Optimal Parametric Search Algorithms in Trees I: Tree Partitioning

Greg N. Frederickson*

Department of Computer Science
Purdue University
West Lafayette, Indiana 47907

gnf@cs.purdue.edu

April 15, 2013

Abstract. Linear-time algorithms are presented for partitioning a tree with weights on the vertices by removing k edges so as to either minimize the maximum weight component or maximize the minimum-weight component.

Key words and phrases. Data structures, max-min, min-max, p-center, parametric search, partial order, sorted matrices, tree partitioning.

*This research was supported in part by the National Science Foundation under grants CCR-86202271 and CCR-9001241, and by the Office of Naval Research under contract N00014-86-K-0689.

1. Introduction

Parametric search is a powerful technique for solving various optimization problems [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]. To solve a given problem by parametric search, one must find a largest (or smallest) value that passes a certain feasibility test. In general, a set of candidate values is explicitly or implicitly identified, and the set is searched, choosing one value at a time upon which to test feasibility. As a result of the test, various values in the set will be discarded from further consideration. Eventually the search will converge on the optimal value.

In [**?**], Megiddo has emphasized the role that parallel algorithms can play in the implementation of such a technique. In [**?**], Cole improves the time bounds for a number of problems addressed in [**?**], abstracting two techniques upon which parametric search can be based: sorting and performing independent binary searches. Despite the clever ideas elaborated in these papers, none of the algorithms presented in [**?**] and [**?**] is known to be optimal. In fact, rarely have algorithms for parametric search problems been shown to be optimal. (For one that is optimal, see [**?**].) In at least some cases the time required for feasibility testing matches known lower bounds for the parametric search problem. But in worst case $\Omega(\log n)$ values must be tested for feasibility. Is this extra factor of at least $\log n$ necessary? For several parametric search problems on trees, we show that a polylogarithmic penalty in the running time can be avoided, and give linear-time (and hence optimal) algorithms for these problems.

We first consider the max-min tree k-partitioning problem [**?**]. Let T be a tree with n vertices and a nonnegative weight associated with each vertex. Let $k < n$ be a positive integer. The problem is to delete k edges in the tree so as to maximize the weight of the lightest of the resulting subtrees. Perl and Schach introduced an algorithm that runs in $O(k^2 rd(T) + kn)$ time, where $rd(T)$ is the radius of the tree [**?**].

Megiddo and Cole first considered the special case in which the tree is a path, and give $O(n(\log n)^2)$-time and $O(n \log n)$-time algorithms, resp., for this case.[1] (Megiddo notes that an $O(n \log n)$-time algorithm is possible, using ideas from [**?**].) For the problem on a tree, Megiddo presents an $O(n(\log n)^3)$-time algorithm [**?**], and Cole presents an $O(n(\log n)^2)$-time algorithm [**?**]. The algorithm we present here runs in $O(n)$ time, which is clearly optimal.

The second problem is the min-max tree k-partitioning problem [**?**]. The problem here is to delete k edges in the tree so as to minimize the weight of the heaviest of the resulting subtrees. This is closely related to the above problem. Becker, Perl and Schach give an algorithm that runs in $O(k^3 rd(T) + kn)$ time [**?**], and Perl and Vishkin give an efficient implementation of this algorithm that brings the time down to $O(krd(T)(k+\log \Delta)+n)$, where Δ is the maximum degree in the tree [**?**]. Megiddo's and Cole's techniques apply to this problem with the bounds quoted above. Again, our techniques yield linear-time algorithms.

In [**?**], Becker and Perl prove that the algorithms of [**?**],[**?**] can be applied to tree partitioning problems that are subject to any one of a number of objective functions from a certain class. We suspect that our techniques can be applied to most (if not all) of the objective functions identified in [**?**]. In [**?**], a somewhat different algorithm is given for the tree partitioning problem in which the maximum diameter is minimized, with a running time of $O(k^2 rd(T)n)$. We note that this problem is equivalent to the p-center problem $E/V/k+1$, which we discuss in [**?**].

Our techniques can be extended and applied to the problem of finding a p-center in a tree [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**], [**?**]. In [**?**] and [**?**] we describe the extensions to our technique that yield $O(n)$-time algorithms for three variations of the p-center problem in trees. Also included in [**?**] is a linear-time algorithm for the problem of

[1] All logarithms are to the base 2.

computing recovery points in trees [**?**], [**?**].

Our results are a contribution to parametric search in several ways beyond merely producing a number of optimal algorithms. First, our results demonstrate that parallel algorithms are not essential or even that helpful in designing optimal algorithms for certain parametric search problems. We choose to ignore the link with parallel computation, and instead emphasize the generation and searching of partial orders. Instead of the two searching paradigms discussed in [**?**], we focus on a third paradigm: searching within a collection of sorted matrices [**?**], [**?**]. Thus our work refocuses the emphasis on a carefully managed search. Second, we show how to manage the parametric search to produce intermediate information on which to base faster feasibility tests. To construct optimal algorithms for our problems, it appears to be necessary to interleave the narrowing of bounds on the optimal value with the construction of better feasibility tests. This is similar in spirit to the method in [**?**], though the reasons that the feasibility tests can be sped up are quite different and the driving search procedure is quite different. Third, we design a pruning strategy for handling problems on trees, so that the feasibility test is not thwarted by trees that are exceptionally bushy.

Our paper is organized as follows. In section 2 we discuss some features of parametric search, and lay the foundation for the optimal algorithms that appear in subsequent sections. In section 3 we present our approach for solving the max-min partitioning problem on a path. In section 4 we build on the results in section 3 and present our approach for solving the max-min partitioning problem on a tree.

A preliminary version of this paper appeared in [**?**].

2. Paradigms for parametric search

In this section we discuss some features of parametric search, and lay the foun-

dation for the optimal algorithms that appear in subsequent sections. We discuss path and tree partitioning problems in this section, note that the methods can be applied also to the p-center problem. We first review straightforward feasibility tests to test the feasibility of a search value in a tree. We then review how to represent all possible search values of a path within a sorted matrix. We next present our approach for searching using values from a collection of sorted matrices. Finally, we describe straightforward approaches for the path and the tree that are as good as any algorithms in [**?**] or [**?**].

We first describe straightforward feasibility tests for cutting edges in a tree so as to either maximize the minimum size of any resulting component (max-min problem), or minimize the maximum size of any resulting component (min-max problem). Our faster feasibility tests will be built on these. For the max-min problem, the feasibility test takes a test value λ, and determines if (at least) k cuts can be placed in the tree such that no component has weight less than λ. For the max-min problem, let λ^* be the largest value of λ that passes the test. For the min-max problem, the feasibility test takes a test value λ, and determines if at most k cuts can be placed in the tree such that no component has weight greater than λ. For the min-max problem, let λ^* be the smallest value of λ that passes the test.

A straightforward test *FTEST0* for the max-min problem in the tree is the following [**?**]. Root the tree at a vertex of degree 1, and initialize *numcut* to -1. Explore the tree starting at the root. At vertex v, set $wgt(v)$ to the weight of v, and for each child w of v first explore the subtree rooted at w, and then add the resulting $wgt(w)$ to $wgt(v)$. After all children of v are explored, if $wgt(v) \geq \lambda$, then add 1 to *numcut* and reset $wgt(v)$ to 0. Whenever *numcut* is incremented, except for the last time, the edge from v to the parent of v should be cut. When the traversal is complete, if $numcut \geq k$, then λ is feasible ($\lambda \leq \lambda^*$), else λ is not feasible ($\lambda > \lambda^*$). The

feasibility test uses $O(n)$ time.

The feasibility test for the min-max problem is similar [**?**]. Root the tree at a vertex of degree 1, and initialize *numcut* to 0. Explore the tree starting at the root. At vertex v, set $wgt(v)$ to the weight of v, and explore the subtree rooted at each child w of v. Determine the child w' with largest $wgt(\cdot)$ value such that $wgt(v)+\sum_{wgt(w)\leq wgt(w')} wgt(w) \leq \lambda$, using lexicography to break ties in $wgt(\cdot)$ values. For each child w, if $wgt(w) \leq wgt(w')$, then add $wgt(w)$ to $wgt(v)$, else cut the edge from w to v and add 1 to *numcut*. Using a standard median-finding algorithm [**?**], the appropriate child w' can be found in time linear in the number of children. When the traversal is complete, if $numcut \leq k$, then λ is feasible ($\lambda \geq \lambda^*$), else λ is not feasible ($\lambda < \lambda^*$). Thus this feasibility test also uses $O(n)$ time.

In a parametric search problem, the search can be restricted to considering values from a finite set of values. The desired value is the largest (in the case of a max-min problem) or the smallest (in the case of a min-max problem) that passes the feasibility test. Let each value in the finite set be called a *candidate value*. We next discuss a data structure that contains all candidate values for problems on a path. This structure is based on ideas in [**?**] and [**?**]. Let a matrix be called a *sorted matrix* if for every row the values are in nonincreasing order, and for every column the values are in nonincreasing order. Let the vertices on the path P be indexed from 1 to n. The set of values to be searched is the set of sums of weights of vertices from i to j, for all pairs $i \leq j$. The data structure contains these values, plus others, in the form of a sorted matrix that is succinctly represented and generated as follows. For $i = 0, 1, \cdots, n$, let A_i be the sum of the weights of vertices 1 to i. Note that for any pair i, j with $i < j$, the sum of the weights from vertex $i+1$ to j is $A_j - A_i$. Let X_1 be the sequence of sums $A_n, A_{n-1}, \cdots, A_1$, and let X_2 be the sequence of sums $A_0, A_1, \cdots, A_{n-1}$. Then sorted matrix $M(P)$ is the $n \times n$ Cartesian matrix $X_1 - X_2$, where the ij-th entry is

$A_{n+1-i} - A_{j-1}$. In determining the above, we can use proper subtraction, i.e., $a - b$ gives $\max\{a - b, 0\}$. Clearly, the values in any row of $M(P)$ are in nonincreasing order, and the values in any column of $M(P)$ are also in nonincreasing order. Note that $M(P)$ can be represented in $O(n)$ space in such a way that any matrix entry can be computed in constant time.

As an example, we show a vertex-weighted path P in Fig. 2.1, and its associated matrix $M(P)$. The sequence X_1 is listed vertically to the left of $M(P)$, and the sequence X_2 is listed horizontally above $M(P)$, in such a way that the ij-th element of $M(P)$ is to the right of the i-th element of X_1 and beneath the j-th element of X_2.

We next describe a searching algorithm *MSEARCH* that combines the methods of [**?**] and [**?**], and adapts them for our purposes. Algorithm *MSEARCH* takes as its arguments a collection of sorted matrices, searching bounds λ_1 and λ_2, and a stopping count. For a max-min problem, λ_1 is feasible and λ_2 is not. (The reverse holds for a min-max problem.) *MSEARCH* will produce a sequence of values one at a time to be tested for feasibility, with elements in the collection of matrices discarded as a result of each test. For a max-min problem, if $\lambda > \lambda_1$ is feasible, then λ_1 is reset to λ. Otherwise, if $\lambda < \lambda_2$ is not feasible, then λ_2 is reset to λ. (The reverse is done for a min-max problem.) *MSEARCH* will halt when the number of matrix elements remaining is no greater than the stopping count. The algorithm consists of a sequence of iterations. On each iteration the largest matrices are split into smaller submatrices, and three test values are produced in turn. On the basis of each test, certain of the submatrices are discarded. The stopping count can prevent the use of many iterations to discard relatively few elements by halting the search before all candidate values have been discarded. We assume that the product of the dimensions of each sorted matrix is a power of 4. If this is not the case, then the matrix can be padded out logically with big values. (Including these big values enlarges only the set of values

to be tested, and does not change the underlying path.) Let the *size* of a matrix or a submatrix be the product of its dimensions.

We now specify *MSEARCH* for a max-min problem. Let a *cell* be a matrix or submatrix contained in a set $\mathcal{C}$.

proc *MSEARCH* (**set_of_matrices** $\mathcal{M}$, **integer** *stop_count*, **real** λ_1, λ_2)
$\mathcal{C} \leftarrow \emptyset$
Let *cell_size* equal the size of the largest matrix in $\mathcal{M}$.
repeat
    Move from $\mathcal{M}$ to $\mathcal{C}$ all matrices whose size equals *cell_size*.
    if *cell_size* > 1 **then**
        for each cell C in $\mathcal{C}$ **do**
            if both dimensions of C are greater than 1
            then Split C into four cells by halving each dimension.
            else Split C into four cells by quartering the larger dimension.
            endif
            Replace C by the resulting four cells.
        endfor
        cell_size $\leftarrow$ *cell_size*$/4$
    endif
    repeat three times
        if *cell_size* $= 1$
        then Let R be the multiset of values in the cells.
        else Let R be the multiset consisting of the smallest and the largest
                element from each cell in $\mathcal{C}$.
        endif
        Select the median element x in R.
        if $\lambda_1 < x < \lambda_2$
        then
            Use the designated feasibility test on element x.
            if x is feasible **then** $\lambda_1 \leftarrow x$ **else** $\lambda_2 \leftarrow x$ **endif**
            Discard from $\mathcal{C}$ any cell with no values in (λ_1, λ_2).
        endif
    endrepeat
until the total size of all matrices and cells in $\mathcal{M} \cup \mathcal{C}$ is at most *stop_count*
endproc

For the min-max problem *MSEARCH* is the same, except that λ_2 is reset to x if x is feasible, and λ_1 is reset to x if x is not feasible.

The following theorem is similar in spirit to Lemma 5 in [**?**] and Theorem 2 in [**?**], and its proof shares similarities with their proofs.

Theorem 2.1. Let $\mathcal{M}$ be a collection of N sorted matrices $\{M_1, M_2, \cdots, M_N\}$ 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 q be the stopping count. The number of iterations needed by *MSEARCH* is $O(\max\{\log \max_j\{n_j\}, \log(m/(q+1))\})$, and the total time of *MSEARCH* exclusive of feasibility tests is $O(\sum_{j=1}^{N} m_j \log(2n_j/m_j))$.

Proof. Let M_j be a matrix in $\mathcal{M}$. Let i' be the last iteration in which cells are divided. Consider any iteration $i \leq i'$, and let c be the current cell size at the end of the iteration. If M_j is still in $\mathcal{M}$ on the i-th iteration, let $b_{ij} = 0$. Otherwise, if $c > n_j/m_j$, then let $b_{ij} = 2\sqrt{n_j m_j/c} - 1$, else let $b_{ij} = m_j$. For any given value x, at most b_{ij} of the cells of M_j on iteration i will have smallest element less than x and largest element greater than x. Let $B_i = \sum_{j=1}^{N} b_{ij}$.

We claim that for all iterations in which cells are divided, the number of cells remaining at the end of the i-th iteration is no greater than $2B_i + 2$. The proof of the claim is by induction on i. Let $ncells_i$ be the number of cells in $\mathcal{C}$ after cells have been quartered on iteration i, and let $ncells'_i$ be the number of cells in $\mathcal{C}$ at the end of iteration i. Since $B_0 = 0$ and $ncells'_0 = 0$, the claim holds before the first iteration. For any iteration i, $0 < i \leq i'$, there are, by the induction hypothesis, $ncells'_{i-1} \leq 2B_{i-1} + 2$ cells before the iteration begins. Suppose that K_i cells are transferred from $\mathcal{M}$ to $\mathcal{C}$ in iteration i. Then $ncells_i \leq 4(K_i + 2B_{i-1} + 2)$. Note that $b_{ij} \geq b_{i-1,j}$ for any M_j not in $\mathcal{M}$ on iteration i. (Equality holds if M_j was not in $\mathcal{M}$ on iteration $i-1$ and the cells resulting from M_j have one dimension equal to 1 on iteration $i-1$. Note that $b_{ij} = b_{i-1,j} + 1$ for any matrix M_j whose cells have both dimensions greater than 1 on iteration $i-1$ but have one dimension equal to 1 on iteration i.) Thus $B_{i-1} + K_i \leq B_i$, and thus $ncells_i \leq 8B_i + 8$.

When the median is selected and cells are discarded, the number of cells discarded will be half of the following quantity: the current number of cells minus B_i. This follows since at most B_i cells have smallest element less than x and largest element greater than x, and thus x will induce a partition of the remaining cells into two equal size sets. The number of cells discarded on each of the three executions of the inner loop will be $d_1 \geq \lfloor (ncells_i - B_i)/2 \rfloor$, $d_2 \geq \lfloor (ncells_i - d_1 - B_i)/2 \rfloor$, and $d_3 \geq \lfloor (ncells_i - d_1 - d_2 - B_i)/2 \rfloor$, resp. Then $ncells'_i = ncells_i - d_1 - d_2 - d_3 \leq (ncells_i + 7B_i + 7)/8 \leq (8B_i + 8 + 7B_i + 7)/8 = (15/8)(B_i + 1) < 2B_i + 2$. This concludes the proof of the claim.

Since the number of cells remaining at the end of the i-th iteration is no greater than $2B_i + 2$, the time to divide and select among cells on the i-th iteration is $O(B_i)$, which is $O(\sum_{j=1}^{N} b_{ij})$. It follows from the definition of b_{ij} that $\sum_{i=1}^{i'} b_{ij}$ is $O(m_j \log(2n_j/m_j))$. Thus the time to perform all iterations $i \leq i'$, exclusive of feasibility tests, is $O(\sum_{j=1}^{N} m_j \log(2n_j/m_j))$. For each iteration $i \geq i'$, the cells selected among contain single elements, so that the number of elements is halved on each iteration. Thus all iterations $i > i'$ will use $O(m)$ time, exclusive of feasibility testing. This is dominated by the time to perform all iterations $i \leq i'$.

Since the current cell size is quartered on each iteration $i \leq i'$, the number of such iterations is $O(\log \max_j\{n_j\})$. Since the number of elements is halved for each iteration $i > i'$, the number of such iterations is $O(\log(m/(q+1)))$. □

We are now ready to use *MSEARCH* to generate (relatively) simple algorithms for partitioning a path and partitioning a tree. These algorithms match the time of the algorithms in [**?**], and set the stage for the improved algorithms that we present in the next two sections. We first describe a simple approach to the max-min problem on a path P. Initialize λ_1 to 0 and λ_2 to ∞. (For path and tree partitioning, we may take ∞ to be the total weight of all vertices in the path or tree.) Run algorithm

MSEARCH on the set containing the single sorted matrix $M(P)$ with stopping count 0. Use the straightforward feasibility test described above. When *MSEARCH* halts, $\lambda^* = \lambda_1$. (For the min-max problem, $\lambda^* = \lambda_2$.) By Theorem 2.1, *MSEARCH* will take $O(n)$ total time, exclusive of the feasibility tests, and will produce a sequence of $O(\log n)$ values to be tested. It follows that the total time for all feasibility tests is $O(n \log n)$. This corresponds to the times achieved by Megiddo and Cole for the path problem. We show how to do better in section 3.

We illustrate the above procedure (and also *MSEARCH*) using path P and in Fig. 2.1, with $k = 3$. We set λ_1 to 0 and λ_2 to ∞ and call *MSEARCH* on $M(P)$. Matrix $M(P)$ is first quartered into 4 cells of dimension 4×4. The set R formed from these cells is $\{59, 3, 28, 0, 13, 0, 0, 0\}$. The median of the set is 1.5 (the average of the lower median 0 and the upper median 3). For $\lambda = 1.5$, 6 cuts are required, so λ_1 is reset to 1.5, and the cell with all values less than or equal to 1.5 is discarded, leaving three cells and a new set $R = \{59, 3, 28, 0, 13, 0\}$. The median of R is $(3 + 13)/2 = 8$. For $\lambda = 8$, 3 cuts are required, so λ_1 is reset to 8. But there are no more cells with all values at most 8, so none gets discarded. The third selection produces the same result. On the next iteration of the outer repeat loop, the three cells are quartered into 12 cells, as shown in Fig. 2.2(a). Set $R = \{59, 45, 44, 23, 28, 20, 17, 0, 42, 25, 27, 3, 11, 0, 0, 0, 31, 22, 16, 0, 15, 0, 0, 0\}$. The median of this set is $(16+17)/2 = 16.5$. For $\lambda = 16.5$, 1 cut is required, so λ_2 is reset to 16.5. There are five cells with all values at least 16.5, and these are discarded. The remaining cells have $R = \{17, 0, 27, 3, 11, 0, 0, 0, 16, 0, 15, 0, 0, 0\}$. The median of this set is $0 \leq \lambda_1$, so that the two cells with all values no larger than 0 are discarded. The remaining cells have $R = \{17, 0, 27, 3, 11, 0, 16, 0, 15, 0\}$. The median of this set is $(3+11)/2 = 7 \leq \lambda_1$. No cells can be discarded, leaving the cells as pictured in Fig. 2.2(b). On the next iteration of the repeat loop, these cells are quartered, giving 20 cells of dimen-

sion 1×1. Then $R = \{17, 6, 11, 0, 27, 12, 18, 3, 11, 9, 2, 0, 16, 1, 15, 0, 15, 7, 8, 0\}$. The median is $(8 + 9)/2 = 8.5$. For $\lambda = 8.5$, 3 cuts are required, so λ_1 is reset to 8.5. Discarding the half of the values and selecting the median gives 15. For $\lambda = 15$, 2 cuts are required, so λ_2 is reset to 15. Discarding all but $\{9, 11, 11, 12\}$ and selecting the median gives 11. For $\lambda = 11$, 3 cuts are required, so λ_1 is reset to 11. All values but 12 are discarded. On the next iteration of the outer repeat-loop, this value is tested and found to require 3 cuts, so λ_1 is reset to 12. With all values at this point discarded, *MSEARCH* will terminate with $\lambda^* = \lambda_1 = 12$.

We next describe a simple approach to the max-min problem on a tree. We first define an *edge-path-partition* of a tree rooted at a vertex of degree 1. Partition the edges of the tree into paths, where a vertex is an endpoint of a path if and only if it is of degree not equal to 2 with respect to the tree. Call any path in an edge-path-partition that contains a leaf in the tree a *leaf-path*. As an example, consider the vertex-weighted tree in Fig. 2.3(a). The edge-path-partition for T is shown in Fig. 2.3(b). There are 7 paths in the partition, as shown. Four of these paths are leaf-paths.

We now proceed with the approach for the tree. Initialize λ_1 to 0 and λ_2 to ∞. Initialize T to be the tree rooted at a vertex of degree 1. Repeat the following. Form an edge-path-partition of T. For each leaf-path P_j in the edge-path-partition, form sorted matrix $M(P_j)$. Run *MSEARCH* on this collection of matrices, with stopping count 0. We use the straightforward feasibility test described above. If T contains just one leaf, halt with $\lambda^* = \lambda_1$. (For the min-max problem, $\lambda^* = \lambda_2$.) If T contains more than one leaf, do the following. For each leaf-path P_j, infer the cuts in P_j such that each component in turn going up in P_j, except the highest component on P_j, has total weight as small as possible but greater than λ_1. (For the min-max problem, infer the cuts on P_j such that each component in turn going up P_j has total weight

as large as possible but less than λ_2.) Delete all vertices beneath the top vertex of each leaf-path P_j, and add to the weight of the top vertex in P_j the weight of the other vertices in the highest component of P_j. This leaves a smaller tree in which all leaf-paths in the original tree have been deleted. The smaller tree has at most half of the number of leaves of T. Reset T to be the smaller tree, and k to be the number of cuts remaining to be made in this tree. This concludes the description of the repeat loop, and with it the algorithm.

As an example we consider the max-min problem on the tree shown in Fig. 2.3(a), with $k = 3$. After a call to *MSEARCH* given the matrices corresponding to the leaf-paths, $\lambda_1 = 9$ and $\lambda_2 = 15$. One cut can be placed between the vertices of weight 15 and 6. The weights of the two leaves of weight 4 are added to the weight of their parent, giving it weight 13. The weights of all descendants of the vertex with weight 2, except the weight of 15, are added to the vertex of weight 2, giving it a weight also of 13. Then all edges in the leaf-paths are deleted, and k is reset to 2. The edge-path-partition of the resulting tree is shown in Fig. 2.4(a). There are two leaf-paths in this partition, with vertex weights 13, 4, and 3 on one, and 13 and 3 on the other. After a call to *MSEARCH* given the matrices corresponding to these leaf-paths, $\lambda_1 = 9$ and $\lambda_2 = 13$. Two cuts can be placed, above each of the vertices of weight 13. The weight of the vertex of weight 4 is then added to its parent, giving it weight 7. Then all edges in the leaf-paths are deleted, and k is reset to 0. The edge-path-partition of the resulting tree is shown in Fig. 2.4(b). After a call to *MSEARCH* given the matrices corresponding to the leaf-paths, $\lambda_1 = 12$ and $\lambda_2 = 13$. Thus $\lambda^* = 12$.

For correctness of the tree algorithm, note that λ_1 always corresponds to a feasible value, and that the cuts are inferred on leaf-paths assuming (correctly) that any subsequent value λ to be tested will have $\lambda > \lambda_1$. Also note that we choose the edge-path-partition to be a partition of edges rather than vertices, so that the inference of

cuts will be correct in the min-max version of the problem too.

We analyze the time as follows. By Theorem 2.1, each call to *MSEARCH* will produce $O(\log n)$ values, and the time to produce these, exclusive of feasibility testing, will be $O(\sum_{j=1}^{N} n_j) = O(n)$. Since each feasibility test takes $O(n)$ time, the test values generated by running *MSEARCH* can be tested in a total of $O(n \log n)$ time. Since the number of leaves is at least halved on each such iteration, the number of iterations needed until all cuts have been made is $O(\log n)$. Thus the total time to partition the tree by this method is $O(n(\log n)^2)$. This beats the time of $O(n(\log n)^3)$ for Megiddo's algorithm, and matches the time of $O(n(\log n)^2)$ for Cole's algorithm. We show how to do better in section 4.

3. Partitioning a Path

In this section we present three algorithms to perform parametric search on a path of n vertices. Each algorithm is an improvement on the previous, and we present them one at a time to assist the reader in understanding our approach. The first algorithm uses two phases of search to achieve $O(n \log \log n)$ time. The first phase gathers information with which subsequent feasibility tests can be performed in $o(n)$ time. The second phase then completes the parametric search using this faster feasibility testing. The second algorithm gathers information on more than one phase, and takes $O(n \log^* n)$ time.[2] The third algorithm uses a variety of refinements to reduce the time down to $O(n)$. Our discussion focuses on the max-min problem; at the end of the section we identify the changes necessary for the min-max problem. Throughout this section we assume that the vertices of any path or subpath are indexed in increasing order from the start to the end of the path.

We describe several key ideas for fast feasibility testing in a path. We partition

[2]The function $\log^* k$ is the iterated logarithm of k, defined by $\log^* 1 = \log^* 2 = 1$ and $\log^* k = 1 + \log^* \lceil \log k \rceil$ for $k > 2$.

the vertices of the path P into subpaths as follows. For a given integer r, define an *r-partition* of a path P as a partition of the vertices of P into subpaths P_j, where each subpath except the last contains r vertices, and the last contains at most r vertices. For each subpath P_j, let f_j and t_j be the indices of the first and last vertices, resp., in P_j. Parametric search is used to generate a data structure that allows us to determine the number of cuts in most subpaths quickly. Let λ_1 and λ_2 be the parametric search bounds on λ^*. Define an *active value* to be any candidate value λ that arises from a subpath and satisfies $\lambda_1 < \lambda < \lambda_2$. For each vertex v_l on any subpath P_j that has no active values, two quantities $ncut(l)$ and $rmdr(l)$ can be determined. The quantity $ncut(l)$ is the maximum number of cuts on the subpath from v_l to t_j such that each connected component in this subpath, with the exception of the last, has weight at least λ_2. (Since we know that λ_1 is feasible, we set up $ncut(\cdot)$ to test values $\lambda > \lambda_1$ for feasibility. Since all candidate values arising from P_j have been discarded, the next value on P_j larger than λ_1 is at least λ_2.) For convenience of description, we assume that a cut may be placed after t_j, producing a component of weight 0. (We compensate for this extra cut by subtracting 1 from the number of cuts.) The quantity $rmdr(l)$ is the maximum weight possible for the last component, given that the number of edges cut in the subpath from v_l to t_j is $ncut(l)$. Given the $ncut(\cdot)$ and $rmdr(\cdot)$ values, each feasibility test *FTEST1* in the second phase runs in $o(n)$ time.

We now describe algorithm *PATH1*. The first phase of *PATH1* is as follows. Initialize λ_1 to 0 and λ_2 to ∞. Let $f(n)$ be the largest power of 2 no larger than $\log n$. Form an $f(n)$-partition of path P. For each subpath P_j, form sorted matrix $M(P_j)$, giving a set $\mathcal{M}_1$ of $\lceil n/f(n) \rceil$ sorted matrices, each (except possibly the last) of dimension $f(n) \times f(n)$. Call algorithm *MSEARCH* with the set $\mathcal{M}_1$ of matrices and a stopping count of $n/(f(n))^2$, using feasibility test *FTEST0*. At most $n/(f(n))^2$ active values remain from $\mathcal{M}_1$. For each subpath P_j that contains no active values, compute

the functions $ncut(\cdot)$ and $rmdr(\cdot)$, using algorithm *DIGEST1*, to be described shortly.

The second phase of *PATH1* is as follows. Let $M(P)$ be the $n \times n$ sorted matrix representing all sums on the path P. Run algorithm *MSEARCH* on the set $\{M(P)\}$ with stopping count 0, using feasibility test *FTEST1*. At the termination of *MSEARCH*, $\lambda^* = \lambda_1$. This completes the description of *PATH1*.

We illustrate *PATH1* on path P in Fig. 2.1, with $k = 3$. (This is the same example that we discussed near the end of the previous section.) For path P, $n = 8$ and thus $f(n) = 2$. The set $\mathcal{M}_1$ is shown in Fig. 3.1. Procedure *MSEARCH* is called on $\mathcal{M}_1$ with stopping count $n/(f(n)^2) = 2$. These are first quartered, giving sixteen 1×1 cells, and then $R = \{17, 6, 11, 0, 11, 9, 2, 0, 16, 1, 15, 0, 15, 7, 8, 0\}$. The median of these is $(7 + 8)/2 = 7.5$. For $\lambda = 7.5$, 3 cuts are required, so λ_1 is reset to 7.5, and appropriate values are discarded, leaving $R = \{17, 11, 11, 9, 16, 15, 15, 8\}$. The median of these is $(11 + 15)/2 = 13$. For $\lambda = 13$, 2 cuts are required, so λ_2 is reset to 13, and appropriate values are discarded, leaving $R = \{11, 11, 9, 8\}$. The median of these is $(9 + 11)/2 = 10$. For $\lambda = 10$, 3 cuts are required, so λ_1 is reset to 10, and 9 and 8 are discarded, leaving $R = \{11, 11\}$. This completes one iteration of the outer repeat-loop of *MSEARCH*. Since the number of values remaining is less than or equal to the stopping count, *MSEARCH* terminates with $\lambda_1 = 10$ and $\lambda_2 = 13$. There are two subpaths with no active values, namely the subpath with the 5-th and 6-th vertices of P, and the one with the 7-th and 8-th vertices of P. Computing *ncut* and *rmdr* values gives $ncut(5) = 1$, $rmdr(5) = 0$, $ncut(6) = 1$, $rmdr(6) = 0$, $ncut(7) = 1$, $rmdr(7) = 0$, $ncut(8) = 0$, $rmdr(8) = 8$. The execution of the second phase of *PATH1* follows the example near the end of the preceding section, except that *FTEST1* is used, and whenever a test value λ does not satisfy $\lambda_1 < \lambda < \lambda_2$, the test need not be performed.

We next describe procedure *DIGEST1* for the max-min problem, which computes

$ncut(l)$ and $rmdr(l)$ for all vertices v_l in subpath P_j. The approach is simple dynamic programming, for which we give the dynamic programming equations. Let $w(u,v) = A_v - A_{u-1}$, the sum of the weights of vertices from vertex u to vertex v. Let l range from $t_j + 1$ down to f_j. If $w(l,t_j) < \lambda_2$, then $ncut(l) = 0$ and $rmdr(l) = w(l,t_j)$. Otherwise, $ncut(l) = 1 + ncut(h+1)$ and $rmdr(l) = rmdr(h+1)$, where h is the smallest index such that $w(l,h) \geq \lambda_2$. The dynamic programming equations can be implemented using a pass over the subpath from the last vertex to the first with pointers l and h monotonically nonincreasing.

Lemma 3.1. Let P_j be a subpath that has no corresponding active values. Then *DIGEST1* computes $ncut(l)$ and $rmdr(l)$ for all vertices v_l in P_j in linear time.

Proof. For each of r values of h, a constant amount of work is done. The remaining work can be apportioned as constant for each of $O(r)$ values of l. □

We next describe feasibility test *FTEST1* that can be performed using the *ncut* and *rmdr* values. The key idea is to use binary search to find the first cut in each subpath, and then, if the subpath has no active values, to index into the data structures to find the number of cuts remaining in the subpath, and the size of the last component. Let λ be the value to be tested, with $\lambda_1 < \lambda < \lambda_2$.

```
proc FTEST1 (path P, integer k, real λ)
numcut ← −1; remainder ← 0
for each subpath P_j starting with the first do
    if remainder + w(f_j, t_j) < λ
    then remainder ← remainder + w(f_j, t_j)
    else Binary search to find the smallest l such that w(f_j, v_l) + remainder ≥ λ.
        if v_l is t_j
        then numcut ← numcut + 1; remainder ← 0
        else if there are no active values for P_j
            then numcut ← numcut + 1 + ncut(l + 1); remainder ← rmdr(l + 1)
            else
                Scan P_j starting with v_{l+1} to compute k′ and r′, the values
```

of $ncut(l+1)$ and $rmdr(l+1)$, resp., as though $\lambda_2 = \lambda$.
$numcut \leftarrow numcut + k'$; $remainder \leftarrow r'$
endif
endif
endif
endfor
if $numcut \geq k$ **then** λ is feasible ($\lambda \leq \lambda^*$) **else** λ is not feasible ($\lambda > \lambda^*$)
endproc

Lemma 3.2. Let P be a path of n vertices partitioned into $\lceil n/r \rceil$ subpaths, each of size at most r. Let all but at most n/r^2 subpaths have the functions $ncut(\cdot)$ and $rmdr(\cdot)$ computed for all vertices on them. Algorithm *FTEST1* determines the feasibility of a test value in $O((n/r)\log r)$ time.

Proof. There are $\lceil n/r \rceil$ subpaths to be examined. Consider the examination of subpath P_j. The search for v_l uses $O(\log r)$ time. If there are no active values for P_j, then all other operations performed in examining P_j take constant time. There are $\Theta(n/r)$ such paths, which take $O((n/r)\log r)$ time in total. If there are active values for P_j, then examining the path takes $O(r)$ time. But there are only $O(n/r^2)$ subpaths with active values, which take $O(n/r)$ time in total. □

Theorem 3.3. Algorithm *PATH1* solves the max-min k-partitioning problem on a path of n vertices in $O(n \log\log n)$ time.

Proof. By Theorem 2.1, the number of test values produced on the first call to *MSEARCH* is $O(\log\log n)$, and these are produced in $O(n)$ time. Since each feasibility test takes $O(n)$ time, the total time for feasibility testing is $O(n \log\log n)$. By Lemma 3.1 the total time to compute the *ncut* and *rmdr* values for $\Theta(n/\log n)$ subpaths is $O(n)$. By Theorem 2.1, the second call to *MSEARCH* produces $O(\log n)$ test values in $O(n)$ time. By Lemma 3.2, each call to feasibility test *FTEST1* takes $O(n \log\log n/\log n)$, or $O(n \log\log n)$ for all tests. □

Applying the above strategy with more than two phases produces our second algorithm, *PATH2*, which runs in $O(n \log^* n)$ time. The idea is to start with the straightforward feasibility test, and bootstrap our computation by repeatedly constructing data structures for ever faster feasibility tests. In each phase, we generate these data structures as we search among sets of sorted matrices that represent larger and larger subpaths. Initially all values are active, so that $ncut(\cdot)$ and $rmdr(\cdot)$ are uninitialized, and *FTEST1* will perform initially much like *FTEST0*. We call *PATH2* only when $n \geq 16$.

proc *PATH2* (**path** P, **integer** k)
Let $f(n)$ be the largest power of 2 no larger than $(\log n)^2$.
Let lev be the smallest nonnegative integer such that $f^{lev}(n) = 16$.
$\lambda_1 \leftarrow 0$; $\lambda_2 \leftarrow \infty$
for each value of i from lev down to 0 **do**
    $r_i \leftarrow f^i(n)$
    Form an r_i-partition of P.
    For each subpath P_j, form a sorted matrix for P_j.
    Let $\mathcal{M}_i$ be the resulting set of sorted matrices.
    Call *MSEARCH* on $\mathcal{M}_i$, with stopping count n/r_i^2, using *FTEST1*.
    Identify the at most n/r_i^2 active values between λ_1 and λ_2.
    For each subpath P_j with no active values, call *DIGEST1*.
endfor
$\lambda^* \leftarrow \lambda_1$
endproc

Theorem 3.4. Algorithm *PATH2* solves the max-min k-partitioning problem on a path of n vertices in $O(n \log^* n)$ time.

Proof. There are $lev + 1$ phases, where lev is $O(\log^* n)$. Each call to *MSEARCH* with matrices of size $r_i \times r_i$ uses $O(n)$ time and generates $O(\log r_i)$ test values. Performing all feasibility tests on phase lev uses $O(n)$ time, and on phase $i < lev$ uses $O((n/r_{i+1}) \log r_{i+1} \log r_i)$ time. Since r_{i+1} is $\Theta((\log r_i)^2)$, the total time for all feasibility tests is $O(n)$. The time to compute $ncut(\cdot)$ and $rmdr(\cdot)$ values at the end of the phase is also $O(n)$. Thus the time for each of the $O(\log^* n)$ phases is $O(n)$. □

We now discuss our third algorithm *PATH3*, which uses linear time. In *PATH2*, $O(n)$ time per phase is spent on each of two activities: computing $ncut(\cdot)$ and $rmdr(\cdot)$ values, and producing test values. We modify *PATH2* to use $O(n)$ time in total on these activities. First, we use the $ncut(\cdot)$ and $rmdr(\cdot)$ data structures already generated to generate the $ncut(\cdot)$ and $rmdr(\cdot)$ data structures on the next phase. Second, we call *MSEARCH* on sets of trimmed submatrices to produce our search values more efficiently.

We first describe our routine *DIGEST2*, which updates the arrays containing $ncut(\cdot)$ and $rmdr(\cdot)$ more efficiently than *DIGEST1*. When *MSEARCH* terminates on phase i, all of the at most n/r_i^2 active values fall between λ_1 and λ_2. For each subpath P_j with no active values, we want to determine *ncut* and *rmdr* values. We shall avoid computing information that we do not need. We shall compute $ncut(l)$ and $rmdr(l)$ for each vertex v_l with $w(f_j, l) \leq \lambda_2$. To assist in computing these data structures, we shall also keep track of $next(l)$, the index of the vertex v_q such that $rmdr(l) = w(q, t_j)$. Before the searching phases begin, initialize $ncut(l)$ to be 0 and $next(l)$ to be l for each vertex v_l. We do not maintain $rmdr(\cdot)$ explicitly for any vertex v_l such that $ncut(l) = 0$.

Let P_j be a subpath in the r_i-partition. We now describe how to generate the appropriate values for P_j, given the *ncut*, *rmdr* and *next* values for the subpaths of P_j that are in the r_{i+1}-partition. Our approach is to identify those vertices of P_j that are within λ^* of f_j and whose *ncut* value will be increased on the phase. These vertices are inserted into a queue. We then sweep through P_j, examining the vertices in the queue, and inserting additional vertices in the queue if they follow a cut inferred by a vertex in the queue. We save the vertices examined on a stack, so that *ncut*, *rmdr* and *next* values can be computed for the vertices examined, in reverse order, once the sweep through P_j is complete. To make the sweep efficient, the entries in the queue

should be in increasing order. However, when we sweep through a subpath of P_j, the entries generated for the queue are in either increasing order, or in rotated increasing order (i.e., some rotation of the whole list gives a list in increasing order). Why the entries to the queue are generated in this order is explained in the proof of Lemma 3.5. The queue entries should be rearranged periodically, so that they are considered in sorted order.

We now restate the above strategy in detail. Recall that f_j and t_j are the indices of the first and last vertices, resp., in P_j.

proc *DIGEST2* (**subpath** P_j) /* P_j contains no active values */
if $w(f_j, t_j) \geq \lambda_2$
then
    Initialize a queue with a mark.
    for each vertex v_l in turn **while** $w(f_j, l-1) < \lambda_2$ and $w(next(l), t_j) \geq \lambda_2$ **do**
        Insert l into the queue.
    end
    Initialize a stack to be empty.
    while the queue contains more than just the mark **do**
        if the mark is at the front of the queue
        then
            Put the queue in sorted order, with the mark at the end.
            Set q to be the largest value in the queue.
        endif
        Extract l from the queue.
        if $next(l) = l$
        then while $w(l, q-1) < \lambda_2$ **do** $q \leftarrow q+1$ **endwhile**
        else $q \leftarrow next(l)$
        endif
        Push the pair (l, q) onto the stack.
        if $w(q, t_j) < \lambda_2$ **then** $rmdr(q) \leftarrow w(q, t_j)$ **endif**
        If q is not in the queue and $w(q, t_j) \geq \lambda_2$, then insert q into the queue.
    endwhile
    while the stack is not empty **do**
        Pop a pair (l, q) from the stack.
        $rmdr(l) \leftarrow rmdr(q)$; $next(l) \leftarrow next(q)$.
        if $ncut(l) = 0$ **then** $ncut(l) \leftarrow 1 + ncut(q)$ **else** $ncut(l) \leftarrow ncut(l) + ncut(q)$
    endwhile
endif

endproc

Lemma 3.5. Over all phases, the time for all calls to *DIGEST2* is $O(n)$.

Proof. First, we bound the time to scan by increments of 1 for a vertex v_q given vertex v_l when $ncut(l) = 0$. A vertex v_l is in the queue only if the total weight to the end of the path is at least λ^*. After the scan, $ncut(l) > 0$ and $next(l) \geq q$, so that the vertices scanned when $ncut(l) = 0$ are not scanned again. Thus in the linear scans over vertices when $ncut(l) = 0$, each vertex v_q is scanned at most once. Thus the total charge for this is $O(n)$ over all phases.

Next, we bound the time to handle vertices inserted into the queue after the queue initialization as follows. Any vertex is inserted into the queue after the queue initialization at most once. This follows since the $next(\cdot)$ function is set to point beyond this vertex for the next phase. Thus the total cost of handling such vertices is $O(n)$ over all phases.

Finally, we bound the cost of handling a vertex inserted into the queue during queue initialization on a phase. Consider a vertex v_l inserted into the queue during queue initialization. On the first phase when this happens, charge 1 to the vertex. Over all phases each vertex can be so charged at most once, yielding a total cost of $O(n)$ overall. Since v_l is inserted into the queue during queue initialization, there is at least one cut placed on the subpath containing v_l in this phase. Consider any subsequent phase i on which the vertex v_l is inserted into the queue during queue initialization. Let P_{j_l} be the subpath of P_j that contains v_l. No subpath of P_j following P_{j_l} contains a vertex inserted into the queue during queue initialization on phase i, since the total weight of vertices on P_{j_l} is at least λ^*. Furthermore, the subpaths of P_j (if any) preceding P_{j_l} have total weight less than λ^*, since v_l was inserted into the queue during queue initialization. Hence this is the first phase that vertices on such subpaths (if any) are inserted into the queue during queue initialization. Thus at

most r_{i+1} vertices, all from subpath P_{j_l}, are the only vertices from the r_i vertices of subpath P_j that are inserted into the queue during queue initialization for a second or greater time on phase i. Thus the total charge for phase i is $(n/r_i)r_{i+1}$ for handling these vertices. Over all phases this latter cost is $O(n)$.

We bound the time for all queue rearrangements as follows. If l and l' are two entries in the portion of the queue after the mark, then $ncut(l)$ and $ncut(l')$ differ by at most one. In fact as one scans the entries in the queue after the mark, then the *ncut* value stays constant while the *rmdr* value is nonincreasing, until at some index (if it exists), the *ncut* value decreases by 1, the *rmdr* value increases, and then for the remaining entries the *ncut* value stays constant while the *rmdr* value is once again nonincreasing. Thus the queue at the time of reorganization is a rotated sorted list. It takes time proportional to its length to reorganize a rotated sorted list. Because of the manner in which marks are handled, a vertex participates in a queue reorganization once for each time it is inserted into the queue. By the above arguments, the total number of insertions into the queue over all phases in $O(n)$. Thus the time for all queue reorganizations is $O(n)$. □

We next discuss how to generate sets of submatrices that are supplied as arguments to *MSEARCH*, so that the total time of *MSEARCH* over all phases is $O(n)$. If a value in a sorted matrix is known to be too small to be λ^*, then we would like not to include it in the set supplied to *MSEARCH*. If a value is known to be too large, then on subsequent phases, it and any values that can be deduced by matrix position to be larger than it should be ignored.

On phase i, we consider a set $\mathcal{M}_i^{\mathcal{I}}$ of submatrices of $M(P)$. To identify small and large values, we consider a set of thin matrices generated as follows. For each submatrix in $\mathcal{M}_i^{\mathcal{I}}$, we take the first row as a thin matrix, and the first column as a thin matrix. We call *MSEARCH* on this resulting set $\mathcal{T}_i$ of thin matrices, with stopping

count n/r_i. When *MSEARCH* returns, only $\lfloor n/r_i \rfloor$ values in the thin matrices are still active. The other values are thus characterized as too small to be λ^* or too large to be λ^*.

We trim (logically) each submatrix in $\mathcal{M}_i^{\mathcal{I}}$ to remove any row whose first element is too small, and any column whose first element is too small. We then call *MSEARCH* to search in the set $\mathcal{M}_i^{\mathcal{T}}$ of resulting trimmed submatrices. We shall also include the set $\mathcal{A}_i$ of elements remaining as active from the previous phase. We explain shortly why these are included.

We cannot trim a row or column whose first element is too large, since this might result in λ^* being discarded. Thus we cannot avoid leaving these rows or columns in. We make sure however that these large values are not considered in subsequent phases. On subsequent phases we could delete the corresponding rows and columns from $M(P)$ and then compact $M(P)$, but this appears to be too expensive to do. Instead, we extract from $M(P)$ a set $\mathcal{M}_{i-1}$ of submatrices to be considered on phase $i-1$. None of these submatrices contain values that are in the same row or column as a large value, or to the left and above, or to the right and below, a large value. There are $O(n/r_i)$ such submatrices, which are not in general square. While subpaths do not change, each is represented by a set of submatrices that do not in general contain all sums associated with the subpath. Also, values that are still active must not be lost from consideration, since they might be in a row or column that is deleted. To prevent losing such values, any value still active at the end of phase i is brought along to the next phase, and the set $\mathcal{A}_{i-1}$ of these values needs to be included in some of the calls to *MSEARCH*.

We note that there is a natural order on submatrices in $\mathcal{M}_i$. Let M' and M'' be distinct submatrices of $\mathcal{M}_i$. Any row of $M(P)$ does not contain elements of both M' and M'', and similarly for columns. Without loss of generality assume that an element

from M' is in a higher-indexed row of $M(P)$ than any element from M''. Then every element of M'' is in a column of $M(P)$ of higher index than any element in M'. We assume that the submatrices of $\mathcal{M}_i$ are maintained in order from lower left of $M(P)$ to upper right of $M(P)$. At the beginning of phase i the elements of the matrices in $\mathcal{M}_i$ are intersected with the elements in the matrices of the set $\mathcal{M}_i^{\mathcal{P}}$ of matrices for the subpaths in the r_i-partition of P to give a set of elements stored in submatrices $\mathcal{M}_i^{\mathcal{I}}$. (This intersection should not be carried out element by element. Instead, it is sufficient to identify each maximal submatrix of $M(P)$ that is a submatrix of a matrix in $\mathcal{M}_i$ and also a submatrix of a matrix in $\mathcal{M}_i^{\mathcal{P}}$. We denote this operation by the symbol $\circledcirc$.) We now summarize our algorithm *PATH3*, which we call only when $n \geq 16$.

proc *PATH3* (**path** P, **integer** k)
Let $f(n)$ be the largest power of 2 no larger than $(\log n)^2$.
Let lev be the smallest nonnegative integer such that $f^{lev}(n) = 16$.
$\mathcal{M}_{lev} \leftarrow \{M(P)\}$
$\mathcal{A}_{lev} \leftarrow \emptyset$
for each vertex v_l **do** $ncut(l) \leftarrow 0$; $next(l) \leftarrow l$ **endfor**
$\lambda_1 \leftarrow 0$; $\lambda_2 \leftarrow \infty$.
for each value of i from lev down to 0 **do**
    $r_i \leftarrow f^i(n)$
    Form the r_i-partition of P, and the set $\mathcal{M}_i^{\mathcal{P}}$ of submatrices of $M(P)$.
    $\mathcal{M}_i^{\mathcal{I}} \leftarrow \mathcal{M}_i \circledcirc \mathcal{M}_i^{\mathcal{P}}$.
    Form the set $\mathcal{T}_i$ of thin matrices for $\mathcal{M}_i^{\mathcal{I}}$, by taking the first row
        and the first column from each matrix in $\mathcal{M}_i^{\mathcal{I}}$.
    Call *MSEARCH* on $\mathcal{T}_i$, with stopping count n/r_i.
    Initialize the set of trimmed submatrices $\mathcal{M}_i^{\mathcal{T}}$ to be empty.
    for each submatrix M' in $\mathcal{M}_i^{\mathcal{I}}$ **do**
        Insert a corresponding matrix M'' into $\mathcal{M}_i^{\mathcal{T}}$, where M'' is M' with
            every row or column that is headed by a small value deleted.
    endfor
    Call *MSEARCH* on $\mathcal{M}_i^{\mathcal{T}} \cup \mathcal{A}_i$ with stopping count n/r_i^2.
    Let $\mathcal{A}_{i-1}$ be the elements remaining upon the return from *MSEARCH*.
    Initialize $\mathcal{M}_{i-1}$ to be $\mathcal{M}_i$.
    for each submatrix M' in $\mathcal{M}_{i-1}$ that contains a large value **do**
        Delete any element in the same row or column as a large value.

Delete any element that is to the left and above a large value,
or to the right and below a large value.
endfor
For each subpath P_j, call *DIGEST2*.
endfor
$\lambda^* \leftarrow \lambda_1$
endproc

We illustrate our approach in Figures 3.2 and 3.3. Suppose that $n = 16$, so that $M(P)$ is a 16×16 matrix. We show this matrix symbolically, with an asterisk in each element position. We start with $\mathcal{M}_{lev}$ containing the one matrix $M(P)$. This is shown in Fig. 3.2(a) by circling each asterisk. For the sake of illustration, suppose that $(\log n)^2$ were equal to 4. (Clearly, this is not true, but choosing a value of n which is a power of 2 and for which $f(n) < n$ makes the example a bit big.) Then our path is partitioned into 4 subpaths. The four submatrices in $\mathcal{M}_{lev}^{\mathcal{P}}$, corresponding to the four subpaths are shown in bold in Fig. 3.2(b). The elementwise intersection of $\mathcal{M}_{lev}$ and $\mathcal{M}_{lev}^{\mathcal{P}}$ gives a set $\mathcal{M}_{lev}^{\mathcal{I}}$ that is the same as $\mathcal{M}_{lev}^{\mathcal{P}}$. The set of elements from the members of $\mathcal{T}_{lev}$, consisting of the first row and first column of each matrix in $\mathcal{M}_{lev}^{\mathcal{I}}$ is shown in Fig. 3.2(c). The characterization of all but at most $n/4$ of these elements is shown in Fig. 3.2(d). The elements in the thin matrices that are not characterized as large (**L**) or small (**S**) are indicated by a question mark. The set $\mathcal{M}_{lev}^{\mathcal{T}}$ of trimmed submatrices is shown in Fig. 3.3(a). The set $\mathcal{M}_{lev-1}$ of submatrices for phase $lev - 1$ is shown in Fig. 3.3(b).

Theorem 3.6. Algorithm *PATH3* solves the max-min k-partitioning problem on a path of n vertices in $O(n)$ time.

Proof. We analyze the time for manipulating and searching matrices as follows. We start phase i with a set $\mathcal{M}_i$ of $O(n/r_{i+1})$ submatrices of $M(P)$ and a set $\mathcal{A}_i$ of $O(n/r_{i+1}^2)$ elements. Since there are $O(n/r_{i+1})$ submatrices in $\mathcal{M}_i$, the time to

form $\mathcal{M}_i^{\mathcal{I}}$ is $O(n/r_{i+1})$. There are $O(n/r_{i+1})$ of these submatrices. Thus there are $O(n/r_{i+1})$ thin matrices, and the time to form the set $\mathcal{T}_i$ of thin matrices is $O(n/r_{i+1})$. By Theorem 2.1, the call to *MSEARCH* on $\mathcal{T}_i$, uses $O((n/r_{i+1}) \log r_i)$ time to produce $O(\log r_i)$ search values. Given the values remaining from this search, it takes $O(n/r_{i+1})$ time to generate the trimmed submatrices $\mathcal{M}_i^{\mathcal{T}}$. Let the l-th trimmed submatrix have dimension $n_l \times m_l$. Let $large(i)$ be the number of large values produced by searching in the set of thin matrices on phase i. Since no two such large values can appear in the same row or in the same column of $M(P)$, $\sum_i large(i) \leq 2n$. A bound on $\sum_l n_l + m_l$ is $2n/r_i + large(i)$. The most time is used in producing search values when as many trimmed matrices as possible are of dimensions $r_i \times r_i$. The number of these is $O((n/r_i + large(i))/r_i)$. Thus by Theorem 2.1, the call to *MSEARCH* on $\mathcal{M}_i^{\mathcal{T}} \cup \mathcal{A}_i$ uses $O((n/r_i + large(i)) + n/r_{i+1})$ time to produce $O(\log r_i)$ search values. The time to form $\mathcal{M}_{i-1}$ using the large values resulting from the call to *MSEARCH* on the thin matrices is $O(n/r_{i+1})$. Thus the time to manipulate matrices and produce search values on phase i is $O((n/r_{i+1}) \log r_i + large(i))$, which is $O(n/\log r_i + large(i))$. This sums to $O(n)$ over all phases. By Lemma 3.2, the time to test a value for feasibility is $O((n/r_{i+1}) \log r_{i+1})$. Thus all feasibility tests on phase i uses time that is $O((n/r_{i+1}) \log r_{i+1} \log r_i)$, which is $O((n/\log r_i) \log\log r_i)$. This sums to $O(n)$ over all phases.

Finally, by Lemma 3.5 the time to generate the functions *ncut* and *rmdr* over all phases is $O(n)$. □

We briefly survey the differences needed to solve the min-max problem. Let $ncut(l)$ be the minimum number of cuts on the subpath in P_j from v_l to t_j such that each connected component in this subpath, has weight at most λ_1, and $rmdr(l)$ is the minimum weight possible for the last component, given that the number of edges cut in P_j is $ncut(l)$. Procedure *DIGEST1* can similarly be described by a set of dynamic

programming equations: If $w(l, t_j) < \lambda_1$, then $ncut(l) = 0$ and $rmdr(l) = w(l, t_j)$. Otherwise, $ncut(l) = 1 + ncut(h + 1)$ and $rmdr(l) = rmdr(h + 1)$, where h is the largest index such that $w(l, h) \leq \lambda_1$. Procedure *FTEST1* is similar except that l is the largest index such that $w(f_j, v_l) + remainder \leq \lambda$, path P_j is scanned as though $\lambda_1 = \lambda$, 1 is subtracted after completion of the for-loop if $remainder = 0$, and λ is feasible if and only if $numcut \leq k$. Algorithms *PATH1*, *PATH2*, and *PATH3* are similar, except that at termination, $\lambda^* = \lambda_2$. Procedure *DIGEST2* is similar, except that any index l inserted into the queue satisfies $w(f_j, l) \leq \lambda_1$ and $w(next(l), t_j) \geq \lambda_1$, $low(j)$ is the largest index such that $w(f_j, low(j)) \leq \lambda_1$, $high(j)$ is the smallest index such that $w(high(j), t_j) \leq \lambda_1$, q is incremented while $w(l, q) \leq \lambda_1$, and the other tests should have λ_1 rather than λ_2.

Theorem 3.7. The min-max k-partitioning problem can be solved on a path of n vertices in $O(n)$ time.

Proof. The above changes will not affect the asymptotic running time of algorithm *PATH3*. □

4. Partitioning a Tree

In this section we present three algorithms to perform parametric search on a tree of n vertices. The structure of this section is similar to that of the previous section, but concentrates on the additional ideas needed to deal with a tree. The first algorithm uses two phases of search to achieve $O(n(\log \log n)^2)$ time. The first phase gathers information with which subsequent feasibility tests can be performed in $o(n)$ time. The second phase then completes the parametric search using this faster feasibility testing. The second algorithm gathers information on more than one phase, and takes $O(n \log^* n)$ time. The third algorithm uses a variety of refinements to reduce the time down to $O(n)$. Throughout this section we assume that the tree

is rooted, that paths in the tree run from descendants to ancestors and are indexed in this order within any such designated path.

We describe several key ideas for fast feasibility testing in a tree. A feasibility test will not be efficient if there are too many paths in the edge-path-partition. Thus we may need to prune the tree to reduce the number of leaves, and hence the number of paths in the edge-path-partition of the resulting tree. Define a *vertex-path-partition* of a tree T as a set of paths of tree T formed as follows. Find an edge-path-partition of T, and delete the top vertex from each path except the path containing the root of T. For each path, let the bottom vertex of the path with respect to the tree be its first vertex. For a tree T and desired subpath length r we define the *path-tree* T_c as follows. Form a vertex-path-partition of T, and form an r-partition of each path in the vertex-path-partition. Let T_c be a tree in which each node represents a subpath formed during the r-partition, and node u in T_c is a parent of node w in T_c if the highest vertex in the subpath for w is a child of the lowest vertex in the subpath for u.

As an example, consider the vertex-weighted tree T in Fig. 4.1(a). (This is the same tree that was used to illustrate the edge-path-partition in Fig. 2.1.) The vertex-path-partition for T is shown in Fig. 4.1(b). The r-partition for each path in the vertex-path-partition is shown in Fig. 4.2(a), with $r = 2$. Note that there is only one path containing more than 2 vertices in the vertex-path-partition, in the lower right corner of Fig. 4.1(b). This path is split accordingly. The path-tree T_c for T with $r = 2$ is shown in Fig. 4.2(b). Here some of the nodes (those representing subpaths that contain more than one vertex) are depicted as large "blobs" so that the correspondence to Fig. 4.2(a) is clearer.

We now describe our first algorithm, *TREE1*, for partitioning a tree. The first phase reduces the problem of partitioning a tree to the problem of partitioning a tree

with fewer than $2n/(\log n)^2$ leaves, and then generates a data structure for a fast feasibility test. Note that when there are at least $2n/(\log n)^2$ leaves, at least half of the leaf-paths are of length at most $(\log n)^2$. Initializing T_c to T indicates that we initially view each subpath in T' to be of length 1.

proc *TREE1* (**tree** T , **integer** k)
$\lambda_1 \leftarrow 0$; $\lambda_2 \leftarrow \infty$
$T' \leftarrow T$; $k' \leftarrow k$; $T_c \leftarrow T$
/* phase 1 */
$r \leftarrow \lfloor (\log n)^2 \rfloor$
Form an edge-path-partition of T'.
while there are at least $2n/r$ leaves in T' **do**
    Let $\mathcal{L}$ be the set of sorted matrices for leaf-paths of length at most r.
    Let n' be the number of vertices on these leaf-paths.
    Call *MSEARCH* on $\mathcal{L}$ with stopping count $n'/(2r)$, using *FTEST2* for T_c and k.
    for each leaf-path with no active values **do**
        Identify the cuts on the path, and subtract their number from k'.
        Add to the top vertex the weight of the other vertices in its component.
        Remove the leaf-path from T' and modify the edge-path-partition of T'.
    endfor
endwhile
Form the vertex-path-partition of T',
    and form an r-partition of each path in the vertex-path-partition.
Let $\mathcal{M}$ be the set of sorted matrices for the resulting subpaths.
Call *MSEARCH* on $\mathcal{M}$, with stopping count n/r^2, using *FTEST2* for T_c and k.
For each subpath P_j in T' with no active values, call *DIGEST1*.
Reset T to a copy of T', reset k to be k', and update T_c for T using r.
/* phase 2 */
Form an edge-path-partition of T'.
while there is more than one leaf in T' **do**
    Let $\mathcal{M}$ be the set of sorted matrices for paths in the edge-path-partition.
    Call *MSEARCH* on $\mathcal{M}$ with stopping count 0, using *FTEST2* for T_c and k.
    for each leaf-path **do**
        Identify the cuts on the path, and subtract their number from k'.
        Add to the top vertex the weight of the other vertices in its component.
        Remove the leaf-path from T' and modify the edge-path-partition of T'.
    endfor
endwhile
$\lambda^* \leftarrow \lambda_1$
endproc

We next describe the sublinear feasibility test *FTEST2* for T_c and k. As before we assume that $\lambda_1 < \lambda < \lambda_2$. Perform a traversal of T_c, starting at the root. At node v, representing subpath P_j, do the following. Set $remainder(v)$ to 0 and set $numcut(v)$ to -1 if v is the root and to 0 otherwise. Then recursively explore each child w of v. For each child w, add $remainder(w)$ to $remainder(v)$ and add $numcut(w)$ to $numcut(v)$. Then handle the subpath represented by v by executing the body of the for-loop in *FTEST1*, but using $remainder(v)$ in place of $remainder$. This completes the description of the examination of a subpath. When the traversal returns to the root of T_c all subpaths have been examined. If $numcut(root) \geq k$, then λ is feasible ($\lambda \leq \lambda^*$), else λ is not feasible ($\lambda > \lambda^*$). This completes the description of *FTEST2*. Note that when all subpaths contain active values, then *FTEST2* reduces in general to the straightforward linear-time feasibility test.

Lemma 4.1. Let T be a tree of n vertices and at most n/r leaves, for some integer r. Let T be partitioned into at most $2\lceil n/r \rceil$ subpaths, each of size at most r. Let all but at most n/r^2 subpaths have no active values and have the functions $ncut(\cdot)$ and $rmdr(\cdot)$ computed for all vertices on them. Algorithm *FTEST2* correctly determines the feasibility of a test value in $O((n/r)\log r)$ time.

Proof. Since there are at most n/r leaves, there is a partition into at most $2\lceil n/r \rceil$ subpaths. As in the proof of Lemma 3.2, the examination of a subpath P_j with no active values will use $O(\log r)$ time. This totals $O((n/r)\log r)$ time for the $\Theta(n/r)$ such paths. If there are active values for P_j, then examining the path will take $O(r)$ time. For all of the $O(n/r^2)$ such subpaths this totals $O(n/r)$ time. □

Theorem 4.2. Algorithm *TREE1* solves the max-min k-partitioning problem on a tree of n vertices in $O(n(\log\log n)^2)$ time.

Proof. We first consider the generation of tree T'. Each call to *MSEARCH*

will take $O(n')$ time, produce $O(\log\log n)$ search values. Thus the total time to produce search values over all iterations will be $O(n)$. It will cost $O(n)$ time to test each search value. Let p be the number of leaf-paths before *MSEARCH* is called to prune the tree. Since $p \geq 2n/r$, the number of leaf-paths of length at most r is at most $p/2$. The number of active values at the termination of *MSEARCH* is at most $n'/(2r) \leq n/(2r) \leq p/4$. Thus there are at least $p/2 - p/4 = p/4$ leaf-paths with no active values on them. Since the removal of as few as two such paths can cause an interior vertex to become a leaf, the tree resulting at the end of an iteration will have at most 7/8 of the leaves of the tree at the beginning of the iteration. The number of iterations needed to reduce the tree to one with at most $2n/r$ leaves will be $O(\log r) = O(\log\log n)$. Thus the total time to generate T' will be $O(n(\log\log n)^2)$.

Once T has been reset to T', T_c can be generated in $O(n)$ time. By Lemma 4.1, any subsequent feasibility test will use $O((n/r)\log r) = O(n\log\log n/(\log n)^2)$ time. Each vertex in T_c will participate in the formation of a sorted matrix for just one call to *MSEARCH*. Thus the total time to produce test values in all calls to *MSEARCH* during the second phase is $O(n)$, and each call will produce $O(\log n)$ test values. Since the number of leaves is at least halved on each iteration of the while loop, the number of calls to *MSEARCH* is $O(\log n)$. Thus the number of values tested is at most $O((\log n)^2)$. Since these can each be tested in $O(n\log\log n/(\log n)^2)$ time, the total time for feasibility testing in the second phase is $O(n\log\log n)$. □

We next apply the above strategy recursively to yield an $O(n\log^* n)$-time algorithm *TREE2*. The idea is to start with the tree and a straightforward feasibility test, and bootstrap the computation by alternately pruning the tree to reduce the number of leaves (and hence the number of paths in an edge-path-partition), and then constructing data structures for an ever faster feasibility tests. Initially all values are active, so that functions $ncut(\cdot)$ and $rmdr(\cdot)$ are uninitialized. We call *TREE2* when

$n \geq 2^9$.

proc *TREE2* (**tree** T , **integer** k)
Let $f(n)$ be the largest power of 2 no larger than $(\log n)^3$.
Let lev be the smallest nonnegative integer such that $f^{lev}(n) = 2^9$.
$\lambda_1 \leftarrow 0$; $\lambda_2 \leftarrow \infty$
$T' \leftarrow T$; $k' \leftarrow k$; $T_c \leftarrow T$
for each value of i from lev down to 0 **do**
    Perform phase 1 of *TREE1*, replacing $\lfloor (\log n)^2 \rfloor$ by $f^i(n)$,
        r by r_i, $\mathcal{L}$ by $\mathcal{L}_i$, n' by n_i, and $\mathcal{M}$ by $\mathcal{M}_i$.
endfor
$\lambda^* \leftarrow \lambda_1$
endproc

Theorem 4.3. Algorithm *TREE2* solves the max-min k-partitioning problem on a path of n vertices in $O(n \log^* n)$ time.

Proof. For correctness, note that when $i = 0$, $r_i = n$ and the while-loop (taken from *TREE1*) will not terminate until T' is a simple path. Then the stopping count on *MSEARCH* in $\mathcal{M}_0$ will be $1/n$, so that no active values will remain at the end of phase 0.

We analyze the time as follows. There will be $lev+1$ phases, where lev is $O(\log^* n)$. One call to *MSEARCH* with matrices of size at most $r_i \times r_i$ will use $O(n')$ time and generate $O(\log r_i)$ test values, so that all such calls in phase i will use $O(n)$ time and produce $O((\log r_i)^2)$ test values. The time to compute $ncut(\cdot)$ and $rmdr(\cdot)$ values using *DIGEST1* at the end of the phase will also be $O(n)$. All other activities will use $O(n)$ time over all phases. In phase lev, performing all feasibility tests will use $O(n)$ time. Performing all feasibility tests on any other phase i will use $O((n/r_{i+1}) \log r_{i+1} (\log r_i)^2)$ time. Since r_{i+1} is $\Theta((\log r_i)^3)$, this time is $O(n)$ in total. The time to recompute $ncut(\cdot)$ and $rmdr(\cdot)$ values for vertices whose weight has changed is $O((n/r_{i+1}) \log r_{i+1})$, since $O(n/r_{i+1})$ nodes will have their weights change, and each recomputation involves a binary search on a subpath of length at most r_{i+1}.

□

We now discuss algorithm *TREE3*, which runs in linear time. In *TREE2*, $O(n)$ time per phase is spent on each of three activities: computing $ncut(\cdot)$ and $rmdr(\cdot)$ values, producing search values, and feasibility testing. We shall modify *TREE2* to use $O(n)$ time in total on these activities. In place of *DIGEST1* use *DIGEST2*. Since we do not know which paths will be removed, we lay out the tree in a certain way and then copy paths when it is determined that they should be "glued" together. As in *PATH3*, we call *MSEARCH* on sets of trimmed submatrices to produce our search values more efficiently. Constituting these matrices is trickier, because we do not know ahead of time by how much certain vertices will have their weight increased. This will force us to represent the paths in a special way.

As in *TREE2*, we will modify the current copy T' of the tree as we do feasibility testing on an old copy T of the tree. In *TREE2* we copy T' into T at the end of a phase, but this straightforward copying is too expensive for *TREE3*. Instead, we maintain two complete sets of data structures, one for T and one for T'. We make changes to the structures for T' as we proceed on a phase, and we make the identical changes to T at the end of the phase.

As *TREE3* progresses, the tree is pruned, and some of the edges in the edge-path-partition are deleted, while others are concatenated together. We rely in several places on a path in a vertex-path-partition being represented in consecutive locations of an array. To facilitate these activities, store the weights of the vertices of the tree initially in an array in the following order. Order the children of each vertex from left to right by nondecreasing height in the tree. Then list the vertices in postorder. This takes $O(n)$ time to set up, since the heights of all vertices can be found in $O(n)$ time, and then pairs containing parent and height can be ordered lexicographically in $O(n)$ time. Arrays that are parallel to the above weight array will be used to store

ncut, *rmdr*, and *next* values, as well as additional information used to compute the weight of specified subpaths. We discuss this additional information in due course.

When paths are removed from the tree, the storage of the tree within the array is then reorganized as follows. Let P be a leaf-path in the edge-path-partition of the current tree, and let t be the top vertex of P. We shall remove $P - t$. Let P' be the path whose bottom vertex is t. If t will have only one child remaining after removal of $P - t$, and this child was originally not the rightmost child of t, do the following. Let P'' be the other path whose top vertex is t. Copy the vertices of $P'' - t$ in order so that the top vertex of this path is in the location preceding the location of t in the array. Modify the weight of t by adding the remainder left from removing $P - t$. Also, *ncut*, *rmdr* and *next* values for all copied vertices should also be copied into the new locations in their arrays. The additional information should be copied and modified, as we discuss shortly.

Note that the bottom vertex of P'' may have been the parent of several vertices. When $P'' - t$ is moved, the children of its bottom vertex (and subtrees rooted at them) are not copied. It is simple to store in a location formerly assigned to the bottom vertex of P'' the current location of this vertex, and to also store a pointer back. If the path containing P'' is copied, then this pointer can be reset easily. When only one child of the bottom vertex of P'' remains, the corresponding path can be copied to in front of P''. We claim that the total time to perform all rearrangements will be $O(n)$. The time to copy each vertex and copy and adjust its accompanying information is constant. Because of the way in which the tree is stored in the array, at most one vertex will be copied from any array location.

We now discuss the additional information that needs to be maintained so that we can compute the weight of any specified subpath quickly. A first attempt might be to initialize the value A_v for every vertex v to be the sum of the weights of all descendants

of v (including itself). Note that for any path in the edge-path-partition, if w is an ancestor of v, w is not the top of the path, and v is not the bottom of the path, then $A_w - A_{v-1}$ will be the sum of all vertex weights from v up to w. Including the bottom or top vertex in the specified subpath involves adding the weight of whichever or both of these vertices into a difference of the same form. When the weight of a vertex t is changed, the value A_t can be changed, and the A_v values for each vertex v in a copied path can be changed. The time for changing values can be charged to that of copying the path. However, path $P'' - t$ should not be copied if the top vertex of $P'' - t$ was originally the rightmost child of t. In this case there is no copying to which the cost of changing A_v values could be charged. We next describe a mechanism that overcomes this problem.

Let a path in the edge-path-partition be called *light* if its weight is less than λ_2, and *heavy* otherwise. We handle light and heavy paths differently. For a light path, we keep track of the total weight in the path. For light paths, if the total weight of vertices in P is at most λ_1, then no value induced by P is in contention for being λ^*. However, there may be many light paths whose weights are greater than λ_1, and thus their weights, and the weights of certain of their subpaths, could be candidates for being λ^*. On phase i, there could be as many as $O(n/r_i)$ after the tree has been pruned. We reduce this number as follows. Form the set of weights of such paths. Then run *MSEARCH* on this set with stopping count n/r_i^2. This will reduce the number of light paths of weight greater than λ_1 to $O(n/r_i^2)$. These are then not included in the call to *MSEARCH* that allows better data structures to be constructed on phase i. Thus we tolerate $O(n/r_i^2)$ light paths that will continue to have active values.

We next discuss the representation of a heavy path. Each heavy path P is represented as a sequence of overlapping subpaths, each of weight at least λ_2. Each vertex

of P is in at most two overlapping subpaths. Each overlapping subpath, except the first, overlaps the previous overlapping subpath on vertices of total weight at least λ_2, and each overlapping subpath, except the last, overlaps the following overlapping subpath on vertices of total weight at least λ_2. Thus any sequence of vertices of weight at most λ_2 that is contained in the path is contained in one of its overlapping subpaths. For each overlapping subpath, we shall maintain A_v values for every vertex v except the top vertex of the overlapping subpath. A vertex v that is in two overlapping subpaths is allowed to have different A_v values for each overlapping subpath (though the A_v values within any given overlapping subpath are consistent).

Initially no path is heavy, since initially $\lambda_2 = \infty$. A heavy path P can be created in one of two ways.

1. Path P was light until the value of λ_2 was reduced by *MSEARCH*.

2. Path P is formed by the concatenation of two or more paths.

If a heavy path P is created in the first way, then represent P by two overlapping subpaths that are both copies of P, designating one as the first overlapping subpath and the other as the second. Set the A_v values of both of the overlapping subpaths so that $A_w - A_{v-1}$ is the correct value for any v and w, where w is not the top vertex, v is not the bottom vertex, and v is a descendant of w. If P is the concatenation of paths, all of which were light, then do the same thing.

Otherwise, path P is formed by the concatenation of paths, with at least one of them being heavy. Do the following to generate the representation for P. While there is a light path P' to be concatenated to a heavy path P'', combine the two as follows. If P' precedes P'', then extend the first overlapping subpath of P'' to incorporate P', adjusting the A_v values for the vertices of P'. If P' follows P'', then extend the last overlapping subpath of P'' and adjust the A_v values for the vertices of P'. This

completes the description of the concatenation of light with heavy paths. While there is more than one heavy path, concatenate adjacent heavy paths P' and P'' as follows. Assume that P' precedes P''. Combine the last overlapping subpath of P' with the first of P'', changing all the A_v values of the first subpath of P''. Note that a vertex can have its A_v value changed at most twice before it is in overlapping subpaths that are neither the first nor the last.

We use three arrays to hold the A_v values of the overlapping subpaths. When the last overlapping subpath of P' is combined with the first overlapping subpath of P'', either both are already in the same array, or there is an array that is empty in the corresponding positions of both subpaths, or the array containing one of them can accommodate the other. To keep track of which overlapping subpaths vertices are in, give each overlapping subpath a unique index, and maintain three additional arrays. For each vertex, the corresponding location in the additional array will contain the index of the overlapping subpath.

As an example, consider the portion of the tree shown in Fig. 4.3. The weights of the vertices are listed in postorder in the first array. Suppose that *TREE3* has been running for a while, so that currently $\lambda_1 = 4$ and $\lambda_2 = 16$. Of the five paths in the edge-path-partition that are shown in the figure, all but the path consisting of vertices with weights $13, 1, 1$ are heavy, and are thus represented in the remaining 2 pairs of arrays. Path $14, 3, 4$ is represented by the two overlapping subpaths of index 1 and 2, path $4, 8, 7$ is represented by the two overlapping subpaths of index 3 and 4, path $4, 11, 1$ is represented by the two overlapping subpaths of index 5 and 6, and path $1, 10, 6$ is represented by the two overlapping subpaths of index 7 and 8. In each case we use the overlapping subpath of smaller index as the first of the two overlapping subpaths. Values of the form A_v are set for each vertex in an overlapping subpath except the top vertex, and are set for convenience to be the sum of the weights of all

vertices in the subtree rooted at v. Any value in the arrays that is unknown because we have not provided a complete example is marked with an asterisk, and any array location whose value need not yet be set is marked with a dash. We have not shown the third pair of arrays, because they are not yet needed.

We continue the example by supposing that $\lambda = 14$ is tested for feasibility. Suppose that λ is not feasible, so that λ_2 is set to 14. The the path $13, 1, 1$ becomes a heavy path, and is represented by overlapping subpaths of index 9 and 10. Then suppose that $\lambda = 7$ is tested for feasibility. Suppose that λ is feasible, so that λ_1 is set to 7. Then a cut can be inferred in the leaf-path $14, 3, 4$, between 14 and 3. The two edges of the leaf-path can be removed, and the remaining weight of 3 can be added to the weight of the top vertex, giving weight 7. Then paths $7, 8, 7$ and $7, 11, 1$ are concatenated together, by combining overlapping subpaths 4 and 5 into overlapping subpath 11. Note that overlapping subpath 11 must be formed in the third pair of arrays. The resulting path has overlapping subpaths 3, 11, and 6. The state of the tree and the arrays at this point is shown in Fig. 4.4.

We continue the example further by testing $\lambda = 13$ for feasibility. Suppose that λ is not feasible, so that λ_2 is set to 13. Then a cut can be inferred in the leaf-path $13, 1, 1$, between 13 and 1. The two edges of the leaf-path can be removed, and the remaining weight of 1 can be added to the weight of the top vertex, giving weight 2. Then paths $7, 8, 7, 11, 2$ and $2, 10, 6$ are concatenated together, by combining overlapping subpaths 6 and 7 into overlapping subpath 12. Note that overlapping subpath 12 must be put in the first pair of arrays, extending forward from overlapping subpath 7. The resulting path has overlapping subpaths 3, 11, 12, and 8. The state of the tree and the arrays at this point is shown in Fig. 4.5.

To compute the weight of vertices from v up to w, where v and $w \neq v$ are in the same heavy path, do the following. Index into the array structure to find the at most

two overlapping subpaths containing v and the at most two overlapping subpaths containing $w-1$. Then test if one of the overlapping subpaths containing v also contains w. If so, compute the desired value as follows. If w is not the top vertex of this overlapping subpath, and v is not the bottom vertex, then compute $A_w - A_{v-1}$. If w is the top vertex and v is the bottom vertex, then add the weights of w and v to $A_{w-1} - A_v$. The other two cases are handled similarly. If no overlapping subpath contains both v and w, then the weight from v to w is too large, and it is sufficient to use λ_2 as the resulting value.

Given such a representation of a path, we discuss the formation of sorted matrices upon which to search. For each overlapping subpath, a sorted matrix will be induced. Note that since each vertex is in at most two overlapping subpaths, the total dimensions of these matrices will be only twice the dimensions of a single matrix representing the path.

We next discuss how to perform efficient feasibility testing, given our representation of paths. In a feasibility test, we would like to search any path P in time proportional to the logarithm of its length. This can be done if we maintain a second representation of the path as a red-black balanced search tree [**?**]. In this tree there will be a node x for every vertex u in P. Node x will contain two additional fields, $wt(x)$ and $ct(x)$. Field $wt(x)$ will contain the sum of weights of all vertices whose nodes are in the subtree rooted at x, and field $ct(x)$ will equal the number of nodes in the subtree rooted at x. With this tree it is easy to search for a vertex v in P such that v is the first vertex in P such that the sum of the weights to v is at least a certain value, and to determine as the same time the position of v in P. The search will take time proportional to the logarithm of the length of P. When two paths P' and P'' need to be concatenated together, the corresponding search trees can be merged in time proportional to the logarithm of the combined path length.

We now summarize our algorithm *TREE3*, which we call only when $n \geq 2^9$.

proc *TREE3* (**tree** T , **integer** k)
Let $f(n)$ be the largest power of 2 no larger than $(\log n)^3$.
Let lev be the smallest nonnegative integer such that $f^{lev}(n) = 2^9$.
$\mathcal{M}_{lev} \leftarrow \emptyset$
$\mathcal{A}_{lev} \leftarrow \emptyset$
$\lambda_1 \leftarrow 0$; $\lambda_2 \leftarrow \infty$
$T' \leftarrow T$; $k' \leftarrow k$; $T_c \leftarrow T$
for each value of i from lev down to 0 **do**
    $r_i \leftarrow f^i(n)$
    Form an edge-path-partition of T'.
    Perform the while-loop of *TREE2* to prune T'. (Refer to *TREE1*.)
    Update the representations of the paths of T' that changed.
    Call *MSEARCH* on the set of weights of light paths,
        with stopping count n/r_i^2, using *FTEST2* for T_c and k.
    Augment $\mathcal{M}_i$ to reflect all changes to the set of overlapping subpaths.
    Perform the body of the for-loop for a phase from *PATH3*,
        except that rather than *FTEST1*, *FTEST2* is used (for T_c and k),
        and *DIGEST2*, rather than *DIGEST1*, is applied to subpaths in T'.
    Update T to reflect the changes to T', reset k to be k', and update T_c for T.
endfor
$\lambda^* \leftarrow \lambda_1$
endproc

We start phase i with a set $\mathcal{M}_i$ of $O(n/r_{i+1})$ submatrices of matrices representing all paths in T, except for $O(n/r_{i+1}^2)$ light paths, and a set $\mathcal{A}_i$ of $O(n/r_{i+1}^2)$ elements. The value λ^* is either λ_1, or is one of the elements of $\mathcal{A}_i$, or is in one of the submatrices of $\mathcal{M}_i$, or is a sum in one of the light paths.

Theorem 4.4. Algorithm *TREE3* solves the max-min k-partitioning problem on a tree of n vertices in $O(n)$ time.

Proof. We analyze the time for *TREE3* as follows. Phase i starts with a set $\mathcal{M}_i$ of (n/r_{i+1}) matrices representing all paths, except for (n/r_{i+1}^2) light paths, and a set $\mathcal{A}_i$ of (n/r_{i+1}^2) elements. Pruning the tree will use $O(n')$ time to prune away $\Theta(n')$ vertices. This will be a total of $O(n)$ over all phases. As discussed earlier, copying

paths in the array that represents the tree will be accounted as a constant charge for each array location, which will be a total of $O(n)$ over all phases. Concatenating two paths by merging their balanced tree representations will cost time proportional to the logarithm of the length of the longer path. Since there will be $O(n/r_{i+1})$ concatenations, the worst case time for these will be $O((n/r_{i+1}) \log\log r_{i+1})$ on phase i. Modifying the path tree T_c and identifying paths will take $O(n/r_{i+1})$ time. Calling *MSEARCH* on the set of weights of light paths will take $O(n/r_i)$ time to produce $O(\log r_i)$ search values. The total time to test these will be $O((n/r_{i+1}) \log r_i \log r_{i+1})$, which is $O((n/r_i) \log\log r_i)$. Augmenting $\mathcal{P}_i$ and $\mathcal{M}_i$ will cost constant time for each vertex appearing for the first or second time in such a path. The remaining operations are analyzed as in the proof of Theorem 3.6. Thus the total time for *TREE3* over all phases is $O(n)$. □

Acknowledgement.

I would like to thank Yehoshua Perl for bringing several references to my attention. I would also like to thank the referees for their many helpful comments.