Dispersion on Trees*†

Paweł Gawrychowski[1], Nadav Krasnopolsky[2], Shay Mozes[3], and Oren Weimann[4]

1 **University of Haifa, Israel**
2 **University of Haifa, Israel**
3 **IDC Herzliya, Israel**
4 **University of Haifa, Israel**

Abstract

In the k-dispersion problem, we need to select k nodes of a given graph so as to maximize the minimum distance between any two chosen nodes. This can be seen as a generalization of the independent set problem, where the goal is to select nodes so that the minimum distance is larger than 1. We design an optimal $O(n)$ time algorithm for the dispersion problem on trees consisting of n nodes, thus improving the previous $O(n \log n)$ time solution from 1997.

We also consider the weighted case, where the goal is to choose a set of nodes of total weight at least W. We present an $O(n \log^2 n)$ algorithm improving the previous $O(n \log^4 n)$ solution. Our solution builds on the search version (where we know the minimum distance λ between the chosen nodes) for which we present tight $\Theta(n \log n)$ upper and lower bounds.

1998 ACM Subject Classification F.2.2 Nonnumerical Algorithms and Problems

Keywords and phrases parametric search, dispersion, k-center, dynamic programming

Digital Object Identifier 10.4230/LIPIcs.ESA.2017.40

1 Introduction

Facility location is a family of problems dealing with the placement of facilities on a network in order to optimize certain distances between the facilities, or between facilities and other nodes of the network. Such problems are usually if not always NP-hard on general graphs. There is a rich literature on approximation algorithms (see e.g. [14, 16] and references therein) as well as exact algorithms for restricted inputs. In particular, many linear and near-linear time algorithms were developed for facility location problems on edge-weighted trees.

In the most basic problem, called *k-center*, we are given an edge-weighted tree with n nodes and wish to designate up to k nodes to be facilities, so as to minimize the maximum distance of a node to its closest facility. This problem was studied in the early 80's by Megiddo et al. [12] who gave an $O(n \log^2 n)$ time algorithm that was subsequently improved to $O(n \log n)$ by Frederickson and Johnson [9]. In the early 90's, an optimal $O(n)$ time solution was given by Frederickson [8, 6] using a seminal approach based on parametric search, also for two other versions where points on edges can be designated as facilities or where we minimize over points on edges. In yet another variant, called weighted k-center, every node has a positive weight and we wish to minimize the maximum weighted distance of a node to its closest facility. Megiddo et al. [12] solved this in $O(n \log^2 n)$ time, and Megiddo and

* The research was supported in part by Israel Science Foundation grant 794/13.
† The full version of this paper, containing missing proofs and supplementary figures, is available at `http://arxiv.org/abs/1706.09185`.

Tamir [11] designed an $O(n \log^2 n \log\log n)$ time algorithm when allowing points on edges to be designated as facilities. The latter complexity can be further improved to $O(n \log^2 n)$ using a technique of Cole [5]. A related problem, also suggested in the early 80's [1, 13], is *k-partitioning*. In this problem the nodes have weight and we wish to delete k edges in the tree so as to maximize the weight of the lightest resulting subtree. This problem was also solved by Frederickson in $O(n)$ time [7] using his parametric search framework.

The focus of this paper is the *k-dispersion* problem, where we wish to designate k nodes as facilities so as to maximize the distances among the facilities. In other words, we wish to select k nodes that are as spread-apart as possible. More formally, let $d(u, v)$ denote the distance between nodes u and v, and for a subset of nodes P let $f(P) = \min_{u,v \in P}\{d(u, v)\}$.

- *The Dispersion Optimization Problem.* Given a tree with non-negative edge lengths, and a number k, find a subset P of nodes of size k such that $f(P)$ is maximized.

The dispersion problem can be seen as a generalization of the classical maximum independent set problem (that can be solved by binary searching for the largest value of k for which the minimum distance is at least 2). It can also be seen as a generalization of the diameter problem (i.e., when $k = 2$).

It turns out that the dispersion and the k-partitioning problems are actually equivalent in the one-dimensional case (i.e., when the tree is a path). The reduction simply creates a new path whose edges correspond to nodes in the original path and whose nodes correspond to edges in the original path. However, such equivalence does not apply to general trees, on which k-dispersion seems more difficult than k-partitioning. In particular, until the present work, no linear time solution for k-dispersion was known. The dispersion optimization problem can be solved by repeatedly querying a *feasibility test* that solves the dispersion search problem.

- *The Dispersion Search Problem (feasibility test).* Given a tree with non-negative edge lengths, a number k, and a number λ, find a subset P of nodes of size k such that $f(P) \geq \lambda$, or declare that no such subset exists.

Bhattacharya and Houle [2] presented a linear-time feasibility test, and used a result by Frederickson [9] that enables binary searching over all possible values of λ (i.e., all pairwise distances in the tree). That is, a feasibility test with a running time τ implies an $O(n \log n + \tau \cdot \log n)$ time algorithm for the dispersion optimization problem. Thus, the algorithm of Bhattacharya and Houle for the dispersion optimization problem runs in $O(n \log n)$ time. We present a linear time algorithm for the optimization problem. Our solution is based on a simplified linear-time feasibility test, which we turn into a sublinear-time feasibility test in a technically involved way closely inspired by Frederickson's approach.

In the *weighted* dispersion problem, nodes have non-negative weights. Instead of k we are given W, and the goal is then to find a subset P of nodes of total weight at least W s.t. $f(P)$ is maximized. Bhattacharya and Houle considered this generalization in [3]. They presented an $O(n \log^3 n)$ feasibility test for this generalization, that by the same reasoning above solves the weighted optimization problem in $O(n \log^4 n)$ time. We give an $O(n \log n)$-time feasibility test, and a matching lower bound. Thus, our algorithm for the weighted optimization problem runs in $O(n \log^2 n)$ time. Our solution uses novel ideas, and differs substantially from Frederickson's approach.

Our technique for the unweighted dispersion problem. Our solution to the k-dispersion problem can be seen as a modern adaptation of Frederickson's approach based on a hierarchy of micro-macro decompositions. While achieving this adaptation is technically involved, we

believe this modern view might be of independent interest. As in Frederickson's approach for k-partitioning and k-center, we develop a feasibility test that requires *linear* time preprocessing and can then be queried in *sublinear* time. Equipped with this sublinear feasibility test, it is still not clear how to solve the whole problem in $O(n)$ time, as in such complexity it is not trivial to represent all the pairwise distances in the tree in a structure that enables binary searching. To cope with this, we maintain only a subset of candidate distances and represent them using matrices where both rows and columns are sorted. Running feasibility tests on only a few candidate entries from such matrices allows us to eliminate many other candidates, and prune the tree accordingly. We then repeat the process with the new smaller tree. This is similar to Frederickson's approach, but our algorithm (highlighted below) differs in how we construct these matrices, in how we partition the input tree, and in how we prune it.

Our algorithm begins by partitioning the input tree T into $O(n/b)$ *fragments*, each with $O(b)$ nodes and at most two *boundary nodes* incident to nodes in other fragments: the root of the fragment and, possibly, another boundary node called the *hole*. We use this to simulate a bottom-up feasibility test by jumping over entire fragments, i.e., knowing λ, we wish to extend in $O(\log b)$ time a solution for a subtree of T rooted at the fragment's hole to a subtree of T rooted at the fragment's root. This is achieved by efficient preprocessing: The first step of the preprocessing computes values λ_1 and λ_2 such that (1) there is no solution to the search problem on T for any $\lambda \geq \lambda_2$, (2) there is a solution to the search problem on T for any $\lambda \leq \lambda_1$, and (3) for *most* of the fragments, the distance between any two nodes is either smaller or equal to λ_1 or larger or equal to λ_2. This is achieved by applying Frederickson's parametric search on sorted matrices capturing the pairwise distances between nodes in the same fragment. The (few) fragments that do not satisfy property (3) are handled naively in $O(b)$ time during query time. The fragments that do satisfy property (3) are further preprocessed. We look at the path from the hole to the root of the fragment and run the linear-time feasibility test for all subtrees hanging off from it. Because of property (3), this can be done in advance without knowing the actual exact value of $\lambda \in (\lambda_1, \lambda_2)$, which will only be determined at query time. Let P be a solution produced by the feasibility test to a subtree rooted at a node u. It turns out that the interaction between P and the solution to the entire tree depends only on two nodes of P, which we call the *certain* node and the *candidate* node. We can therefore conceptually replace each hanging subtree by two leafs, and think of the fragment as a caterpillar connecting the root and the hole. After some additional pruning, we can precompute information that will be used to accelerate queries to the feasibility test. During a query we will be able to jump over each fragment of size $O(b)$ in just $O(\log b)$ time, so the test takes $O(\frac{n}{b} \log b)$ time.

The above sublinear-time feasibility test is presented in Section 3, with an overall preprocessing time of $O(n \log\log n)$. The test is then used to solve the optimization problem within the same time. This is done, again, by maintaining an interval $[\lambda_1, \lambda_2)$ and applying Frederickson's parametric search, but now we apply a heavy path decomposition to construct the sorted matrices. To accelerate the $O(n \log\log n)$ time algorithm, we construct a hierarchy of feasibility tests by partitioning the input tree into larger and larger fragments. In each iteration we construct a feasibility test with better running time, until finally, after $\log^* n$ iterations we obtain a feasibility test with $O(\frac{n}{\log^4 n} \cdot \log\log n)$ query-time, which we use to solve the dispersion optimization problem in linear time. It is relatively straightforward to implement the precomputation done in a single iteration in $O(n)$ time. However, achieving total $O(n)$ time over all the iterations, requires reusing the results of the precomputation across iterations as well as an intricate global analysis of the overall complexity. Thus, the details of the linear-time algorithm are technically involved and appear in the full version.

Our technique for the weighted dispersion problem. Our solution for the weighted case differs substantially from Frederickson's approach. In contrast to the unweighted case, where it suffices to consider a single candidate node, in the weighted case each subtree might have a large number of candidate nodes. To overcome this, we represent the candidates of a subtree with a *monotonically decreasing polyline*: for every possible distance d, we store the maximum weight $W(P)$ of a subset of nodes P such that the distance of every node of P to the root of the subtree is at least d. This can be conveniently represented by a sorted list of breakpoints, and the number of breakpoints is at most the size of the subtree. We then show that the polyline of a node can be efficiently computed by merging the polylines of its children. If the polylines are stored in augmented balanced search trees, then two polylines of size x and y can be merged in time $O(\min(x,y)\log\max(x,y))$, and by standard calculation we obtain an $O(n\log^2 n)$ time feasibility test. To improve on that and obtain an optimal $O(n\log n)$ feasibility test, we need to be able to merge polylines in $O(\min(x,y)\log\frac{\max(x,y)}{\min(x,y)})$ time. An old result of Brown and Tarjan [4] is that, in exactly such time we can merge two *2-3 trees* representing two sorted lists of length x and y (and also delete x nodes in a tree of size y). This was later generalized by Huddleston and Mehlhorn [10] to any sequence of operations that exhibits a certain locality of reference. However, in our specific application we need various non-standard batch operations on the lists. any balanced search tree with split and join capabilities. Our data structure both simplifies and extends that of Brown and Tarjan [4], and might be of independent interest.

2 A Linear Time Feasibility Test

Given a tree T with non-negative lengths and a number λ, the feasibility test finds a subset of nodes P such that $f(P)\geq\lambda$ and $|P|$ is maximized, and then checks if $|P|\geq k$. To this end, the tree is processed bottom-up while computing, for every subtree T_r rooted at a node r, a subset of nodes P such that $f(P)\geq\lambda$, $|P|$ is maximized, and in case of a tie $\min_{u\in P} d(r,u)$ is additionally maximized. We call the node $u\in P$, s.t. $d(r,u)<\frac{\lambda}{2}$, the *candidate* node of the subtree (or a candidate with respect to r). There is at most one such candidate node. The remaining nodes in P are called *certain* (with respect to r) and the one that is nearest to the root is called the certain node. When clear from the context, we will not explicitly say which subtree we are referring to.

In each step we are given a node r, its children nodes $r_1, r_2, \ldots, r_\ell$ and, for each child r_i, a maximal valid solution P_i for the feasibility test on T_{r_i} together with the candidate and the certain node. We obtain a maximal valid solution P for the feasibility test on T_r as follows:

1. Take all nodes in $P_1,\ldots,P_\ell$, except for the candidate nodes.
2. Take all candidate nodes u s.t. $d(u,r)\geq\frac{\lambda}{2}$ (i.e., they are certain w.r.t. r).
3. If it exists, take u', the candidate node farthest from r s.t. $d(u',r)<\frac{\lambda}{2}$ and $d(u',x)\geq\lambda$, where x is the closest node to u' we have taken so far.
4. If the distance from r to the closest vertex in P is at least λ, add r to P.

Iterating over the input tree bottom-up as described results in a valid solution P for the whole tree. Finally, we check if $|P|\geq k$.

▶ **Lemma 1.** *The above feasibility test works in linear time and finds P such that $f(P)\geq\lambda$ and $|P|$ is maximized.*

3 An $O(n \log\log n)$ Time Algorithm for the Dispersion Problem

To accelerate the linear-time feasibility test described in Section 2, we will partition the tree into $O(n/b)$ *fragments*, each of size at most b. We will preprocess each fragment to implement the bottom-up feasibility test in sublinear time by "jumping" over fragments in $O(\log b)$ time instead of $O(b)$. The preprocessing takes $O(n \log b)$ time (Section 3.1), and each feasibility test can then be implemented in sublinear $O(\frac{n}{b} \cdot \log b)$ time (Section 3.2). Using heavy-path decomposition, we design an algorithm for the unweighted dispersion optimization problem whose running time is dominated by $O(\log^2 n)$ calls it makes to the sublinear feasibility test (Section 3.3). By setting $b = \log^2 n$ we obtain an $O(n \log\log n)$ time algorithm.

Each fragment is defined by one or two *boundary* nodes: a node u, and possibly a descendant v of u. The fragment whose boundary nodes are u and v consists of the subtree of u without the subtree of v (v does not belong to the fragment). Thus, each fragment is connected to the rest of the tree only through its boundary nodes. We call the path from u to v the fragment's *spine*, and v's subtree its *hole*. If the fragment has only one boundary node, i.e., the fragment consists of a node and all its descendants, we say that there is no hole. A partition of a tree into $O(n/b)$ such fragments, each of size at most b, is called a *good partition*. Note that we can assume that the input tree is binary: given a non-binary tree, we can replace every degree $d \geq 3$ node with a binary tree on d leaves. The edges of the binary tree are all of length zero, so at most one node in the tree can be taken.

▶ **Lemma 2.** *For any binary tree on n nodes and a parameter b, a good partition of the tree can be found in $O(n)$ time.*

3.1 The preprocessing

Recall that the goal in the optimization problem is to find the largest feasible λ^*. Such λ^* is a distance between an unknown pair of vertices in the tree. The first goal of the preprocessing step is to eliminate many possible pairwise distances, so that we can identify a small interval $[\lambda_1, \lambda_2)$ that contains λ^*. We want this interval to be sufficiently small so that for (almost) every fragment F, handling F during the bottom up feasibility test for any value λ in $[\lambda_1, \lambda_2)$ is the same. Observe that the feasibility test in Section 2 for value λ only compares distances to λ and to $\lambda/2$. We therefore call a fragment F *inactive* if for any two nodes $u_1, u_2 \in F$ the following two conditions hold: (1) $d(u_1, u_2) \leq \lambda_1$ or $d(u_1, u_2) \geq \lambda_2$, and (2) $d(u_1, u_2) \leq \frac{\lambda_1}{2}$ or $d(u_1, u_2) \geq \frac{\lambda_2}{2}$. For an inactive fragment F, all the comparisons performed by the feasibility test for any $\lambda \in [\lambda_1, \lambda_2)$ only depend on the interval $[\lambda_1, \lambda_2)$, but not on the particular value of λ. Therefore, once we find an interval $[\lambda_1, \lambda_2)$ for which (almost) all fragments are inactive, we can precompute, for each inactive fragment F, information that will enable us to process F in $O(\log b)$ time during any subsequent feasibility test with $\lambda \in (\lambda_1, \lambda_2)$.

The first goal of the preprocessing step is therefore to find a small enough interval $[\lambda_1, \lambda_2)$. For each fragment F, we construct an implicit representation of $O(b)$ sorted matrices of total side length $O(b \log b)$, s.t. for every two nodes u_1, u_2 in F, $d(u_1, u_2)$ (and also $2d(u_1, u_2)$) is an entry in some matrix. This is done using the standard centroid decomposition, in $O(\frac{n}{b} \cdot b \log b) = O(n \log b)$ total time using the following lemma.

▶ **Lemma 3.** *Given a tree T on b nodes, we can construct in $O(b \log b)$ time an implicit representation of $O(b)$ sorted matrices of total side length $O(b \log b)$ such that, for any $u, v \in T$, $d(u, v)$ is an entry in some matrix.*

Then, we repeatedly choose an entry of a matrix and run a feasibility test with its value. Depending on the outcome, we then appropriately shrink the current interval $[\lambda_1, \lambda_2)$ and

discard this entry. Because the matrices are sorted, running a single feasibility test can actually allow us to discard multiple entries in the same matrix (and, possibly, also entries in some other matrices). The following theorem by Frederickson shows how to exploit this to discard most of the entries with very few feasibility tests.

▶ **Theorem 4** ([7]). *Let $M_1, M_2, ..., M_N$ be a collection of sorted matrices in which matrix M_j is of dimension $m_j \times n_j$, $m_j \leq n_j$, and $\sum_{j=1}^{N} m_j = m$. Let p be nonnegative. The number of feasibility tests needed to discard all but at most p of the elements is $O(\max\{\log(\max_j\{n_j\}), \log(\frac{m}{p+1})\})$, and the total running time exclusive of the feasibility tests is $O(\sum_{j=1}^{N} m_j \cdot \log(2n_j/m_j))$.*

Setting $m = b \log b \cdot \frac{n}{b} = n \log b$ and $p = n/b^2$, the theorem implies that we can use $O(\log b)$ calls to the linear time feasibility test and discard all but n/b^2 elements of the matrices. Therefore, all but at most n/b^2 fragments are inactive.

The second goal of the preprocessing step is to compute information for each inactive fragment that will allow us to later "jump" over it in $O(\log b)$ time when running the feasibility test. We next describe this computation. We choose λ arbitrarily in (λ_1, λ_2). This is done just so that we have a concrete value of λ to work with.

1. **Reduce the fragment to a caterpillar:** a fragment consists of the spine and the subtrees hanging off the spine. We run our linear-time feasibility test on the subtrees hanging off the spine, and obtain the candidate and the certain node for each of them. The fragment can now be reduced to a caterpillar with at most two leaves attached to each spine node: a candidate node and a certain node.
2. **Find candidate nodes that cannot be taken into the solution:** for each candidate node we find its nearest certain node. Then, we compare their distance to λ and remove the candidate node if it cannot be taken. To find the nearest certain node, we first scan all nodes bottom-up (according to the natural order on the spine nodes they are attached to) and compute for each of them the nearest certain node below it. Then, we repeat the scan in the other direction to compute the nearest certain node above. This gives us, for every candidate node, the nearest certain node above and below. We delete all candidate nodes for which one of these distances is smaller than λ. We store the certain node nearest to the root, the certain node nearest to the hole and the total number of certain nodes, and from now on ignore certain nodes and consider only the remaining candidate nodes.
3. **Prune leaves to make their distances to the root non-decreasing:** let the i-th leaf, u_i, be connected with an edge of length y_i to a spine node at distance x_i from the root, and order the leaves so that $x_1 < x_2 < \ldots < x_s$. Note that $y_i < \frac{\lambda}{2}$, as otherwise u_i would be a certain node. Suppose that u_{i-1} is farther from the root than u_i (i.e., $x_{i-1}+y_{i-1} > x_i+y_i$), then: $d(u_i, u_{i-1}) = x_i - x_{i-1} + y_i + y_{i-1} = x_i + y_i - x_{i-1} + y_{i-1} < 2y_{i-1} < \lambda$. Therefore an optimal solution cannot contain both u_i and u_{i-1}. We claim that if the solution contains u_i then it can be replaced with u_{i-1}. To prove this, it is enough to argue that u_{i-1} is farther away from any node above it than u_i, and u_i is closer to any node below it than u_{i-1}. Consider a node u_j that is above u_{i-1} (so $j < i-1$), then: $d(u_j, u_{i-1}) - d(u_j, u_i) = y_{i-1} - (x_i - x_{i-1}) - y_i = x_{i-1} + y_{i-1} - (x_i + y_i) > 0$. Now consider a node u_j that is below u_i (so $j > i$), then: $d(u_j, u_{i-1}) - d(u_j, u_i) = y_{i-1} + (x_i - x_{i-1}) - y_i > 2(x_i - x_{i-1}) > 0$. So in fact, we can remove the i-th leaf from the caterpillar if $x_{i-1} + y_{i-1} > x_i + y_i$. To check this condition efficiently, we scan the caterpillar from top to bottom while maintaining the most recently processed non-removed leaf. This takes linear time in the number of candidate nodes and ensures that the distances of the remaining leaves from the root are non-decreasing.

4. **Prune leaves to make their distances to the hole non-increasing:** this is done as in the previous step, except we scan in the other direction.
5. **Preprocess for any candidate and certain node with respect to the hole:** we call $u_1, u_2, \ldots, u_i$ a *prefix* of the caterpillar and, similarly, $u_{i+1}, u_{i+2}, \ldots, u_s$ a *suffix*. For every possible prefix, we would like to precompute the result of running the linear-time feasibility test on that prefix. In Section 3.2 we will show that, in fact, this is enough to efficiently simulate running the feasibility test on the whole subtree rooted at r if we know the candidate and the certain node w.r.t. the hole. Consider running the feasibility test on $u_1, u_2, \ldots, u_i$. Recall that its goal is to choose as many nodes as possible, and in case of a tie to maximize the distance of the nearest chosen node to r. Due to distances of the leaves to r being non-decreasing, it is clear that u_i should be chosen. Then, consider the largest $i' < i$ such that $d(u_{i'}, u_i) \geq \lambda$. Due to distances of the leaves to the hole being non-decreasing, nodes $u_{i'+1}, u_{i'+2}, \ldots, u_{i-1}$ cannot be chosen and furthermore $d(u_j, u_i) \geq \lambda$ for any $j = 1, 2, \ldots, i'$. Therefore, to continue the simulation we should repeat the reasoning for $u_1, u_2, \ldots, u_{i'}$. This suggests the following implementation: scan the caterpillar from top to bottom and store, for every prefix $u_1, u_2, \ldots, u_i$, the number of chosen nodes, the certain node and the candidate node. While scanning we maintain i' in amortized constant time. After increasing i, we only have to keep increasing i' as long as $d(u_i, u_{i'}) \geq \lambda$. To store the information for the current prefix, copy the computed information for $u_1, u_2, \ldots, u_{i'}$ and increase the number of chosen nodes by one. Then, if the certain node is set to NULL, we set it to be u_i. If there is no $u_{i'}$, and u_i is the top-most chosen candidate, we need to set it to be the candidate (if $d(r, u_i) < \frac{\lambda}{2}$) or the certain node otherwise.

3.2 The feasibility test

The sublinear feasibility test for a value $\lambda \in (\lambda_1, \lambda_2)$ processes the tree bottom-up. For every fragment with root r, we would like to simulate running the linear-time feasibility test on the subtree rooted at r to compute: the number of chosen nodes, the candidate node, and the certain node. We assume that we already have such information for the fragment rooted at the hole of the current fragment. If the current fragment is active, we process it naively in $O(b)$ time using the linear-time feasibility test. If it is inactive, we process it (jump over it) in $O(\log b)$ time. This can be seen as, roughly speaking, attaching the hole as another spine node to the corresponding caterpillar and executing steps (2)-(5).

We start by considering the case where there is no candidate node w.r.t. the hole. Let v be the certain node w.r.t. the hole. Because distances of the leaves from the hole are non-increasing, we can compute the prefix of the caterpillar consisting of leaves that can be chosen, by binary searching for the largest i such that $d(v, u_i) \geq \lambda$. Then, we retrieve and return the result stored for $u_1, u_2, \ldots, u_i$ (after increasing the number of chosen nodes and, if the certain node is set to NULL, updating it to v).

Now consider the case where there is a candidate node u w.r.t. the hole. We start with binary searching for i as explained above. Then, we check if the distance between u and the certain node nearest to the hole is smaller than λ or $d(u_i, r) > d(u, r)$, and if so return the result stored for $u_1, u_2, \ldots, u_i$. Then, again because distances of the leaves to the hole are non-increasing, we can binary search for the largest $i' \leq i$ such that $d(u_{i'}, u) \geq \lambda$ (note that this also takes care of pruning leaves u_k that are closer to the hole than u). Finally, we retrieve and return the result stored for $u_1, u_2, \ldots, u_{i'}$ (after increasing the number of chosen nodes and possibly updating the candidate and the certain node).

We process every inactive fragment in $O(\log b)$ time and every active fragment in $O(b)$ time, so the total time is $O(\frac{n}{b} \cdot \log b) + O(\frac{n}{b^2} \cdot b) = O(\frac{n}{b} \cdot \log b)$.

3.3 The algorithm for the optimization problem

The general idea is to use a heavy path decomposition to solve the optimization problem with $O(\log^2 n)$ feasibility tests. The *heavy edge* of a non-leaf node of the tree is the edge leading to the child with the largest number of descendants. The heavy edges define a decomposition of the nodes into heavy paths. A heavy path p starts with a head $\mathsf{head}(p)$ and ends with a tail $\mathsf{tail}(p)$ such that $\mathsf{tail}(p)$ is a descendant of $\mathsf{head}(p)$, and its depth is the number of heavy paths p' s.t. $\mathsf{head}(p')$ is an ancestor of $\mathsf{head}(p)$. The depth is always $O(\log n)$ [15].

We process all heavy paths at the same depth together while maintaining an interval $[\lambda_1, \lambda_2)$ such that λ_1 is feasible while λ_2 is not, that is, the sought λ^* belongs to the interval. The goal of processing the heavy paths at depth d is to further shrink the interval so that, for any heavy path p at depth d, the result of running the feasibility test on any subtree rooted at $\mathsf{head}(p)$ is the same for any $\lambda \in [\lambda_1, \lambda_2)$ and therefore can be already determined. We start with the heavy paths of maximal depth and terminate with $\lambda^* = \lambda_1$ after having determined the result of running the feasibility test on the whole tree.

Let n_d denote the total size of all heavy paths at depth d. For every such heavy path we construct a caterpillar by replacing any subtree that hangs off by the certain and the candidate node (this is possible, because we have already determined the result of running the feasibility test on that subtree). To account for the possibility of including a node of the heavy path in the solution, we attach an artificial leaf connected with a zero-length edge to every such node. The caterpillar is then pruned similarly to steps (2)-(4) from Section 3.1, except that after having found the nearest certain node for every candidate node we cannot simply compare their distance to λ. Instead, we create an 1×1 matrix storing the relevant distance for every candidate node. Then, we apply Theorem 4 with $p = 0$ to the obtained set of $O(n_d)$ matrices of dimension 1×1. This allows us to determine, using only $O(\log n)$ feasibility tests and $O(n_d)$ time exclusive of the feasibility tests, which distances are larger than λ^*, so that we can prune the caterpillars and work only with the remaining candidate nodes. Then, for every caterpillar we create a row- and column-sorted matrix storing pairwise distance between its leaves. By applying Theorem 4 with $p = 0$ on the obtained set of square matrices of total side length $O(n_d)$ we can determine, with $O(\log n)$ feasibility tests and $O(n_d)$ time exclusive of the feasibility tests, which distances are larger than λ^*. This allows us to run the bottom-up procedure described in Section 2 to produce the candidate and the certain node for every subtree rooted at $\mathsf{head}(p)$, where p is a heavy path at depth d.

All in all, for every d we spend $O(n_d)$ time and execute $O(\log n)$ feasibility tests. Summing over all depths d, this is $O(n)$ plus $O(\log^2 n)$ calls to the feasibility test. Setting $b = \log^2 n$, the total time is thus $O(n + n \log\log n + \frac{n}{\log^2 n} \cdot \log\log n \cdot \log^2 n) = O(n \log\log n)$.

4 The Weighted Dispersion Problem

In this section we present an $O(n \log n)$ time algorithm for the weighted search problem (a matching lower bound is shown in the full version). As explained in the introduction, this then implies an $O(n \log^2 n)$ time solution for the optimization problem. Similarly to the unweighted case, we compute for each node of the tree, the subset of nodes P in its subtree s.t. $f(P) \geq \lambda$ and the total weight of P is maximized. We compute this by going over the nodes of the tree bottom-up. Previously, the situation was simpler, as for any subtree we had just one candidate node (i.e., a node that may or may not be in the optimal solution

for the entire input tree). This was true because nodes had uniform weights. Now however, there could be many candidates in a subtree, as the certain nodes are only the ones that are at distance at least λ from the root (and not $\frac{\lambda}{2}$ as in the unweighted case).

Let P be a subset of the nodes in the subtree rooted at v, and h be the node in P minimizing $d(h, v)$. We call h the *closest chosen node* in v's subtree. In our feasibility test, v stores an optimal solution P for each possible value of $d(h, v)$ (up to λ, otherwise the closest chosen node does not affect nodes outside the subtree). That is, a subset of nodes P in v's subtree, of maximal weight, s.t. the closest chosen node is at distance *at least* $d(h, v)$ from v, $f(P) \geq \lambda$. This can be viewed as a monotone polyline, since the weight of P (denoted $W(P)$) only decreases as the distance of the closest chosen node increases (from 0 to λ). $W(P)$ changes only at certain points called *breakpoints* of the polyline. Each point of the polyline is a key-value pair, where the key is $d(h, v)$ and the value is $W(P)$. We store with each breakpoint the value of the polyline between it and the next breakpoint, i.e., for a pair of consecutive breakpoints with keys a and $a + b$, the polyline value of the interval $(a, a + b]$ is associated with the former. The representation of a polyline consists of its breakpoints, and the value of the polyline at key 0.

The algorithm computes such a polyline for the subtrees rooted at every node v of the tree by merging the polylines computed for the subtrees rooted at v's children. We assume w.l.o.g. that the input tree is binary (for the same reasoning as in the unweighted case), and show how to implement this step in time $O(x \log(\frac{2y}{x}))$, where x is the number of breakpoints in the polyline with fewer breakpoints, and y is the number of breakpoints in the other.

Constructing a polyline. We now present a single step of the algorithm. We postpone the discussion of the data structure used to store the polylines for now, and first describe how to obtain the polyline of v from the polylines of its children. Then, we state the exact interface of the data structure that allows executing such a procedure efficiently, show how to implement such an interface, and finally analyze the complexity of the resulting algorithm.

If v has only one child, u, we build v's polyline by querying u's polyline for the case that v is in the solution (i.e., query u's polyline with distance of the closest chosen node being $\lambda - d(v, u)$), and add to this value the weight of v itself. We then construct the polyline by taking the obtained value for $d(h, v) = 0$ and merging it with the polyline computed for u, shifted to the right by $d(v, u)$ (since we now measure the weight of the solution as a function of the distance of the closest chosen node to v, not to u). The value between zero and $d(v, u)$ will be the same as the value of the first interval in the polyline constructed for u, so the shift is actually done by increasing the keys of all but the first breakpoint by $d(v, u)$.

If v has a left child u_1 and a right child u_2, we have two polylines p_1 and p_2 (that represent the solutions inside the subtrees rooted at u_1 and u_2), and we want to create the polyline p for the subtree rooted at v. Denote the number of breakpoints in p_1 by x and the number of breakpoints in p_2 by y. Assume w.l.o.g. that $x \leq y$. We begin with computing the value of p for key zero (i.e. v is in the solution). In this case we query p_1 and p_2 for their values with keys $\lambda - d(v, u_1)$ and $\lambda - d(v, u_2)$ respectively (if one of these is negative, we take zero instead), and add them together with the weight of v. Note that it is possible for the optimal solution in v's subtree not to include v. Therefore we need to check, after constructing the rest of the polyline, whether the value stored at the first breakpoint (which is the weight of the optimal solution where v is not included) is greater than the value we computed for the case v is chosen. If so, we store the value of the first breakpoint also as the value for key zero.

It remains to construct the rest of the polyline p. Notice that we need to maintain that $d(h_1, h_2) \geq \lambda$ (where h_1 is the closest chosen node in u_1's subtree and h_2 is the closest chosen

node in u_2's subtree). We start by shifting p_1 and p_2 to the right by $d(v, u_1)$ and $d(v, u_2)$ respectively, because now we measure the distance of h from v, not from u_1 or u_2. We then proceed in two steps, each computing half of the polyline p.

4.1 Constructing the second half of the polyline.

We start by constructing the second half of the polyline, where $d(h, v) \geq \frac{\lambda}{2}$. In this case we query both polylines with the same key, since $d(h_1, v) \geq \frac{\lambda}{2}$ and $d(h_2, v) \geq \frac{\lambda}{2}$ implies that $d(h_1, h_2) \geq \lambda$. The naive way to proceed would be to iterate over the second half of both polylines in parallel, and at every point sum the values of the two polylines. This would not be efficient enough, and so we only iterate over the breakpoints in the second half of p_1 (the smaller polyline). These breakpoints induce intervals of p_2. For each of these intervals we increase the value of p_2 by the value in the interval in p_1. This might require inserting some of the breakpoints from p_1, where there is no such breakpoint already in p_2. Thus, we obtain the second half of p by modifying the second half of p_2.

4.2 Constructing the first half of the polyline.

We need to consider two possible cases: either $d(h_1, v) < d(h_2, v)$ (i.e. the closest chosen node in v's subtree is inside u_1's subtree), or $d(h_1, v) > d(h_2, v)$ (h is in u_2's subtree). Note that in this half of the polyline $d(h, v) < \frac{\lambda}{2}$, and therefore $d(h_1, v) \neq d(h_2, v)$. For each of the two cases we will construct the first half of the polyline, and then we take the maximum of the two resulting polylines at every point, in order to have the optimal solution for each key.

Case I: $d(h_1, v) < d(h_2, v)$. Since we are only interested in the first half of the polyline, we know that $d(h_1, v) < \frac{\lambda}{2}$. Since $d(h_2, v) + d(h_1, v) \geq \lambda$ we have that $d(h_2, v) > \frac{\lambda}{2}$. Again, we cannot afford to iterate over the breakpoints of p_2, so we need to be more subtle.

We start by splitting p_1 at $\frac{\lambda}{2}$ and taking the first half (denoted by p_1'). We then split p_2 at $\frac{\lambda}{2}$ and take the second half (denoted by p_2'). Consider two consecutive breakpoints of p_1' with keys x and $x + y$. We would like to increase the value of p_1' in the interval $(x, x + y]$ s.t. the new value is the maximal weight of a valid subset of nodes from *both* subtrees rooted at u_1 and u_2, s.t. $x < d(h_1, v) \leq x + y$. Therefore $d(h_2, v) \geq \lambda - x - y$. p_2' is monotonically decreasing, and so we query it at $\lambda - x - y$, and increase by the resulting value.

This process might result in a polyline which is not monotonically decreasing, because as we go over the intervals of p_1' from left to right we increase the values there more and more. To complete the construction, we make the polyline monotonically decreasing by scanning it from $\frac{\lambda}{2}$ to zero and deleting unnecessary breakpoints. We can afford to do this, since the number of breakpoints in this polyline is no larger than the number of breakpoints in p_1. Note that we have assumed we have access to the original data structure representing p_2, but this structure has been modified to obtain the second half of p. However, we started with computing the second half of p only to make the description simpler. We can simply start with the first half.

Case II: $d(h_1, v) > d(h_2, v)$. Symmetrically to the previous case, we increase the values in the intervals of p_2 induced by the breakpoints of p_1 by the appropriate values of p_1 (similarly to what we do in Subsection 4.1). Again, the resulting polyline may be non-monotone, but this time we cannot solve the problem by scanning the new polyline and deleting breakpoints, since there are too many of them. Instead, we go over the breakpoints of the second half of p_1. For each such breakpoint with key k, we check if the new polyline has a breakpoint

with key $\lambda - k$. If so, denote its value by w, otherwise continue to the next breakpoint of p_1. These are the points where we might have increased the value of p_2. We then query the new polyline with a *value predecessor* query: this returns the breakpoint with the largest key s.t. its key is smaller than $\lambda - k$ and its value is at least w. If this breakpoint exists, and it is not the predecessor of the breakpoint at $\lambda - k$, then the values of the new polyline between its successor breakpoint and $\lambda - k$ should all be w (i.e. we delete all breakpoints in this interval and set the successor's value to w). If it does not exist, then the values between zero and $\lambda - k$ should be w (i.e. we delete all the previous breakpoints). This ensures that the resulting polyline is monotonically decreasing.

Merging cases I and II. We now need to build one polyline for the first half of the polyline, taking into account both cases. Let p_a and p_b denote the polylines we have constructed in cases I and II respectively (so the number of breakpoint in p_a is at most x, the number of breakpoints in p_b is at most y, and $x \leq y$).

We now need to take the maximum of the values of p_a and p_b, for each key. We do this by finding the intersection points of the two polylines. Notice that since both polylines are monotonically decreasing, these intersections can only occur at (i) the breakpoints of p_a, and (ii) at most one point between two consecutive breakpoints of p_a.

We iterate over p_a and for each breakpoint, we check if the value of p_b for the same key is between the values of this breakpoint and the predecessor breakpoint in p_a. If so, this is an intersection point. Then, we find the intersection points which are between breakpoints of p_a, by running a value predecessor query on p_b for every breakpoint in p_a except for the first. After such computation, we know which polyline gives us the best solution for every point between zero and $\frac{\lambda}{2}$, and where are the intersection points where this changes. We can now build the new polyline by doing insertions and deletions in p_b according to the intersection points: For every interval of p_b defined by a pair of consecutive intersection points, we check if the value of p_a is larger than the value of p_b in the interval, and if so, delete all the breakpoints of p_b in the interval, and insert the relevant breakpoints from p_a. The number of intersection points is linear in the number of breakpoints of p_a, and so the total number of interval deletions and insertions is $O(x)$.

To conclude, the final polyline p is obtained by concatenating the value computed for key zero, the polyline computed for the first half, and the polyline computed for the second half.

4.3 The polyline data structure

We now specify the data structure for storing the polylines. The required interface is:

1. Split the polyline at some key.
2. Merge two polylines (s.t. all keys in one polyline are smaller than all keys in the other).
3. Retrieve the value of the polyline for a certain key $d(h, v)$.
4. Return a sorted list of the breakpoints of the polyline.
5. Batched interval increase – Given a list of disjoint intervals of the polyline, and a number for each interval, increase the values of the polyline in each interval by the appropriate number. Each interval is given by the keys of its endpoints.
6. Batched value predecessor – Given a list of key-value pairs, (k_i, v_i), find for each k_i, the maximal key k'_i, s.t. $k'_i < k_i$ and the value of the polyline at k'_i is at least v_i, assuming that the intervals (k'_i, k_i) are disjoint.
7. Batched interval insertions – Given a list of pairs of consecutive breakpoints in the polyline, insert between each pair a list of breakpoints.

8. Batched interval deletions – Given a list of disjoint intervals of the polyline, delete all the breakpoints inside the intervals.

We now describe the data structure implementing the above interface. We represent a polyline by storing its breakpoints in an augmented 2-3 tree, where the data is stored in the leaves. Each node stores a key-value pair, and we maintain the following property: the key of each breakpoint is the sum of the keys of the corresponding leaf and of all its ancestors, and similarly for the values. In addition, we store in each node the maximal sum of keys and values on a path from that node to a leaf in its subtree. We also store in each node the number of leaves in its subtree. Operations 1 and 2 use standard split and join procedures for 2-3 trees in logarithmic time. Operation 3 runs a predecessor query and returns the value at the returned breakpoint in logarithmic time. Operation 4 is done by an inorder traversal of the tree (of p_1 in $O(x)$ time). Operations 1-4 are performed only a constant number of times per step, and so their total cost is $O(\log x + \log y + x)$. The next four operations are more costly, since they consists a batch of $O(x)$ operations given in sorted order (by keys).

Operation 5 – batched interval increase. Consider the following implementation for Operation 5. We iterate over the intervals, and for each of them, we find its left endpoint, and traverse the path from the left endpoint, through the LCA, to the right endpoint. The traversal is guided by the maximal key stored in the current node (that are used to find the maximal key of a breakpoint stored in its subtree by adding the sum of all keys from the root to the current node, which is maintained in constant time after moving to a child or the parent). While traversing the path from the left endpoint to the LCA (from the LCA to the right endpoint), we increase the value of every node hanging to the right (left) of this path. We also update the maximal value field in each node we reach (including the nodes on the path from the LCA to the root). Notice that if one of the endpoints of the interval is not in the structure, we need to insert it. We might also need to delete a breakpoint if it is a starting point of some interval and its new value is now equal to the value of its predecessor. This implementation would take time which is linear in the number of traversed nodes, plus the cost of insertions and deletions (whose number is linear in the number of intervals). Because the depth of a 2-3 tree of size $O(y)$ is $O(\log y)$, this comes up to $O(x \log y)$. Such time complexity for each step would imply $O(n \log^2 n)$ total time for the feasibility test.

We improve the running time by performing the operations on smaller trees. The operation therefore begins by splitting the tree into $O(x)$ smaller trees, each with $O(\frac{y}{x})$ leaves. This is done by recursively splitting the tree, first into two trees with $O(\frac{y}{2})$ leaves, then we split each of these trees into two trees with $O(\frac{y}{4})$ leaves, and so on, until we have trees of size $O(\frac{y}{x})$. We then increase the values in the relevant intervals using the small trees. For this, we scan the roots of the small trees, searching for the left endpoint of the first interval (by using the maximal key stored in the root of each tree). Once we have found the left endpoint of the interval, we check if the right endpoint of the interval is in the same tree or not (again, using the maximal key). In the first case, the interval is contained in a single tree, and can be increased in this tree in time $O(\log(\frac{2y}{x}))$ using the procedure we have previously described. In the second case, the interval spans several trees, and so we need to do an interval increase in the two trees containing the endpoints of the interval, and additionally increase the value stored in the root of every tree that is entirely contained in the interval. We then continue to the next interval, and proceed in the same manner. Since the intervals are disjoint and we do at most two interval increases on small trees per interval, the total time for the increases in the small trees is $O(x \cdot \log(\frac{2y}{x}))$. Scanning the roots of the small trees adds $O(x)$ to the complexity, leading to $O(x \cdot \log(\frac{2y}{x}) + x) = O(x \log(\frac{2y}{x}))$ overall for processing the small trees.

Before the operation terminates, we need to join the small trees to form one large tree. This is symmetric to splitting and analyzed with the same calculation.

▶ **Lemma 5.** *The time to obtain the small trees is* $O(x \log(\frac{2y}{x}))$.

The cost of all joins required to patch the small trees together can be bounded by the same calculation as the cost of the splits made to obtain them, and so the operation takes $O(x \log(\frac{2y}{x}))$ time in total. The rest of the batched operations are also done by splitting the tree into small trees. There is an additional technical difficulty in Operation 6, as in our case the intervals $(k_{i'}, k_i)$ might not be disjoint. We make them disjoint with some extra work. In Operation 7, some of the small trees might become much larger due to the insertions. This also requires some extra work, see the full version for a complete description.

▶ **Theorem 6.** *The above implementation implies an* $O(n \log n)$ *weighted feasibility test.*

References

1 R.I. Becker, S.R. Schach, and Y. Perl. A shifting algorithm for min-max tree partitioning. *J. ACM*, 29(1):56–67, 1982.

2 B.K. Bhattacharya and M.E. Houle. Generalized maximum independent sets for trees. In *CATS*, pages 17–25, 1997.

3 B.K. Bhattacharya and M.E. Houle. Generalized maximum independent sets for trees in subquadratic time. In *ISAAC*, pages 435–445, 1999.

4 M.R. Brown and R.E. Tarjan. Design and analysis of a data structure for representing sorted lists. *SIAM Journal on Computing*, 9(3):594–614, 1980.

5 Richard Cole. Slowing down sorting networks to obtain faster sorting algorithms. *J. ACM*, 34(1):200–208, 1987.

6 G.N. Frederickson. Optimal algorithms for partitioning trees and locating p-centers in trees. Technical Report CSD-TR-1029, Purdue University, 1990.

7 G.N. Frederickson. Optimal algorithms for tree partitioning. In *SODA*, pages 168–177, 1991.

8 G.N. Frederickson. Parametric search and locating supply centers in trees. In *WADS*, pages 299–319, 1991.

9 G.N. Frederickson and D.B. Johnson. Finding k-th paths and p-centers by generating and searching good data structures. *J. Algorithms*, 4(1):61–80, 1983.

10 Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Inf.*, 17:157–184, 1982.

11 N. Megiddo and A. Tamir. New results on the complexity of p-center problems. *SIAM J. Computing*, 12(3):751–758, 1983.

12 N. Megiddo, A. Tamir, E. Zemel, and R. Chandrasekaran. An $O(n \log^2 n)$ algorithm for the k-th longest path in a tree with applications to location problems. *SIAM J. Computing*, 10(2):328–337, 1981.

13 Y. Perl and S.R. Schach. Max-min tree partitioning. *J. ACM*, 28(1):5–15, 1981.

14 D.B. Shmoys, É. Tardos, and K. Aardal. Approximation algorithms for facility location problems. In *STOC*, pages 265–274, 1997.

15 D.D. Sleator and R.E. Tarjan. A data structure for dynamic trees. *Journal of Computer and System Sciences*, 26(3):362–391, 1983.

16 V.V. Vazirani. *Approximation Algorithms*. Springer, 2003.