma 4. *Suppose T is optimal. Let x be any test node, not necessarily in T. Let $u_1 \to \cdots \to u_{j+1}$ be a prefix of the x-consistent path from u_1. For $1 \le i \le j-1$, let δ_i be the number of ancestors u_s of u_i on the path such that outcomes $u_s \to u'_{s+1}$ and $u_i \to u'_{i+1}$ are consistent with opposite outcomes at u_j. Let β' be the number of ancestors u_s of u_j whose outcome $u_s \to u'_{s+1}$ is consistent with the yes-outcome of x (so $0 \le \beta' \le j$). Then*

$$w(u'_2) \;\ge\; \min(j-1-\beta', \beta'-1)w(u_j) + \textstyle\sum_{i=3}^{j}(\delta_{i-1}-1)w(u'_i).$$

Proof. Consider splitting subtree T_{u_1} around x. By Lemma 2, for each i with $2 \le i \le j$, each descendant of u'_i gains one new ancestor (x) and loses δ_{i-1} ancestors, namely those ancestors u_s such that outcomes $u_{i-1} \to u'_i$ and $u_s \to u'_{s+1}$ are consistent with opposite outcomes at x. Each proper descendant of u_j in the yes-subtree of T'_x gains one new ancestor (x) and loses at least $j-\beta'$ ancestors, namely the ancestors u_s of u_j on the path whose outcome $u_s \to u'_{s+1}$ is inconsistent with the yes-outcome at x. Each proper descendant of u_j in the no-subtree of T'_x gains one new ancestor (x) and loses at least β' ancestors, namely the ancestors u_s of u_j on the path whose outcome $u_s \to u'_{s+1}$ is inconsistent with the no-outcome at x. So the search depth of each proper descendant of u_j increases by at most $\max(1+\beta'-j, 1-\beta')$. So (using Lemmas 1 and 2(iii)) splitting increases the cost by at most $\max(1+\beta'-j, 1-\beta')w(u_j) + \sum_{i=2}^{j}(1-\delta_{i-1})w(u'_i)$. By the optimality of T, this quantity must be non-negative. Substituting $\delta_1 = 0$ and rearranging gives the bound. □

3.2 Structural theorem

This section proves the following theorem. The next section uses it to prove Theorem 1. As in the previous section, for any downward path $u_1 \to u_2 \to \cdots \to u_j$, by u'_i we will denote u_i's sibling $(2 \le i \le j)$.

Theorem 3. *Suppose T is optimal. Let $u_1 \to u_2 \to \cdots \to u_d$ be any downward path in T (not necessarily starting at the root) such that $w(u'_2) < w(u_d)$. Then, for all different nodes u_i, u_j on the path, with $i, j < d$, both outcomes at u_i are consistent with outcome $u_j \to u_{j+1}$.*

Consider the following example for intuition. Suppose that some node u in T does an equality test $\langle = h \rangle$, and, in the no-subtree of u, some node x has $w(x) > w(h)$. By the theorem, then, the query value $q = h$ satisfies all outcomes along the path from the no-child of u to x.

The only property of the admissible tests that Theorem 3 relies on is laminarity.

Proof of Theorem 3. If $i > j$ then the theorem follows directly from Property 2(i). So for the rest of the proof we assume that $i < j < d$, and we only need to prove that outcomes $u_i \to u'_{i+1}$ and $u_j \to u_{j+1}$ are consistent (since we already know that outcomes $u_i \to u_{i+1}$ and $u_j \to u_{j+1}$ are consistent).

Applying Lemma 3 to the path $u_1 \to u_2 \to u_3$ (so $j = 2$) gives $w(u'_2) \ge (1 - \beta)\, w(u_3) + \beta w(u'_3)$, where β is 1 if $u_1 \to u'_2$ is consistent with $u_2 \to u_3$ and zero otherwise. But $w(u'_2) < w(u_d) \le w(u_3)$, so $\beta = 1$. So $w(u'_2) \ge w(u'_3)$ and $u_1 \to u'_2$ is consistent with $u_2 \to u_3$.

With $w(u'_2) < w(u_d)$, this implies $w(u'_3) < w(u_d)$. Applying the theorem inductively to the (shorter) path $u_2 \to \cdots \to u_d$, we have that, for all i and j with $2 \le i < j < d$, $u_i \to u'_{i+1}$ is consistent with $u_j \to u_{j+1}$.

Then the only remaining case is for $i = 1$ and $3 \le j < d$. That is, we need to prove that $u_1 \to u'_2$ is consistent with $u_j \to u_{j+1}$, for all j with $3 \le j < d$. Suppose otherwise for contradiction. Fix j with $3 \le j < d$ such that $u_1 \to u'_2$ is not consistent with $u_j \to u_{j+1}$. Then, by laminarity and the irreducibility of T, $u_1 \to u'_2$ is consistent with $u_j \to u'_{j+1}$. By the previous paragraph $u_2 \to u'_3$ is consistent with $u_j \to u_{j+1}$. So $u_1 \to u'_2$ and $u_2 \to u'_3$ are consistent with different outcomes at u_j.

Apply Lemma 3 to the path $u_1 \to \cdots \to u_{j+1}$. Let δ_i and β be as defined in the lemma. As $u_1 \to u'_2$ and $u_2 \to u'_3$ are consistent with different outcomes at u_j, we have $\delta_i \ge 1$ for all $2 \le i \le j - 1$. Likewise we have $1 \le \beta \le j - 2$, so the bound from the lemma implies $w(u'_2) \ge w(u_{j+1}) + w(u'_{j+1}) = w(u_j) \ge w(u_d)$, contradicting $w(u'_2) < w(u_d)$. □

3.3 Proof of Theorem 1 (some optimal tree is admissible)

The proofs above rely only on laminarity. The proofs below use the particular structure of less-than and equality tests, and the properties of u-consistent paths. In particular, in the special case when u is an equality test, say u is $\langle = h \rangle$, the u-consistent path from x is the path that the search for h would take if started at x.

Lemma 5. *Suppose the instance has distinct weights and T is optimal. Consider any equality-test node $\langle = h \rangle$ and a key k with $w(k) > w(h)$ reaching this node. Then a search for h from the no-child of $\langle = h \rangle$ would end at the leaf L_k for k, and the path from $\langle = h \rangle$ to L_k has at most four nodes (including $\langle = h \rangle$ and L_k). Also, h is not in the class that T assigns to k.*

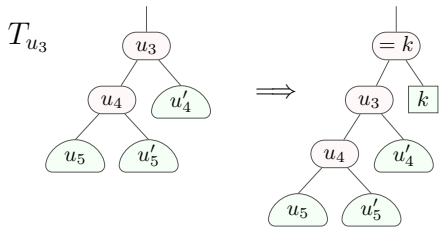


Figure 8: (Lemma 6) Inserting a new node $\langle = k \rangle$ above u_3 to pull k out of Q_{u_5}.

Proof. Let $u_1 \to u_2 \to \cdots \to u_d$ be the path from $\langle = h \rangle$ to L_k. Note that u_1, u_2, and u_d are $\langle = h \rangle$, the no-child of $\langle = h \rangle$, and L_k. We have $w(u_d) = w(L_k) \geq w(k) > w(h) = w(u'_2)$. (Recall that u'_2 denotes the yes-child of u_1.) So, by Theorem 3, we obtain

(∗) For any two different test nodes u_i, u_j along the path with $i, j < d$, both outcomes at u_i are consistent with $u_j \to u_{j+1}$.

Applying this to $i = 1$, we obtain that the $\langle = h \rangle$-consistent path from u_2 ends at L_k. So the yes-outcome of $\langle = h \rangle$ is consistent with all outcomes along this path, and thus a search for h starting from u_2 would end in L_k, as claimed.

To see that h cannot be in the class that T assigns to k, suppose for contradiction that it is. A search for h starting at u_2 would end at L_k, so changing the test key at $\langle = h \rangle$ to k (and relabeling u'_2 with a class containing k) gives a correct tree. The modification decreases the cost by $(w(k) - w(h))(d - 2)$. By assumption $w(k) > w(h)$. By the irreducibility of T, the node $\langle = h \rangle$ cannot be replaced by a leaf, so $d \geq 3$. So the modification gives a correct tree that is cheaper than T, contradicting the optimality of T.

It remains only to show that the length d of the path from $\langle = h \rangle$ to L_k is at most four. The argument uses the following claim:

Claim 1. $2\,w(h) \geq w(u_3)$.

We postpone the proof of Claim 1, and show first how the bound $d \leq 4$ follows from this claim. Assume towards contradiction that $d \geq 5$, and consider the modification to T_{u_1} illustrated in Figure 8. Namely, replace T_{u_3} by a new equality test $\langle = k \rangle$ (for the key k from the lemma) whose yes-child is a new leaf labeled with any answer that k accepts, and whose no-subtree is a copy of T_{u_3}. This increases the search depth of every query reaching u_3, except key k, by 1. It decreases the search depth of k by at least 1. Thus, the increase in cost is at most $(w(u_3) - w(k)) - w(k)$. The optimality of T implies $w(u_3) \geq 2\,w(k) > 2\,w(h)$, contradicting Claim 1. So we must have $d \leq 4$.

To complete the proof of the lemma, it remains to prove Claim 1. The basic idea is to consider splitting T_{u_1} around a suitably choosen test node x, and to apply the bound from Lemma 4 to derive the inequality in Claim 1.

For any two less-than tests along the path, the yes-outcome of the test with smaller key is inconsistent with the no-outcome of the other test, so, using Property (∗), the outcomes on the path must be the no-outcome of the test with smaller key and the yes-outcome of the test with larger key. It follows that the path has at most two less-than tests.

For any equality test, its yes-outcome is inconsistent with one outcome of every test. By this and Property (∗), for each equality test $\langle = k_i \rangle$ on the path, its outcome along the path is the

no-outcome, and the yes-outcome at $\langle = k_i \rangle$ is consistent with every other outcome along the path. That is, the value k_i satisfies every outcome along the path except the no-outcome at $\langle = k_i \rangle$.

Let k_i be the key of test node u_i ($1 \le i \le 4$). Let $k_1^*, k_2^*, k_3^*, k_4^*$ be a permutation of k_1, k_2, k_3, k_4 such that $k_1^* \le k_2^* \le k_3^* \le k_4^*$. Let $u_1^*, u_2^*, u_3^*, u_4^*$ be the corresponding permutation of u_1, u_2, u_3, u_4. Break ties when choosing the permutation so that u_2^* and u_3^* do equality tests and $k_1^* \le k_2^* < k_3^* \le k_4^*$. (This is possible by the conclusions of the previous two paragraphs.) The only possible less-than tests are at u_1^* and at u_4^*: node u_1^* could be $\langle < k_1^* \rangle$ whose outcome along the path is negative, and node u_4^* could be $\langle < k_4^* \rangle$ whose outcome along the path is positive.

Now create a new node $x = \langle < k_3^* \rangle$. For each equality-test on the path, the outcome on the path is the no-outcome, which is consistent with both outcomes at x. By the previous paragraph and using the key ordering, $k_1^* \le k_2^* < k_3^* \le k_4^*$, for each of the two possible less-than tests on the path, its outcome along the path is consistent with both outcomes of x. So both outcomes at x are consistent with all outcomes along the path. Therefore path $u_1 \to \cdots \to u_5$ is a prefix of the x-consistent path from u_1, satisfying the assumptions of Lemma 4 with $j = 4$. The rest of the argument relies on this lemma.

The following observation will be useful: *among the four nodes on $u_1 \to u_2 \to u_3 \to u_4$, two have both outcomes consistent with the yes-outcome at x, while the other two have both outcomes consistent with the no-outcome at x.* (Indeed, by the ordering of the keys and routine inspection, the yes-outcome at x is consistent with both outcomes at u_1^* and with both outcomes at u_2^*. Similarly, the no-outcome at x is consistent with both outcomes at u_3^* and with both outcomes at u_4^*.)

Next, we claim that outcomes $u_1 \to u_2'$ and $u_2 \to u_3'$ are consistent with the same outcome at x. Towards contradiction, suppose that $u_1 \to u_2'$ and $u_2 \to u_3'$ are consistent with opposite outcomes at x, so, in the notation from Lemma 4, $\delta_2 = 1$. The observation above implies that outcomes $u_3 \to u_4'$ and $u_4 \to u_5'$ are also consistent with opposite outcomes at x, so $\delta_3 = 1$. But then (recalling $j = 4$) Lemma 4 gives the bound $w(u_2') \ge w(u_4)$, contradicting $w(u_2') < w(u_d)$, and proving the claim.

Since $u_1 \to u_2'$ and $u_2 \to u_3'$ are consistent with the same outcome at x, the earlier observation implies that $u_3 \to u_4'$ and $u_4 \to u_5'$ are consistent with the other outcome at x. In this case (again in the notation of Lemma 4) $\delta_2 = 0$, $\delta_3 = 2$, and (as before) $\beta' = 2$ and $j = 4$, so the lemma gives the bound $w(u_2') \ge w(u_4) - w(u_3') + w(u_4') = w(u_3) - w(u_3')$. That is, $w(u_2') + w(u_3') \ge w(u_3)$.

It must be that $w(u_2') \ge w(u_3')$. (Otherwise, by Theorem 3 applied to path $u_1 \to u_2 \to u_3'$, the u_1-consistent path from u_2 would include u_3', contradicting that it includes u_3.) With the previous inequality this implies $2\,w(u_2') \ge w(u_3)$. Since $w(u_2') = w(h)$, this completes the proof of Claim 1, and the whole lemma. □

Lemma 6. *If the instance has distinct weights, every irreducible optimal tree is admissible.*

Proof. Let T be any irreducible optimal tree. Consider any node u in T. To prove the lemma we show that u's query set is admissible. If Q_u has no light holes, then we are done, so assume otherwise. Let $k^* = k^*(Q_u)$ be the heaviest key reaching u. Let $H_u = \mathsf{holes}(Q_u) \cap \mathsf{lighter}(k^*)$ be the set of light holes at u and $b = |H_u|$. Let c be the class that T assigns to k^* and $S = [\min Q_u, \max Q_u]_K \cap \mathsf{lighter}(k^*) \setminus c$. We want to show $H_u = \mathsf{heaviest}_b(S)$ and $b \in [3]$.

First we show $H_u \subseteq S$. By definition, $H_u \subseteq [\min Q_u, \max Q_u]_K \cap \mathsf{lighter}(k^*)$. For any light hole $h \in H_u$, key k^* is heavier than h and reaches the ancestor $\langle = h \rangle$ of u. Applying Lemma 5 to that ancestor, hole h is not in c. It follows that $H_u \subseteq S$.

Next we show $H_u = \mathsf{heaviest}_b(S)$. Suppose otherwise for contradiction. That is, there are $k \in S \setminus H_u \subseteq Q_u$ and $h \in H_u$ such that k is heavier than h. Keys k^* and k reach the ancestor $\langle = h \rangle$ of u. Applying Lemma 5 (twice) to that ancestor, the search path for h starting from the no-child of $\langle = h \rangle$ ends both at L_{k^*} and at the leaf L_k for k. So $L_k = L_{k^*}$, which implies that k is in c, contradicting $k \in S$. Therefore $H_u = \mathsf{heaviest}_b(S)$.

Finally, we show that $b \leq 3$. Let $h \in H_u$ be the light hole whose test node $\langle = h \rangle$ is closest to the root. Key k^* reaches $\langle = h \rangle$ and weighs more than h. Applying Lemma 5 to that ancestor, the path from $\langle = h \rangle$ to L_{k^*} has at most four nodes (including the leaf). Each light hole has a unique equality-test node on that path. So there are at most three light holes. □

Finally we prove Theorem 1:

Theorem 1. *If the instance is feasible, then some optimal tree is admissible.*

Proof. We use a perturbation argument to extend Lemma 6. Assume the instance $I = (Q, w, \mathcal{C}, K)$ is feasible (otherwise we are done). Recall that $\tilde{w}(q)$ is the rank of q in the sorting of Q by weight, breaking ties arbitrarily but consistently, as defined at the beginning of Section 2.

Let $I^* = (Q, w^*, \mathcal{C}, K)$ be an instance obtained from I by perturbing the query weights infinitesimally so that (i) the perturbed weights are distinct and (ii) sorting Q by w^* gives the same order as sorting by $\tilde{w}$. (Specifically, take $w^*(q) = w(q) + \epsilon\,\tilde{w}(q)$, for ϵ such that $0 < \epsilon < \delta/n^3$, where $\delta > 0$ is less than the absolute difference in cost between any two irreducible trees with distinct costs, and less than the absolute difference between any two distinct weights.) Note that the sets of valid trees for I and for I^* are the same and finite, and that I^* is a feasible instance with distinct weights.

Let T^* be an optimal, irreducible tree for I^*. Applying Lemma 6 to T^* and I^*, tree T^* is admissible for I^*. By inspection of Definition 1, whether or not a tree is irreducible for I is independent of w. So T^* is also irreducible for I. By inspection of Definition 4, whether or not a tree is admissible for I depends only on the tree and the set of query pairs (h, k) such that $\tilde{w}(h) < \tilde{w}(k)$. This and $\tilde{w}(h) < \tilde{w}(k) \iff \tilde{w}^*(h) < \tilde{w}^*(k)$ imply that T^* is also admissible for I. To finish we observe that T^* is also optimal for I.

Recall that T is an optimal, irreducible tree for I. Letting $\mathsf{cost}(T)$ and $\mathsf{cost}^*(T)$ be the costs of T under weight functions w and w^*, we have $\mathsf{cost}(T^*) \leq \mathsf{cost}^*(T^*) \leq \mathsf{cost}^*(T) \leq \mathsf{cost}(T) + n^3\epsilon < \mathsf{cost}(T) + \delta$. So by the choice of δ we have $\mathsf{cost}(T^*) \leq \mathsf{cost}(T)$. So T^* is optimal for I as well. □

4 Algorithm

This section proves Theorem 2, that the problem admits an $O(n^3 m)$-time algorithm. The input is an arbitrary 2WCDT instance $(Q, w, \mathcal{C}, K)$. In this section, for any $R \subseteq Q$ redefine $\mathsf{cost}(R)$ to be the minimum cost of any *admissible* tree for the subproblem $\pi(R) = (R, w, \mathcal{C}, K)$ obtained by restricting the query set to R. (Take $\mathsf{cost}(R) = \infty$ if there is no admissible tree for $\pi(R)$.) The algorithm returns $\mathsf{cost}(Q)$, the minimum cost of any admissible tree for $(Q, w, \mathcal{C}, K)$. By Theorem 1, this equals the minimum cost of any tree.

The algorithm computes $\mathsf{cost}(Q)$ by using memoized recursion on the following recurrence relation:

Recurrence 1. *For any $R \subseteq Q$,*

$$\mathsf{cost}(R) = \begin{cases} \infty & (R \notin \mathcal{A}) \\ 0 & (R \in \mathcal{A} \wedge (\exists c \in \mathcal{C})\, R \subseteq c) \\ w(R) + \min_u \left(\mathsf{cost}(R_u^{\mathsf{yes}}) + \mathsf{cost}(R_u^{\mathsf{no}})\right), & (\textit{otherwise}) \end{cases}$$

where above $\mathcal{A}$ denotes the set of admissible query subsets of Q (per Definition 4), $(R_u^{\mathsf{yes}}, R_u^{\mathsf{no}})$ is the bipartition of R into those values that satisfy test u and those that don't, and u ranges over the allowed tests such that R_u^{yes} and R_u^{no} are admissible. (If there are no such tests then the minimum is infinite.)

There are $O(n^2 m)$ admissible query sets. (Indeed, each admissible set R with no light holes is determined by the triple $(\min R, \max R, k^*(R))$. Per Definition 4, each admissible set R with light holes is determined by a triple $(\min R, \max R, k^*(R), b, c)$, where $(b, c) \in [3] \times \mathcal{C}$ with $k^*(R) \in c$.) So $O(n^2 m)$ subproblems arise in recursively evaluating $\mathsf{cost}(Q)$. To finish we describe how to evaluate the right-hand side of Recurrence 1 for a given R in $O(n)$ amortized time.

Assume (by renaming elements in Q in a preprocessing step) that $Q = [n]$. Given a non-empty query set $R \subseteq Q$, define the *signature* of R to be

$$\tau(R) = (\min R, \max R, k^*(R), H(R)),$$

where $H(R) = \mathsf{holes}(R) \cap \mathsf{lighter}(k^*(R))$ is the set of light holes in R.

For any R, its signature is easily computable in $O(n)$ time (for example, bucket-sort R and then enumerate the hole set $[\ell, r]_Q \setminus R$ to find $H(R)$). Each signature is in the set

$$\mathcal{S} = Q \times Q \times (K \cup \{\perp\}) \times 2^Q$$

of *potential signatures*. Conversely, given any potential signature $t = (\ell, r, k, H') \in \mathcal{S}$, the set $\tau^{-1}(t)$ with signature t, if any, is unique and computable from t in $O(n)$ time. (Specifically, $\tau^{-1}(t)$ is $Q_t = [\ell, r]_Q \setminus ((K \cap \mathsf{heavier}(k)) \cup H')$ provided Q_t is non-empty and has signature t.)

Lemma 7. *After an $O(n^3 m)$-time preprocessing step, given the signature $\tau(R)$ of any $R \in \mathcal{A}$, the right-hand of Recurrence 1 can be computed in amortized time $O(n)$.*

Proof. Note that the admissible sets can be enumerated in $O(n^3 m)$ time as follows. First do the $O(n^3)$ admissible sets without light holes: for each $(\ell, r, k) \in Q \times Q \times (K \cup \{\perp\})$, output $\tau^{-1}(\ell, r, k, \emptyset)$ if it exists. Next do the $O(n^2 m)$ admissible sets with at least one light hole, following Definition 4: for each $(\ell, r, k, b, c) \in Q \times Q \times K \times [3] \times \mathcal{C}$ with $k \in c$, letting $H' = \mathsf{heaviest}_b([\ell, r]_K \cap \mathsf{lighter}(k) \setminus c)$, if H' is well-defined then output $\tau^{-1}(\ell, r, k, H')$ if it exists.

First we describe the preprocessing step.

Initialize a dictionary A holding a record $A[\tau(R)]$ for each set R in $\mathcal{A}$. To be able to determine whether a given query set R is in $\mathcal{A}$, and to store information (including the memoized cost) for each admissible set R, build a dictionary A that holds a record $A[\tau(R)]$ for each $R \in \mathcal{A}$, indexed by the signature $\tau(R)$. For now, assume the dictionary A supports constant-time access to the record $A[\tau(R)]$ for each $R \in \mathcal{A}$ given the signature $\tau(R)$ of R. (We describe a suitable implementation later.) Initialize A to hold an empty record $A[\tau(R)]$ for each $R \in \mathcal{A}$ by enumerating all $R \in \mathcal{A}$ as described above. This takes $O(n^3 m)$ time.

Identify the leaves. To identify the sets $R \in \mathcal{A}$ that are leaves (that is, such that $(\exists c \in \mathcal{C})\ R \subseteq c$) in $O(n^3 m)$ time, for each triple $(\ell, r, k) \in Q \times Q \times (K \cup \{\perp\})$, do the following steps.

1. Let $\mathcal{R}$ contain the admissible sets R such that $\tau(R) = (\ell, r, k, H')$ for some H'. Assume $\mathcal{R}$ is non-empty (otherwise move on to the next triple). Let $R_\emptyset$ be the set with signature $(\ell, r, k, \emptyset)$, so that $R = R_\emptyset \setminus H(R)$ for $R \in \mathcal{R}$. Let $\mathcal{C}_\ell$ contain the classes $c \in \mathcal{C}$ such that $\ell \in c$. Observe that $|\mathcal{R}| \leq 4|\mathcal{C}_\ell|$, because $R_\emptyset$ is unique for the triple (ℓ, r, k), and then R is determined from $R_\emptyset$ by the class $c \in \mathcal{C}$ and the number $b \in [3]$ of light holes, per Definition 4.

2. Each set $R \in \mathcal{R}$ contains ℓ, so R is a leaf if and only if $R \subseteq c$ for some $c \in \mathcal{C}_\ell$. The condition $R \subseteq c$ is equivalent to $R_\emptyset \setminus H(R) \subseteq c$, which is equivalent to $R_\emptyset \setminus c \subseteq H(R)$. So, any given set $R \in \mathcal{R}$ is a leaf if and only if some subset of $H(R)$ equals $R_\emptyset \setminus c$ for some $c \in \mathcal{C}_\ell$. Identify all such R in time $O(n|\mathcal{R}| + n|\mathcal{C}_\ell|)$. (Recalling that $|H(R)| \leq 3$ for each $R \in \mathcal{R}$, this is straightforward. One way is to construct the collection $\mathcal{H} = \bigcup_{R \in \mathcal{R}} 2^{H(R)}$ of subsets of the light-hole sets. Order the elements within each subset in $\mathcal{H}$ by increasing value, then radix sort $\mathcal{H}$ into lexicographic order. Do the same for the collection $\mathcal{L} = \{R_\emptyset \setminus c : c \in \mathcal{C}_\ell, |R_\emptyset \setminus c| \leq 3\}$. Then merge the two collections to find the elements common to both. A given $R \in \mathcal{R}$ is a leaf if and only if some subset of $H(R)$ in $\mathcal{H}$ also occurs in $\mathcal{L}$.)

As noted above, we have $|\mathcal{R}| \leq 4|\mathcal{C}_\ell|$, so the time spent above on a given triple (ℓ, r, k) is $O(n|\mathcal{C}_\ell|)$. Summing over all triples (ℓ, r, k), the total time is $O(n^2 \sum_{\ell \in K} n|\mathcal{C}_\ell|) = O(n^3 m)$.

In $O(n^3 m)$ time, identify the $O(n^2 m)$ leaves $R \in \mathcal{A}$ as described above. For each, record in its entry $A[\tau(R)]$ that R is a leaf and $\mathsf{cost}(R) = 0$.

Implementing Recurrence 1. Next we describe how to compute $\mathsf{cost}(R)$, given the signature $\tau(R) = (\ell, r, k, H')$ of any set $R \subseteq Q$, in $O(n)$ time.

If A contains no record $A[\tau(R)]$, then R is not admissible, so $\mathsf{cost}(R) = \infty$ and we are done. (Checking this takes constant time, using that if $|H'| \geq 4$ then no lookup is necessary.) So assume A contains a record $A[\tau(R)]$.

If the record $A[\tau(R)]$ already holds a memoized cost for R, then we are done, so assume otherwise. (This implies that R is not a leaf.) In $O(n)$ time, build R from $\tau(R)$ and calculate the sum $w(R)$. Then calculate $\mathsf{cost}(R)$ in $O(n)$ time by evaluating the right-hand side of Recurrence 1, in two stages. In the first stage enumerate all less-than tests that the root u in Recurrence 1 for $\mathsf{cost}(R)$ can be, using the following steps:

1. Using bucket sort, compute $R = (q_1, q_2, \ldots, q_j)$ in sorted order. For $0 \leq i \leq j$ define $R_i = (q_1, q_2, \ldots, q_i)$ and $\overline{R}_i = (q_{i+1}, q_{i+2}, \ldots, q_j)$.
2. Compute $k^*(R_i)$ for $0 \leq i \leq j$ in constant time per i as follows. Start with $k^*(R_0) = \perp$. Then, for $i \leftarrow 1, \ldots, j$, compute $k^*(R_i)$ by using $k^*(R_i) = q_i$ if $q_i \in K$ and q_i is heavier than $k^*(R_{i-1})$, and otherwise $k^*(R_i) = k^*(R_{i-1})$.
3. Compute the light-hole set $H(R_i)$ for $0 \leq i \leq j$ from $H(R)$ in constant time per i, using $H(R_i) = \{h \in H(R) : h \leq q_i\}$ (recall that $|H(R)| \leq 3$).
4. Using the results of Steps 2 and 3, for $0 \leq i \leq j$, in constant time per i compute and store the signature $(q_1, q_i, k^*(R_i), H(R_i))$ of R_i.
5. Similarly, compute and store the signature $(q_{i+1}, q_j, k^*(\overline{R}_i), H(\overline{R}_i))$ of $\overline{R}_i$ for each i.
6. By "merging" R and K (each sorted), identify for each $h \in K$ the i such that $(R^{\mathsf{yes}}_{\langle <h \rangle}, R^{\mathsf{no}}_{\langle <h \rangle}) = (R_i, \overline{R}_i)$, thereby determining $\tau(R^{\mathsf{yes}}_{\langle <h \rangle})$ and $\tau(R^{\mathsf{no}}_{\langle <h \rangle})$. Then there is a term for $u = \langle < h \rangle$ in the recurrence for each h such that the corresponding i satisfies $1 \leq i < j$ (that is, R_i and $\overline{R}_i$ are not empty).

In the second stage, enumerate all the equality-tests $\langle = h\rangle$ for $h \in K \cap R$ that the root u can be. For each such test u, we have $R_u^{\mathsf{yes}} = \{h\}$, so $\tau(R_u^{\mathsf{yes}}) = (h, h, h, \emptyset)$. For all $h \notin \{\min R, \max R, k^*(R)\}$ (using that $|R| \geq 2$, as R is not a leaf, so $R_u^{\mathsf{no}} \neq \emptyset$) use that $\tau(R_u^{\mathsf{no}})$ is $(\min R, \max R, k^*(R), H(R) \cup \{h\})$, which (as $|H(R) \cup \{h\}| \leq 4$) is computable from $\tau(R)$ in constant time. For each of the (at most three) values $h \in \{\min R, \max R, k^*(R)\}$, using $R_u^{\mathsf{no}} = R \setminus \{h\}$, explicitly compute R_u^{no} and its signature in $O(n)$ time. This completes the second stage.

For all tests u considered above, the values of $\mathsf{cost}(R_u^{\mathsf{yes}})$ and $\mathsf{cost}(R_u^{\mathsf{no}})$ are computed recursively from their signatures $\tau(R_u^{\mathsf{yes}})$ and $\tau(R_u^{\mathsf{no}})$.

Finally, for $\mathsf{cost}(R)$, return (and memoize in $A[\tau(R)]$) $w(R)$ plus the minimum of $\mathsf{cost}(R_u^{\mathsf{yes}}) + \mathsf{cost}(R_u^{\mathsf{no}})$, over all such u.

In this way, for each $R \in \mathcal{A}$, the time to evaluate the right-hand side of the recurrence is $O(n)$. There are $O(n^2 m)$ sets in $\mathcal{A}$, so the total time is $O(n^3 m)$. (Note that $\mathsf{cost}(R) = \infty$ may also be computed for $O(n^3 m)$ non-admissible sets $R \notin \mathcal{A}$, but each of these takes constant time.)

How to implement the dictionary A. For each admissible query set $R \in \mathcal{A}$, the set $H(R)$ of light holes has size at most three, so the signature $\tau(R) = (\ell, r, k, H(R))$ has size $O(1)$. So one way to implement the dictionary A (so as to support constant-time lookup) is to use a hash table with universal hashing. Then the algorithm uses space $O(n^2 m)$, but is randomized. If a deterministic implementation is needed, one can implement the dictionary by storing an $n \times n \times n$ matrix T of buckets such that a given bucket $T[\ell, r, k]$ holds the records for the admissible query sets R with signatures of the form $\tau(R) = (\ell, r, k, H')$ for some H'. Organize the records in this bucket using a trie (prefix tree) of depth 3 keyed by the (sorted) keys in H'. This still supports constant-time access, but increases the space to $O(n^3 m)$. More generally, for any $d \geq 1$, one can represent each element $k \in [n]$ within each set H' as a sequence of $\lceil \log_2(n)/d \rceil$ d-bit words, then use a trie with alphabet $\{0, 1, \ldots, 2^d - 1\}$ and depth at most $3\lceil \log_2(n)/d \rceil$. Then space is $\Theta(2^d n^2 m)$ while the access time is $\Theta(\log(n)/d)$. For example, we can take $d = \lceil \epsilon \log_2 n \rceil$ for any constant ϵ to achieve space $O(n^{2+\epsilon} m)$ and access time $\Theta(1/\epsilon) = \Theta(1)$. Or we can take $d = 1$ and achieve space $O(n^2 m)$ and access time $\Theta(\log n)$, increasing the total time to $O(n^3 m \log n)$. □

Remarks. Theorem 2 follows from Lemma 7.

We note without proof that there is a deterministic variant of the algorithm that uses space $O(n^2 m)$ and time $O(n^3 m)$. This variant is more complicated so we chose not to present it.

In the common case that $\mathcal{C}$ partitions Q, so each query $q \in Q$ is contained in just one class $c \in \mathcal{C}$, our algorithm can be implemented in time and space $O(n^2 m) = O(n^3)$. To do this, in the above implementation of the dictionary using a matrix T of buckets, each bucket $T[\ell, r, k]$ stores the records of at most four sets, so no prefix tree is needed to achieve constant access time and space.

Extending the algorithm to other inequality tests. Our model considers decision trees that use less-than and equality tests. Allowing the negations of these tests is a trivial extension. (E.g., every greater-than-or-equal test $\langle \geq k\rangle$ is equivalent by swapping the children to the less-than test $\langle < k\rangle$.) We note without proof that our results also extend easily to the model that allows less-than-or-equal tests (of the form $\langle \leq k\rangle$): the proof of Theorem 3 requires only a minor adjustment—specifically, such tests need to be taken into account when proving Claim 1 in the proof of Lemma 5; the extended algorithm then allows such tests in Recurrence 1.

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] C. Chambers and W. Chen. Efficient multiple and predicated dispatching. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999.*, pages 238–255, 1999.

[3] C. Chambers and W. Chen. Efficient multiple and predicated dispatching. *SIGPLAN Not.*, 34(10):238–255, Oct. 1999.

[4] M. Chrobak, M. Golin, J. I. Munro, and N. E. Young. A simple algorithm for optimal search trees with two-way comparisons. *ACM Trans. Algorithms*, 18(1):2:1–2:11, Dec. 2021.

[5] M. Chrobak, M. Golin, J. I. Munro, and N. E. Young. On Huang and Wong's algorithm for generalized binary split trees. *Acta Informatica*, 59(6):687–708, Dec. 2022.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms.* The MIT Press, Cambridge, Massachusett, fourth edition edition, 2022.

[7] Y. Dagan, Y. Filmus, A. Gabizon, and S. Moran. Twenty (simple) questions. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2017, Montreal, QC, Canada, June 19-23, 2017*, pages 9–21, 2017.

[8] E. Gilbert and E. Moore. Variable-length binary encodings. *Bell System Technical Journal, The*, 38(4):933–967, July 1959.

[9] 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.

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

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

[12] L. Hyafil and R. L. Rivest. Constructing optimal binary decision trees is NP-complete. *Information Processing Letters*, 5(1):15–17, May 1976.

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

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

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

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

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

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